CS 676 Assignment 2

Ying Kit Hui (Graduate student)

February 28, 2022

Listings

1 Q1

Listing 1: Q1

```
%% paremeters initialization

sigma = 0.2;
r = 0.03;
T = 1;
K = 10;
S0 = 10;
D0 = 0.5;
t_d = T/3;

dt = 0.05;

%% Q1a

testlength = 9;

dtlist = (0.05 ./ 2.^(0:testlength))';

% convergence test

% no dividends, call option
rho = 0;
```

```
D0 = 0;
opttype = 0;

convertestcall = convergencetest(sigma, r, T, K, S0, opttype, rho,
    D0, t_d, dt, testlength);
convertestcall = array2table(convertestcall, ...
    'VariableNames',{'Delta t','Value','Change', 'Ratio'})

% no dividends, put option
rho = 0;
D0 = 0;
opttype = 1;

convertestput = convergencetest(sigma, r, T, K, S0, opttype, rho,
    D0, t_d, dt, testlength);
convertestput = array2table(convertestput, ...
    'VariableNames',{'Delta t','Value','Change', 'Ratio'})

[exactCall,exactPut] = blsprice(S0, K, r, T, sigma)

% We see that our binomial pricing value converges to the exact
    solution.
% Moreover, the ratio is roughly 2, indicating a linear convergence
     rate.

%% Q1b
sigma = 0.2;
r = 0.03;
T = 1;
K = 10;
S0 = 10;
D0 = 0.5;
t_d = T/3;

dt = 0.005;

% we add rho = 10% to show the trend of call and put values with
    respect to
% rho more clearly.
rholist = [0, 0.02, 0.04, 0.08,0.1];

dividendresult = zeros(length(rholist),2);

for i = 1:length(rholist)
    % call
```

```
    dividendresult(i,1) = mybin_div(sigma, r, T, K, S0,dt, 0, rholist(i), D0, t_d);
    % put
    dividendresult(i,2) = mybin_div(sigma, r, T, K, S0,dt, 1, rholist(i), D0, t_d);
end

dividendresult = array2table(dividendresult', 'VariableNames',...
    {'rho=0%', 'rho=2%','rho=4%','rho=8%','rho=10%'}, 'RowNames',{'Call','Put'})

% We see that call values decrease as dividend yield rho increases. Also,
% we see that put values increase as dividend yield rho decreases. It can
% intutively explained. A higher dividend yield rho means that the
% ex-dividend stock value will be smaller and hence the possible payoff
% (thinking in terms of expectation) for call option will decrease while
% the possible payoff for put option (thinking in terms of expectation)
% will increase.


function value = mybin_div(sigma, r, T, K, S0,dt, opttype, rho, D0, t_d)
% mybin_div return put/call options value with a discrete dividend
%   More explanation on the formula for dividend can be found in the assignment.
%
% Input:
% sigma: volatility of the underlying
% r: interest rate
% T: time of expiry
% K: strike price
% dt: size of timestep
% opttype: option type, 0 for call, otherwise for put
% rho: constant dividend rate
% D0: constant dividend floor
% t_d: dividend payment time

% up and down ratio in bin. model
u = exp(sigma * sqrt(dt));
d = exp(-sigma * sqrt(dt));
% risk neutral probability of having an up
```

```
q = (exp(r * dt) - d) / (u-d);

% find the index of the closest timestep that is larger than t_d
err = (0:dt:T) - t_d ;
idx = find(err>= 0, 1, 'first');
% so (idx - 1)th timestep (i.e. (idx-1)*dt) approximates t^+, note that we
% count the timestep from 0 to N

N = T / dt; % assume it is an integer

% vectorized approach, find payoff at final time T, denoted by W

% first: the stock values at final time T
%           values are arranged in desceding order
S = S0*d.^([N:-1:0]') .* u.^([0:N]');

% second: distinguish the case between call and put
if opttype == 0
    W = max(S - K,0);
else
    W = max(K - S,0);
end

% Backward iteration
for i = N:-1:1
    S = S(2:i+1,1) / u; % asset prices at (i-1)th timestep (i.e. (i-1)*dt)
                        % we count timestep from 0 to N
    if i == idx % when we are at the timestep approximation of t^+

        % note that the values of S right now actually refers to S(t^-)
        % since no deduction on S has been made yet and we know
        % S(t^+) = S(t^-) - D. Thus, based on dividend formula:
        div_value = max(rho * S, D0);

        % obtain option values at (i-1)th timestep, by risk neutral valuation
        W = exp(-r *dt) *(q* W(2:i+1) + (1-q)* W(1:i));

        % compute option values at t^- by interpolation
        W = dividend(W, S, div_value);
    else
        % obtain option values at (i-1)th timestep, by risk neutral valuation
```

```
        W = exp(-r *dt) *(q* W(2:i+1) + (1-q)* W(1:i));
    end
end

value = W(1);
end

function W_out = dividend( W_in, S, div_value)
% W_in: value of option at t^+
% S : asset prices (at t^-)
% div_value: discrete dollar dividend
%
% W_out: option value at t^-
%
% assume American constraint applied in caller

S_min = min(S);
S_ex = S - div_value; % ex dividend stock value
S_ex = max( S_ex, S_min); % make sure that
                            % dividend payment does
                            % not cause S < S_min
W_out = interp1( S, W_in, S_ex);
end

function testtable = convergencetest(sigma, r, T, K, S0, opttype, rho, D0, t_d, dt, testlength)
% convergencetest return a table that contains results of convergence test
% as described by Table 2 in assignment 2
%
% Similar input parameters as mybin_div, with the additional parameters dt
% and testlength.
% dt: initial size of testing timestep
% testlength: (the number of size of timesteps) - 1, note that each subsequent
% size of timestep is obtained by size of previous timestep divided by 2

dtlist = (dt ./ 2.^(0:testlength))';

testtable = zeros(testlength + 1, 4);

testtable(:,1) = dtlist;
for i = 1:testlength + 1
```

```
    testtable(i,2) = mybin_div(sigma, r, T, K, S0,dtlist(i),
        opttype, rho, D0, t_d);
end

testtable(2:end,3) = testtable(2:end,2) - testtable(1:end-1,2);
testtable(3:end,4) = testtable(2:end-1,3)./ testtable(3:end,3);
end
```

1.1 Result

convertestcall =

10×4 table

Delta t	Value	Change	Ratio
0.05	0.93149	0	0
0.025	0.9364	0.0049084	0
0.0125	0.93887	0.0024671	1.9896
0.00625	0.9401	0.0012366	1.995
0.003125	0.94072	0.00061904	1.9976
0.0015625	0.94103	0.00030971	1.9988
0.00078125	0.94119	0.0001549	1.9994
0.00039063	0.94126	7.7461e-05	1.9997
0.00019531	0.9413	3.8733e-05	1.9999
9.7656e-05	0.94132	1.9367e-05	1.9999

convertestput =

10×4 table

Delta t	Value	Change	Ratio
0.05	0.63595	0	0
0.025	0.64085	0.0049084	0
0.0125	0.64332	0.0024671	1.9896
0.00625	0.64456	0.0012366	1.995
0.003125	0.64518	0.00061904	1.9976
0.0015625	0.64549	0.00030971	1.9988
0.00078125	0.64564	0.0001549	1.9994
0.00039063	0.64572	7.7461e-05	1.9997
0.00019531	0.64576	3.8733e-05	1.9999
9.7656e-05	0.64578	1.9367e-05	1.9999

```
exactCall =

    0.9413


exactPut =

    0.6458


dividendresult =

  2×5 table

            rho=0%     rho=2%     rho=4%     rho=8%    rho=10%
            -------    -------    -------    ------    -------

    Call    0.68309    0.68309    0.68246    0.5266    0.44517
    Put     0.88254    0.88254    0.88257    1.0311     1.1496
```

1.2 Discussion

We see that our binomial results of call and put values converge to the *blsprice* value, which is given by `exactCall, exactPut`.

For the codes above, there are descriptions and explanation on how the codes work in the comment. To clarify a bit, the usage of *timestep* in the comments above refers to 2 different things, depending on the context. It could mean Δt, i.e. the size of our time discretization scheme. Also, by *i-th timestep*, we mean $t_i = i\Delta t$. Our partition of the time interval $[0, T]$ is given by $0 = t_0 < t_1 < \ldots < t_N = T$, where $N = T/\Delta t$. So we count from 0-th timestep to N-th timestep.

We obtain the fair values of put or call options with underlying that has dividend by using interpolation to estimate the option value at t_d^-. In particular, we approximate t_d by using (`idx`-1)-th timestep, that is we choose `idx` such that : $(idx - 2)\Delta t < t_d \leq (idx - 1)\Delta t$.

Question 1a: What is the ratio when convergence is quadratic? Does your convergence table indicate a linear or quadratic convergence rate? Explain.

Answer: The ratio will be 4 if the convergence in quadratic. This can be proved by the following argument:

Assume the convergence is quadratic. Then

$$V_0^{\text{tree}}\ (\Delta t) = V_0^{\text{exact}}\ + \alpha(\Delta t)^2 + o\left((\Delta t)^2\right), \tag{1}$$

where α is some constant independent of Δt. We also assume $\alpha \neq 0$.

Then

$$\begin{aligned} V_0^{\text{tree}}\ ((\Delta t)/2) - V_0^{\text{tree}}\ (\Delta t) &= V_0^{\text{exact}}\ + \alpha\frac{(\Delta t)^2}{4} - V_0^{\text{exact}}\ - \alpha(\Delta t)^2 + o\left((\Delta t)^2\right) \\ &= -\alpha\frac{3(\Delta t)^2}{4} + o\left((\Delta t)^2\right) \end{aligned}$$

Similarly, we have

$$V_0^{\text{tree}}\left((\Delta t)/4\right) - V_0^{\text{tree}}\left(\Delta t/2\right) = -\alpha\frac{3(\Delta t)^2}{16} + o\left((\Delta t)^2\right).$$

Thus, assuming $\alpha \neq 0$,

$$\lim_{\Delta t \to 0} \frac{V_0^{\text{tree}}\left((\Delta t)/2\right) - V_0^{\text{tree}}\left(\Delta t\right)}{V_0^{\text{tree}}\left((\Delta t)/4\right) - V_0^{\text{tree}}\left((\Delta t)/2\right)} = \lim_{\Delta t \to 0} \frac{-\alpha\frac{3(\Delta t)^2}{4} + o\left((\Delta t)^2\right)}{-\alpha\frac{3(\Delta t)^2}{16} + o\left((\Delta t)^2\right)} = 4$$

Thus my convergence tables indicate a linear convergence rate since the ratio converges to 2 instead of 4.

Question 1b: Generate tables of fair values of the same call and put options using $\Delta t = 0.005$, assuming dividend yield $\rho = 0, 2\%, 4\%, 8\%$ respectively. How do call and put values change with the dividend yield ρ?

Answer: We add $\rho = 10\%$ to show the trend of call and put values with respect to ρ more clearly. We see that call values decrease as dividend yield rho increases. Also, we see that put values increase as dividend yield rho decreases. It can intuitively explained. A higher dividend yield rho means that the ex-dividend stock value will be smaller and hence the possible payoff (thinking in terms of expectation) for call option will decrease while the possible payoff for put option (thinking in terms of expectation) will increase.

2 Q2

Listing 2: Q2

```
%% paremeters initialization

sigma = 0.2;
r = 0.03;
T = 1;
K = 10;
S0 = 10;
dt = 0.05;

%% Q2a

testlength = 9;

% Exact solution
[exactCall,exactPut] = blsprice(S0, K, r, T, sigma)

% put option
opttype = 1;

convertestput = convergencetest_drift(sigma, r, T, K, S0, opttype, dt,
    testlength);
convertestput = array2table(convertestput, ...
    'VariableNames',{'Delta t','Value','Change', 'Ratio'})

% We see that the put values converge to the exact solutions as Delta t
% goes to 0.

% We observe that the ratio in the above table are not constant and
% flutuate a lot. This means that the convergenece rate is neither linear
% nor quadartic. By p.11 in Lec 5, this may indicate that the strike price
% is not at a binomial mode.

%% Q2b smoothed payoff
% call option
opttype = 0;
smoothtestcall = convergencetest_drift_smoothed(sigma, r, T, K, S0, opttype
    , dt, testlength);
smoothtestcall = array2table(smoothtestcall, ...
    'VariableNames',{'Delta t','Value','Change', 'Ratio'})

% put option
opttype = 1;
smoothtestput = convergencetest_drift_smoothed(sigma, r, T, K, S0, opttype,
     dt, testlength);
smoothtestput = array2table(smoothtestput, ...
    'VariableNames',{'Delta t','Value','Change', 'Ratio'})
```

```
function value = driftlat(sigma, r, T, K, S0,dt, opttype)
% driftlat return put/call options value using drifting lattice
%
% Input:
% sigma: volatility of the underlying
% r: interest rate
% T: time of expiry
% K: strike price
% dt: size of timestep
% opttype: option type, 0 for call, otherwise for put
% rho: constant dividend rate
% D0: constant dividend floor
% t_d: dividend payment time


term = (r- sigma^2/2 ) * dt ;
% up and down ratio in drifting lattice
u = exp(sigma * sqrt(dt) + term);
d = exp(-sigma * sqrt(dt) + term);
% probability of going up
q = 1/2;

N = T / dt; % assume it is an integer

% vectorized approach, find payoff at final time T
% first: the stock values at final time T
%           values are arranged in desceding order
W = S0*d.^([N:-1:0]') .* u.^([0:N]');

% second: distinguish the case between call and put
if opttype == 0
    W = max(W - K,0);
else
    W = max(K - W,0);
end

% Backward iteration
for i = N:-1:1
    W = exp(-r *dt) *(q* W(2:i+1) + (1-q)* W(1:i));
end

value = W(1);
end

function value = driftlat_smoothed(sigma, r, T, K, S0,dt, opttype)
% similar to driftlat, except that we implement smoothing payoff

term = (r- sigma^2/2 ) * dt ;
u = exp(sigma * sqrt(dt) + term);
```

```
d = exp(-sigma * sqrt(dt) + term);
q = 1/2;

N = T / dt; % assume it is an integer

% vectorized approach, find payoff at final time T
% first: the stock values at final time T
%           values are arranged in desceding order
W = S0*d.^([N:-1:0]') .* u.^([0:N]');

% some useful constants to simplify the expression
up = exp(sigma * sqrt(dt));
down = exp(-sigma * sqrt(dt));

% second: distinguish the case between call and put
% smoothed payoff, cf (5.49), (5.50) in the pdf course notes
if opttype == 0
    % call case:
    idx = (W *down <= K) & (K <= W*up);
    idx1 = W* down > K;
    idx2 = W *up < K;
    W(idx2) = 0;
    W(idx1) = W(idx1).*(up -down)./ (2 * sigma * sqrt(dt)) - K;
    W(idx) = (1 / (2* sigma * sqrt(dt))) .*(W(idx) .*(up - K./W(idx))- ...
        K *(sigma * sqrt(dt) - log(K./ W(idx))));
else
    % put case:
    idx = (W *down <= K) & (K <= W*up);
    idx1 = W* down > K;
    idx2 = W *up < K;
    W(idx1) = 0;
    W(idx2) = K - W(idx2).*(up -down)./ (2 * sigma * sqrt(dt));
    W(idx) = (1 / (2* sigma * sqrt(dt))) .*(K *(log(K./ W(idx)) + sigma * sqrt(dt)) ...
        - W(idx) .*(K./W(idx) - down));
end

% Backward iteration
for i = N:-1:1
    W = exp(-r *dt) *(q* W(2:i+1) + (1-q)* W(1:i));
end

value = W(1);
end

function testtable = convergencetest_drift(sigma, r, T, K, S0, opttype, dt, testlength)
% convergencetest_drift return a table that contains results of convergence test
% for drifting lattice (without smoothing payoff)
```

```
%
% Similar input parameters as driftlat, with the additional parameters dt
% and testlength.
% dt: initial size of testing timestep
% testlength: (the number of size of timesteps) - 1, note that each subsequent
% size of timestep is obtained by size of previous timestep divided by 2


dtlist = (dt ./ 2.^(0:testlength))';

testtable = zeros(testlength + 1, 4);

testtable(:,1) = dtlist;
for i = 1:testlength + 1
    testtable(i,2) = driftlat(sigma, r, T, K, S0,dtlist(i), opttype);
end

testtable(2:end,3) = testtable(2:end,2) - testtable(1:end-1,2);
testtable(3:end,4) = testtable(2:end-1,3)./ testtable(3:end,3);
end

function testtable = convergencetest_drift_smoothed(sigma, r, T, K, S0, opttype, dt, testlength)
% convergencetest_drift_smoothed return a table that contains results of convergence test
% for drifting lattice with smoothing payoff
%
% Similar input parameters as driftlat_smoothed, with the additional parameters dt
% and testlength.
% dt: initial size of testing timestep
% testlength: (the number of size of timesteps) - 1, note that each subsequent
% size of timestep is obtained by size of previous timestep divided by 2

dtlist = (dt ./ 2.^(0:testlength))';

testtable = zeros(testlength + 1, 4);

testtable(:,1) = dtlist;
for i = 1:testlength + 1
    testtable(i,2) = driftlat_smoothed(sigma, r, T, K, S0,dtlist(i), opttype);
end

testtable(2:end,3) = testtable(2:end,2) - testtable(1:end-1,2);
testtable(3:end,4) = testtable(2:end-1,3)./ testtable(3:end,3);
end
```

2.1 Result

```
exactCall =

    0.9413


exactPut =

    0.6458


convertestput =

  10×4 table
```

Delta t	Value	Change	Ratio
0.05	0.6436	0	0
0.025	0.64602	0.0024108	0
0.0125	0.6467	0.0006804	3.5432
0.00625	0.64666	-3.3536e-05	-20.288
0.003125	0.64638	-0.00028119	0.11927
0.0015625	0.64605	-0.00032746	0.8587
0.00078125	0.64576	-0.00029579	1.1071
0.00039063	0.64584	7.9321e-05	-3.729
0.00019531	0.64581	-2.9122e-05	-2.7238
9.7656e-05	0.64581	6.3044e-06	-4.6193

```
smoothtestcall =

  10×4 table
```

Delta t	Value	Change	Ratio
0.05	0.95269	0	0
0.025	0.94706	-0.005631	0
0.0125	0.94421	-0.0028479	1.9773
0.00625	0.94278	-0.0014323	1.9884
0.003125	0.94206	-0.00071824	1.9941
0.0015625	0.9417	-0.00035978	1.9963
0.00078125	0.94152	-0.00018012	1.9975
0.00039063	0.94143	-8.9995e-05	2.0014
0.00019531	0.94139	-4.4964e-05	2.0015
9.7656e-05	0.94136	-2.2499e-05	1.9985

```
smoothtestput =

  10×4 table

     Delta t        Value        Change        Ratio
    ----------    -------    -----------    ------

          0.05    0.65388              0         0
         0.025    0.64988     -0.0039974         0
        0.0125    0.64785     -0.0020312    1.9681
       0.00625    0.64683     -0.0010239    1.9837
      0.003125    0.64631    -0.00051407    1.9918
     0.0015625    0.64605     -0.0002577    1.9948
    0.00078125    0.64592    -0.00012908    1.9965
    0.00039063    0.64586    -6.4474e-05     2.002
    0.00019531    0.64583    -3.2203e-05    2.0021
    9.7656e-05    0.64581    -1.6119e-05    1.9978
```

2.2 Discussion

For part a), note that `convertestput` is a table that contains convergence test results for the drifting lattice (without smoothing) method. We observe that the `Ratio` in `convertestput` is not constant and fluctuate a lot. This means that the convergence rate is neither linear nor quadratic. By p.11 in Lec 5, this may indicate that the strike price is not at a binomial mode. Luckily, we still see that put values converge to the exact solutions (i.e. `exactPut` $= 0.6458$) as Δt goes to 0.

For part b, the smoothing payoff method is based on the following formula:
The smoothed put payoff $\hat{P}_j^N$ is

$$\hat{P}_j^N = \begin{cases} 0 & S_j^N e^{-\sigma\sqrt{\Delta t}} > K \\ K - S_j^N \left(\frac{e^{\sigma\sqrt{\Delta t}} - e^{-\sigma\sqrt{\Delta t}}}{2\sigma\sqrt{\Delta t}}\right) & S_j^N e^{+\sigma\sqrt{\Delta t}} < K \\ \frac{1}{2\sigma\sqrt{\Delta t}}\left(K\left[\log\left(K/S_j^N\right) + \sigma\sqrt{\Delta t}\right] - S_j^N\left[\left(K/S_j^N\right) - e^{-\sigma\sqrt{\Delta t}}\right]\right) & S_j^N e^{-\sigma\sqrt{\Delta t}} \le K \le S_j^N e^{\sigma\sqrt{\Delta t}} \end{cases}$$

The smoothed call payoff $\hat{C}_j^N$ is

$$\hat{C}_j^N = \begin{cases} 0 & S_j^N e^{+\sigma\sqrt{\Delta t}} < K \\ S_j^N \left(\frac{e^{\sigma\sqrt{\Delta t}} - e^{-\sigma\sqrt{\Delta t}}}{2\sigma\sqrt{\Delta t}}\right) - K & S_j^N e^{-\sigma\sqrt{\Delta t}} > K \\ \frac{1}{2\sigma\sqrt{\Delta t}}\left(S_j^N\left[e^{\sigma\sqrt{\Delta t}} - \left(K/S_j^N\right)\right] - K\left[\sigma\sqrt{\Delta t} - \log\left(K/S_j^N\right)\right]\right) & S_j^N e^{-\sigma\sqrt{\Delta t}} \le K \le S_j^N e^{\sigma\sqrt{\Delta t}} \end{cases}$$

The table of convergence results for the drifting method with smoothing payoff are stored in `smoothtestcall` and `smoothtestput`, for call and put options respectively. We see that the corresponding call and put values converge to the exact solutions `exactCall, exactPut`. Moreover, the `Ratio` in `smoothtestcall` and `smoothtestput` are roughly 2 and seem to converge to 2 as Δt goes to 0. This implies that when smoothing method is employed, it can address the problem that there are no lattice nodes at the strike[1], and recover the linear convergence rate.

[1] this is the reason why the convergence behavior is erratic in the no smoothing case

3 Q3

3.1 Q3a

Listing 3: Q3a

```matlab
%% Parameters initialization

K = 102;
B = 100;
T = 0.5;
x = 15;
sigma = 0.2;
r = 0.03;
S0 = 100:2:120;

% fair value of down-and-out cash-or-nothing put option
V = do_cashornth_put(sigma, r, T, K, S0, x, B)

% why the first entry is actually negative ??
% is it because of floating point error?, maybe it should be small very
% small positive value, but after rounding it becomes negative

% plot the down-and-out cash-or-nothing put option for S=100:2:120
h= figure;
plot(S0,V)
xlabel('initial stock value: S0')
ylabel('option value: V')
saveas(h, 'Q3a','epsc') % stored as EPS, there is no loss in quality


function V = do_cashornth_put(sigma, r, T, K, S0, x, B)
% do_cashornth_put returns the exact down-and-out cash-or-nothing put option value
%
% allow vector input
%
% Input:
% sigma: volatility of the underlying
% r: interest rate
% T: time of expiry
% K: strike price
% S0: initial asset value
% x: cash payout
% B: down barrier

z_1 = log(S0./K)./ (sigma .* sqrt(T)) + sigma.*sqrt(T)/2;
z_2 = log(S0./B)./ (sigma .* sqrt(T)) + sigma.*sqrt(T)/2;
y_1 = log(B.^2./(S0.*K))./ (sigma .* sqrt(T)) + sigma.*sqrt(T)/2;
y_2 = log(B./S0)./ (sigma .* sqrt(T)) + sigma.*sqrt(T)/2;
```

```

inter = sigma.*sqrt(T);

V = x.*exp(-r.*T) .*(normcdf(-z_1 + inter) - normcdf(-z_2 + inter) ...
    + (S0./B) .* normcdf(y_1 - inter) - (S0./B).*normcdf(y_2 - inter));
end
```

3.1.1 Result

[2]

```
V =

   -0.0000    0.0160    0.0310    0.0444    0.0556    0.0642    0.0702
   0.0735    0.0744    0.0732    0.0703
```

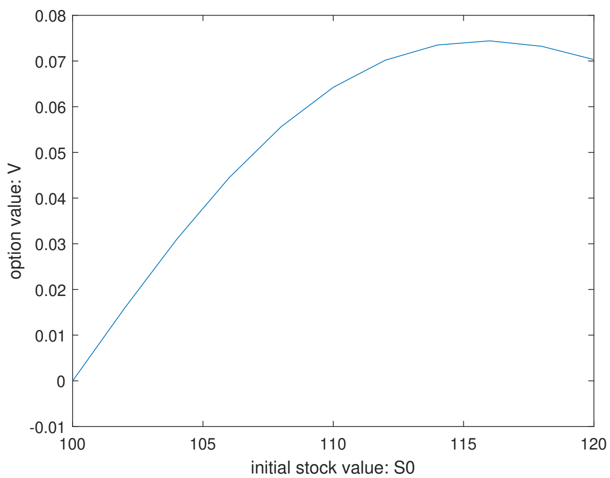

Figure 1: initial values of down-and-out cash-or-nothing put option for S = 100:2:120

Initial values of down-and-out cash-or-nothing put option are shown in `V`. The first value corresponds to $S = 100$ while the last value 0.0703 corresponds to $S = 120$.

3.2 Q3b

Listing 4: Q3b

```
%% Parameters initialization

K = 102;
B = 100;
```

[2] change a little bit of result representation so that it will not be out of box

```
T = 0.5; % expiry of 6 months
x = 15;
sigma = 0.2;
r = 0.03;
S0 = 105;

% Discussion for time discretization error
%
% The error in the computed  $\Tilde{V}$ actually depend on the time
% discretization. Note that barrier option is a path-dependent option,
% meaning that its payoff not only depend on the final value of the stock
% but also the path taken to reach this final value. The way that we
% approximate the payoff is based on a time discretization. That is, if our
% simulation suggests that at each time step $t_n$, we have $S(t_n) > B$
% and also we have $S(T) < K$, then we receive the cash amount x. However,
% between successive time step $t_n, t_{n+1}$, although we have $S(t_n),
% S(t_{n+1}) >B$, there is no guarantee that $S(s) > B$, for all $s$
% strictly between $t_n$ and $t_{n+1}$. If this is the case, then by the
% definition of down barrier option, we cannot receive cash amount $x$ in
% the final time. This produces an error where by our simulation we
% approximate the payoff by $x$ whereas the true payoff is actually 0. We
% see that the computed  $\Tilde{V}$ actually depend on the time
% discretization, in the sense that the time discretization fails to
% capture the possible fluctuation of price during the time between
% successive time discretization steps.

% Code up the MC algorithm

dtlist = [1/200, 1/400, 1/800, 1/1600, 1/3200, 1/6400];
Mlist = 1000:3000:100000;

rng('default') % for reproducibility

output = zeros(length(dtlist), length(Mlist));
for i = 1: length(dtlist)
    for j = 1:length(Mlist)
        output(i,j) = MC_barrier(sigma, r, T, K, S0, x, B, dtlist(i), Mlist
            (j));
    end
end

% obtain the exact price by part a
exact = do_cashornth_put(sigma, r, T, K, S0, x, B);

% for each of Delta t, plot the computed option values using MC for
% M=1000:3000:100000
for i = 1:length(dtlist)
    h(i) = figure(i);
    plot(Mlist, output(i, :))
    xlabel('number of sample paths M')
```

```
    ylabel('option value')
    title(strcat('timestep: ',string(dtlist(i))))
    hold on
    plot(Mlist, exact* ones(length(Mlist),1) ,'r-*')
    legend('MC price','exact')
    hold off
end

% We observe from the plots that as M goes up, that is more path
% realizations are used, the MC option value is closer to the exact price.
% However, such improvements are not very significant in the case of large
% timesteps like 1/200, 1/400, 1/800. It is more significant for the
% small timestep like 1/6400. It is because in such case the time
% discretization error (presumably O(Delta t)) is small and so the dominant
% error is sampling error O(1/sqrt(M)). In order to match these two errors
% and achieve optimal error, we need M to be O(1/Delta t ^ 2). In the case of
% timestep 1/6400, since 1/(1/6400)^2 = 40960000, way larger than 100000,
% increasing M will give us more significant improvement. Another
% observation is that when M goes up, the MC values seem to stabilize. This
% can intuitively explained by that increase in M will reduce the effect of
% random path realization.

% Here are some other plots that I think might be of interest

%% Explanation for figure g(1)
% Here we fix the number of sample paths to be 100000 and plot the option
% value against different time discretization. We see the general pattern
% that the finest the time discretization is, the more accurate the MC
% price is. This is becasuse we have time discretization error and using a
% finer time discretization scheme will reduce such error.

g(1) = figure;
plot(dtlist, output(:,end))
xlabel('Delta t')
ylabel('option value')
title('100000 sample paths for each time discretization')
hold on
plot(dtlist, exact* ones(length(dtlist),1) ,'r-*')
legend('MC price','exact')
hold off
% Consider the smallest (or finest) time discretization scheme and plot the
% MC price for different number of sample paths

%% Explanation of figure g(2)
% Here we plot a 3D plot of the absolute value of the difference between MC
% price and exact price. We observe that as time discretization becomes
% finer and finer, the accuracy of MC price increase significantly. This is
% because time discretization is reduced.
% Moreover, we see that as M increases, the MC price becomes stabler. This
```

```
% is because the reduction of the effect of randomness. We also see slight
% increase in MC price accuracy as M increases, due to reduction in
% sampling error.
%
% The value of the sampling error and time discretization error can be
% somewhat estimated by observing the following
% 1./ sqrt(Mlist)
% dtlist

g(2) = figure;
[X,Y] = meshgrid(Mlist, dtlist);
mesh(X,Y,abs(output- exact))
ylabel('Delta t')
xlabel('M')
title('3D plot of absolute error')

% Save the figures
for i=1:length(h)
    saveas(h(i),strcat('fig_needed',string(i)),'epsc')
end
for i =1:length(g)
    saveas(g(i),strcat('fig_extra3',string(i)),'epsc')
end

function value = MC_barrier(sigma, r, T, K, S0, x, B, dt, M)
% MC_barrier returns the down-and-out cash-or-nothing put option
%   value by Monte Carlo simulation
%
% Input:
% sigma: volatility of the underlying
% r: interest rate
% T: time of expiry
% K: strike price
% S0: initial asset value
% x: cash payout
% B: down barrier
% dt: timestep
% M: number of path realization

N = T/ dt; % assume it is integer
V = zeros(M,1); % placeholder for the option value of each path
idx = true(M,1); % storing the indices of path that has positive final payout
S = S0 * ones(M,1); % initial asset value

for n = 0:N-1
    % obtain phi_n for each path
    sample = randn(M,1);
    % calculate S(t_{n+1}), see (8) in assignment
    S = S.*exp((r - 1/2 * sigma^2) * dt + sigma.* sample .* sqrt(dt));
```

```matlab
    % record the indices of path of S that is below the barrier at this timestep
    check = S <= B;
    idx(check) = false;
end

% final check: only realization with final asset value less than K can have
% cash payout
endcheck = S >= K;
idx(endcheck) = false;

% compute the MC price by (9) in assignment
V(idx) = x;
value = exp(-r*T) *mean(V);
end

function V = do_cashornth_put(sigma, r, T, K, S0, x, B)
% do_cashornth_put returns the exact down-and-out cash-or-nothing put option value
%
% allow vector input
%
% Input:
% sigma: volatility of the underlying
% r: interest rate
% T: time of expiry
% K: strike price
% S0: initial asset value
% x: cash payout
% B: down barrier

z_1 = log(S0./K)./ (sigma .* sqrt(T)) + sigma.*sqrt(T)/2;
z_2 = log(S0./B)./ (sigma .* sqrt(T)) + sigma.*sqrt(T)/2;
y_1 = log(B.^2./(S0.*K))./ (sigma .* sqrt(T)) + sigma.*sqrt(T)/2;
y_2 = log(B./S0)./ (sigma .* sqrt(T)) + sigma.*sqrt(T)/2;

inter = sigma.*sqrt(T);

V = x.*exp(-r.*T) .*(normcdf(-z_1 + inter) - normcdf(-z_2 + inter) ...
    + (S0./B) .* normcdf(y_1 - inter) - (S0./B).*normcdf(y_2 - inter));
end
```

3.2.1 Result

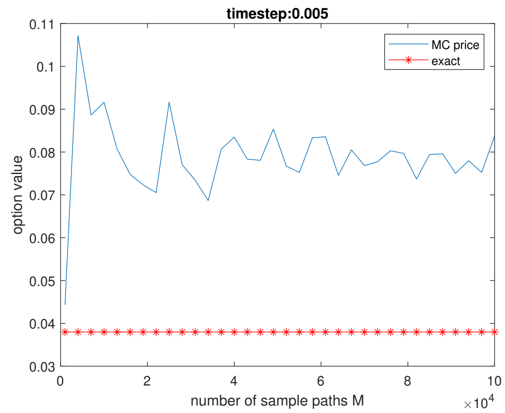

Figure 2: $\Delta t = 1/200$, MC price against M

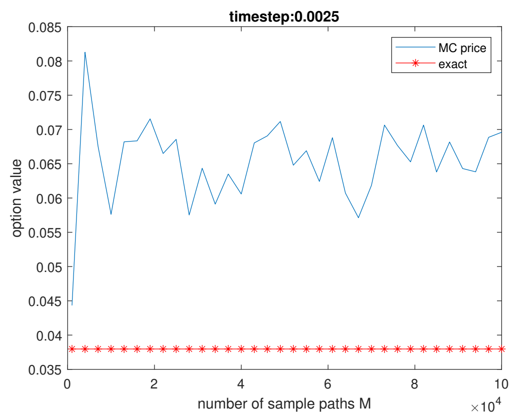

Figure 3: $\Delta t = 1/400$, MC price against M

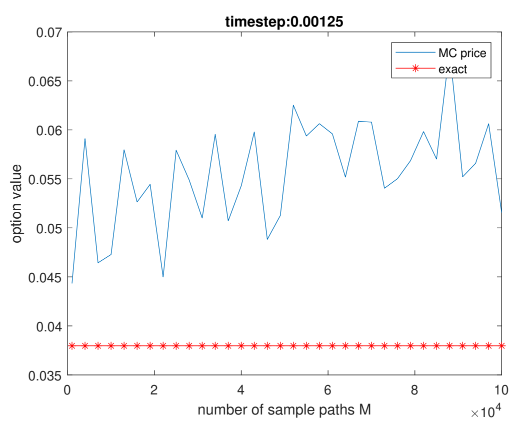

Figure 4: $\Delta t = 1/800$, MC price against M

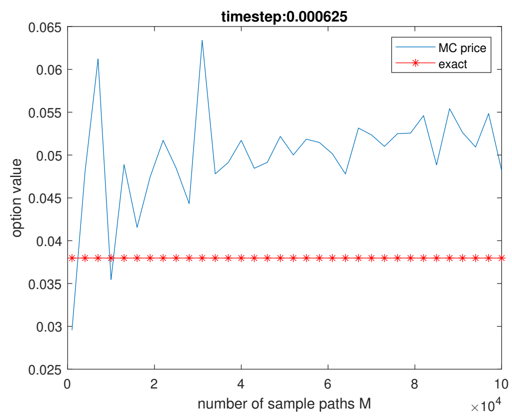

Figure 5: $\Delta t = 1/1600$, MC price against M

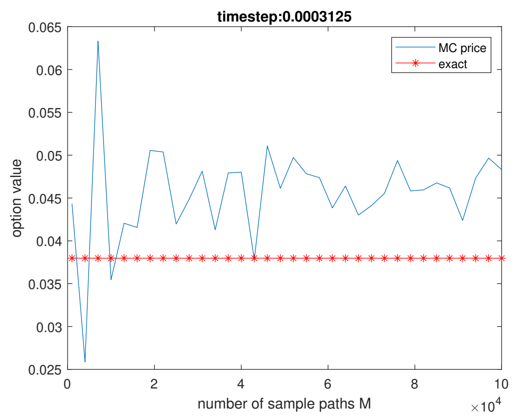

Figure 6: $\Delta t = 1/3200$, MC price against M

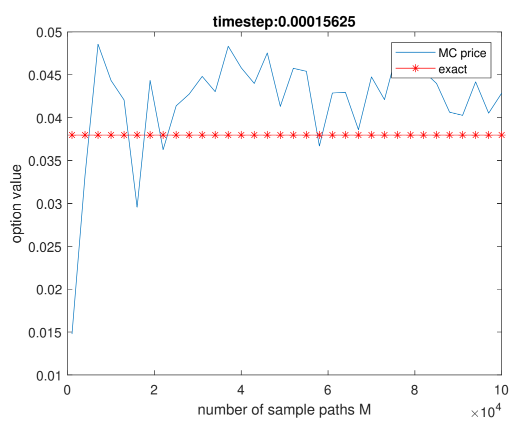

Figure 7: $\Delta t = 1/6400$, MC price against M

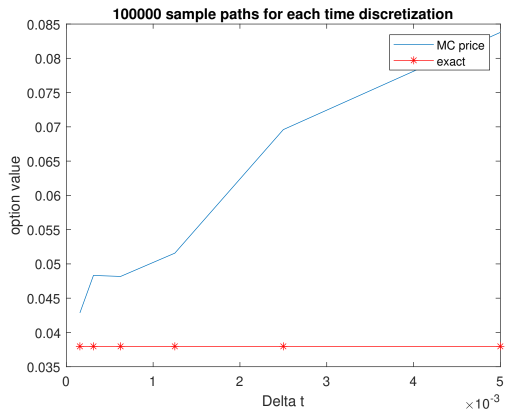

Figure 8: $M = 100000$, MC price against Δt

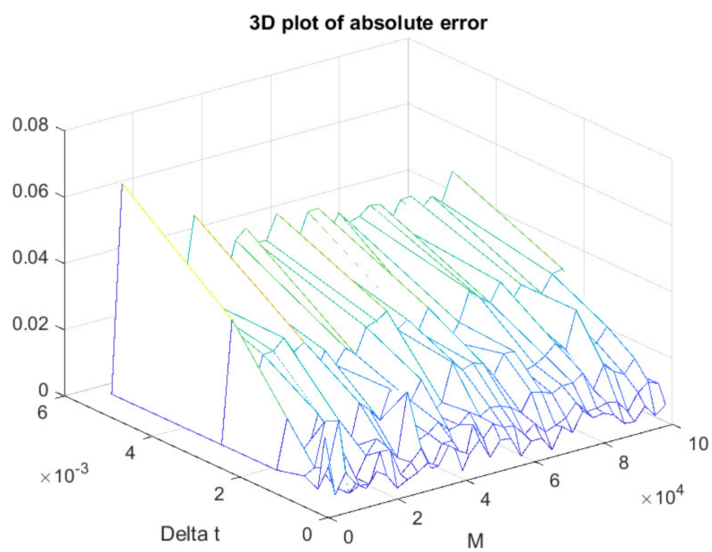

Figure 9: 3D plot of the absolute value of the difference between MC price and exact price

3.2.2 Discussion

Question: The price simulation using (8) has no time discretization error. Does the error in the computed value V (S(0); 0) depend on the time discretization? Explain.

Answer: The error in the computed $\tilde{V}$ actually depend on the time discretization. Note that barrier option is a path-dependent option, meaning that its payoff not only depend on the final value of the stock but also the path taken to reach this final value. The way that we approximate the payoff is based on a time discretization. That is, if our simulation suggests that at each time step t_n, we have $S(t_n) > B$ and also we have $S(T) < K$, then we receive the cash amount x. However, between successive time step t_n, t_{n+1}, although we have $S(t_n), S(t_{n+1}) > B$, there is no guarantee that $S(s) > B$, for all s strictly between t_n and t_{n+1}. If this is the case, then by the definition of down barrier option, we cannot receive cash amount x in the final time. This produces an error where by our simulation we approximate the payoff by x whereas the true payoff is actually 0. We see that the computed $\tilde{V}$ actually depend on the time discretization, in the sense that the time discretization fails to capture the possible fluctuation of price during the time between successive time discretization steps.

Question: Assume that the initial asset price $S(0) = 105$ and other parameters the same as in (a). For each of $\Delta t = \frac{1}{200}, \frac{1}{400}, \frac{1}{800}, \frac{1}{1600}, \frac{1}{3200}, \frac{1}{6400}$, plot the computed option values using MC for $M = 1000 : 3000 : 100000$ respectively. Show the exact price computed from (a) on the plot. Explain what you see.

Answer: We observe from the plots (Figure 2 to 7) that as M goes up, that is more path realizations are used, the MC option value is closer to the exact price. However, such improvements are not very significant in the case of large timesteps like $1/200, 1/400, 1/800$. It is more significant for the small timestep like 1/6400. It is because in such case the time discretization error (presumably $O(\Delta t)^3$) is small and so the dominant error is sampling error $O(1/\sqrt{M})$. In order to match these two errors and achieve optimal error, we need M to be $O(1/(\Delta t)^2)$. In the case of timestep 1/6400, since $1/(1/6400)^2 = 40960000$, way larger than 100000, increasing M will give us more significant improvement. Another observation is that when M goes up, the MC values seem to stabilize. This can intuitively explained by that increase in M will reduce the effect of random path realization.

We also attach two extra plots that might be of interest, they are Figure 8 and Figure 9. For Figure 8, we fix the number of sample paths to be 100000 and plot the option value against different time discretization. We see the general pattern that the finer the time discretization is, the more accurate the MC price is. This is becasuse we have time discretization error and using a finer time discretization scheme will reduce such error. For Figure 9, we plot a 3D plot of the absolute value of the difference between MC price and exact price. We observe that as time discretization becomes finer and finer, the accuracy of MC price increase significantly. This is because time discretization is reduced. Moreover, we see that as M increases, the MC price becomes stabler. This is because the reduction of the effect of randomness. We also see slight increase in MC price accuracy as M increases, due to reduction in sampling error.

The value of the sampling error and time discretization error can be somewhat estimated by following commands: `1./sqrt(Mlist)`, `dtlist`.

[3] After doing Q4, the paper claimed that this should be $O(\sqrt{\Delta t})$, so the discussion following it is not technically correct. I have added an appendix for this question.

3.2.3 Appendix

After doing the graduate level question and reading the corresponding paper, I know that time discretization error should be $O(\sqrt{\Delta t})$. This makes my interpretation in the above on the effect of increasing M not quite right. Based on the time discretization error $O(\sqrt{\Delta t})$, for the sampling error to match the order of the time discretization error, we require $M = O(1/\Delta t)$. Note that the smallest Δt is $1/6400$ so the largest M required (for matching the errors) in this question is $M = O(6400)$. Any M that is larger than $O(6400)$ will not give significant improvement on the total error since the total error will then be dominated by the time discretization error. This explains why we don't see significant improvements on our MC price when increasing M in Figure 2 to 7. But the effect of reducing randomness still exist when we increase M.

4 Q4 Graduate Student Question

Listing 5: Q4

```
close all

%% parameters initialization
% tstart = tic; & count the time spent

dtlist = [0.02, 0.01, 0.005, 0.0025];
M = 100000; % quite large

K = 102;
B = 100;
T = 0.5; % expiry of 6 months
x = 15;
sigma = 0.2;
r = 0.03; % use 0.03 or 0.1? (paper value: 0.1), (Table 1 in asg: 0.03)
% based on discussion on piazza, choose 0.03

% some numeric experiements show that if we change sigma to be less than r,
% we will have large error and that the modified MC actually not as good as
% usual MC. This can be seen by choosing r = 0.1, sigma = 0.1.
% I am not sure why this happens. Would it be possible that the analytic
% formula in part 3a can only be applied to the case that sigma > r? Again,
% I am not sure.

S0 = 105;

%% reproduction of figures of example 1 in the paper

rng('default') % for reproducibility

% exact put option value
exact = do_cashornth_put(sigma, r, T, K, S0, x, B)

% V for usual MC price, V_mod for modified MC
V = zeros(length(dtlist),1);
V_mod = V;

for i = 1:length(dtlist)
    V(i) = MC_barrier(sigma, r, T, K, S0, x, B, dtlist(i), M);
end
V

for i = 1:length(dtlist)
    V_mod(i) = mod_MC_barrier(sigma, r, T, K, S0, x, B, dtlist(i), M);
end
V_mod

```

```
% tend = toc(tstart) % shows the time spent for this program

% plot the modified MC prices and exact price
h = figure(1);
plot(dtlist, V_mod','-or')
hold on
plot(dtlist, ones(1,length(dtlist))* exact,'-|b')
xlabel('DeltaT')
ylabel('Option price')
legend('V-Monte Carlo','Vexact')
hold off

% plot the approximation errors between the standard MC and the improved MC
g = figure(2);
plot(T./dtlist, abs(V- exact),'-+b')
hold on
plot(T./dtlist, abs(V_mod - exact), '-or')
xlabel('Number of steps')
ylabel('Error (absolute)')
legend('Standard MC','Improve MC')
hold off

saveas(h,'Q4fig1','epsc')
saveas(g,'Q4fig2','epsc')

% more plot, also show the time discretization error
g2 = figure(3);
plot(T./dtlist, abs(V- exact),'-+b')
hold on
plot(T./dtlist, abs(V_mod - exact), '-or')
plot(T./dtlist, sqrt(dtlist), '-')
xlabel('Number of steps')
ylabel('Error (absolute)')
legend('Standard MC','Improve MC','sqrt(dtlist)')
hold off
saveas(g2,'Q4fig3','epsc')


function V = do_cashornth_put(sigma, r, T, K, S0, x, B)
% do_cashornth_put returns the exact down-and-out cash-or-nothing put option value
%
% allow vector input
%
% Input:
% sigma: volatility of the underlying
% r: interest rate
% T: time of expiry
% K: strike price
% S0: initial asset value
```

```
% x: cash payout
% B: down barrier

z_1 = log(S0./K)./ (sigma .* sqrt(T)) + sigma.*sqrt(T)/2;
z_2 = log(S0./B)./ (sigma .* sqrt(T)) + sigma.*sqrt(T)/2;
y_1 = log(B.^2./(S0.*K))./ (sigma .* sqrt(T)) + sigma.*sqrt(T)/2;
y_2 = log(B./S0)./ (sigma .* sqrt(T)) + sigma.*sqrt(T)/2;

inter = sigma.*sqrt(T);

V = x.*exp(-r.*T) .*(normcdf(-z_1 + inter) - normcdf(-z_2 + inter) ...
    + (S0./B) .* normcdf(y_1 - inter) - (S0./B).*normcdf(y_2 - inter));
end

function value = MC_barrier(sigma, r, T, K, S0, x, B, dt, M)
% MC_barrier returns the down-and-out cash-or-nothing put option
%   value by Monte Carlo simulation
%
% Input:
% sigma: volatility of the underlying
% r: interest rate
% T: time of expiry
% K: strike price
% S0: initial asset value
% x: cash payout
% B: down barrier
% dt: timestep
% M: number of path realization

N = T/ dt; % assume it is integer
V = zeros(M,1); % placeholder for the option value of each path
idx = true(M,1); % storing the indices of path that has positive final payout
S = S0 * ones(M,1); % initial asset value

for n = 0:N-1
    % obtain phi_n for each path
    sample = randn(M,1);
    % calculate S(t_{n+1}), see (8) in assignment
    S = S.*exp((r - 1/2 * sigma^2) * dt + sigma.* sample .* sqrt(dt));
    % record the indices of path of S that is below the barrier at this timestep
    check = S <= B;
    idx(check) = false;
end

% final check: only realization with final asset value less than K can have
% cash payout
endcheck = S >= K;
idx(endcheck) = false;
```

```
% compute the MC price by (9) in assignment
V(idx) = x;
value = exp(-r*T) *mean(V);
end

function value = mod_MC_barrier(sigma, r, T, K, S0, x, B, dt, M)
% mod_MC_barrier returns the down-and-out cash-or-nothing put option
%   value by modified Monte Carlo, using exceedance probabiliy
%
% Input:
% sigma: volatility of the underlying
% r: interest rate
% T: time of expiry
% K: strike price
% S0: initial asset value
% x: cash payout
% B: down barrier
% dt: timestep
% M: number of path realization

N = T/ dt; % assume it is integer
V = zeros(M,1); % placeholder for the option value of each path
idx = true(M,1); % storing the indices of path that has positive final payout
Sold = S0 * ones(M,1); % initial asset value, will also be used as S(t_n)
Snew = zeros(M,1); % S(t_{n+1})

for n = 0:N-1
    % obtain phi_n for each path
    sample = randn(M,1);
    % calculate S(t_{n+1}), see (8) in assignment
    Snew = Sold.*exp((r - 1/2 * sigma^2) * dt + sigma.* sample .* sqrt(dt));
    % record the indices of path of S that is below the barrier at this timestep
    badcheck = Snew <= B;
    idx(badcheck) = false;

    % exceedance probability, p_{n+1} in p.68 of the paper
    exceedprob = exp(-2.*(B-Sold(idx)).*(B - Snew(idx))./ ...
        (sigma^2.*Sold(idx).^2 * dt));
    % unifromly distributed RV, u_n
    uniformRV = unifrnd(0,1, size(exceedprob));
    % If p_n \geq u_n, that means exceedane probability is high and we
    % regard that as S reach the barrier in the time interval (t_{n},
    % t_{n+1}). We record the indices that exceedance probability is small.
    goodcheck2 = uniformRV > exceedprob;
    % only those indices with small exceedance probability can have
    % positive payoff
```

```
    idx(idx) = goodcheck2;

    Sold = Snew;
end

% final check: only realization with final asset value less than K can have
% cash payout
endcheck = Snew >= K;
idx(endcheck) = false;

% compute the MC price by (9) in assignment
V(idx) = x;
value = exp(-r*T) *mean(V);
end
```

4.1 Result

```
exact =

    0.0380


V =

    0.1281
    0.0987
    0.0703
    0.0613


V_mod =

    0.0355
    0.0352
    0.0366
    0.0405
```

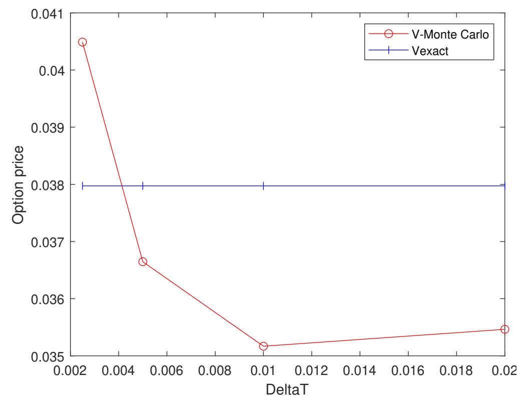

Figure 10: Exact and new Monte Carlo values

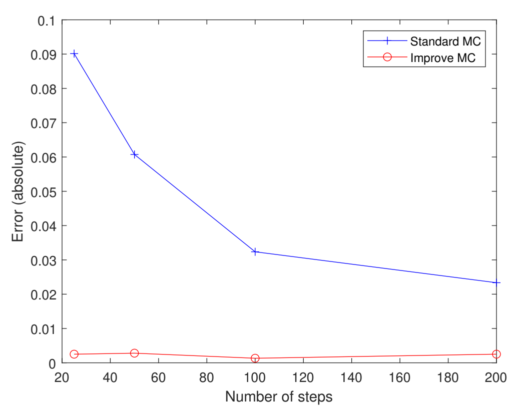

Figure 11: Comparison of approximation errors between the standard MC and the improve MC

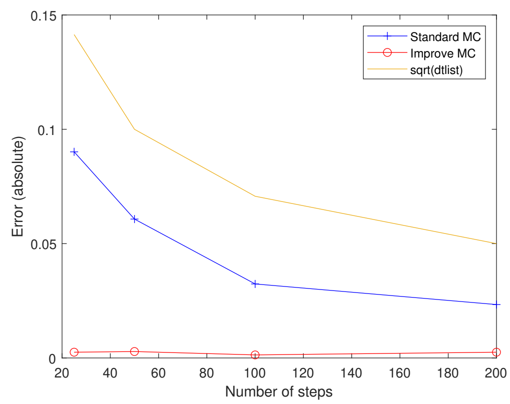

Figure 12: Comparison of approximation errors between the standard MC and the improve MC, with `sqrt(dtlist)` for time discretization error

4.2 Discussion

Question: Explain why the Monte Carlo method in Question 3 is slow in obtaining an accurate barrier option value.

Answer: We have discussed a bit on the optimal choice of M in Q3 so that the sampling error can match the time discretization error. To facilitate discussion, here we employ the notation in the paper and lay down some terminologies first.

We let τ be the hitting time, that is, the first time our underlying asset S_t hit the barrier B. Let $\tilde{\tau}$ be our approximation of the hitting time τ [4]. Let $\Lambda(S_\tau, \tau)$ be the discounted payoff function and also let Q be a risk-neutral measure. Then we know that, by risk-neutral valuation that the price of the option under consideration is

$$V(s,t) = E^Q(\Lambda(S_\tau, \tau)|S_t = s)$$

And the monte carlo price for M simulations is:

$$\tilde{V}(s,t) = \frac{1}{M}\sum_{i=1}^{M}\Lambda(S_{\tilde{\tau}}, \tilde{\tau}, \omega_i)$$

where $\omega_i \in \Omega$ and Ω is our probability space. So basically, we are approximating $V(s,t)$ by $\tilde{V}(s,t)$.

Now, following the paper, we can split the global error into the first hitting time error ε_T and statistical error (that is sampling error) ε_S as follows:

$$\begin{aligned}\varepsilon := |V(s,t) - \widetilde{V}(s,t)| &= \left(E^Q\left[\Lambda\left(S_\tau, \tau\right) - \Lambda\left(S_{\tilde{\tau}}, \widetilde{\tau}\right) \mid S_{\mathrm{t}} = s\right]\right) \\ &+ \left(E^Q\left[\Lambda\left(S_{\tilde{\tau}}, \widetilde{\tau}\right)\right] - \frac{1}{M}\sum_{j=1}^{M}\Lambda\left(S_{\tilde{\tau}}, \widetilde{\tau}; \omega_j\right)\right) \\ &= \varepsilon_T + \varepsilon_S\end{aligned}$$

This gives precisely the time discretization error (i.e. first hitting time error) ε_T and the sampling error ε_S. Note that the Monte Carlo method in Question 3 does not address the time discretization ε_T. And by the discussion in the paper (p.68), we know that for continuously monitored barrier options, the hitting time error ε_T is of order $O(\frac{1}{\sqrt{N}})$, where N is the number of time steps, which is just $T/(\Delta t)$, where T is the time to expiry. So equivalently, we see that $\varepsilon_T = O(\sqrt{\Delta t})$, which is quite big.

Now we do a complexity analysis to justify why the Monte Carlo employed in Question 3 is slow in obtaining an accurate barrier option value. Note that in this question we set $M = 100000$ which is quite large. In the following analysis we will use $M = 100000$ so that we don't have to worry about matching sampling error to the time discretization error. We see that the sampling error $\varepsilon_S = O(1/\sqrt{M}) = O(0.0032)$ is generally much less than the $\varepsilon_T = O(\sqrt{\Delta t})$ for our choice of $\Delta t = \frac{1}{200}, \frac{1}{400}, \frac{1}{800}, \frac{1}{1600}, \frac{1}{3200}, \frac{1}{6400}$ in Question 3 [5]. This means the global error is dominated by the time discretization error $\varepsilon_T = O(\sqrt{\Delta t})$. We know that complexity (i.e. computational cost, or even more precisely the numbers of floating point operations)

[4] Note that we did not make very precise our terminology. For instance, we can make $\tilde{\tau}$ precise, by defining it to be smallest t_n such that $S_{t_n} \leq B$, and ∞ if such t_n does not exist, where t_n are our time discretization time steps. I don't think such level of precision is required in this question, so I stick to the rather intuitive and less formal definition, as it was done in the paper.

[5] Note that $O(\sqrt{1/6400}) = O(0.0125)$

is given by $O((\frac{T}{\Delta t})M)$. Thus, to reduce the the time discretization error by one half, we have to take $\Delta t/4$, obtaining $\varepsilon_T = O(\sqrt{\Delta t/4}) = \frac{O(\Delta t)}{2}$. Then our complexity will be $O((\frac{T}{\Delta t/4})M) = 4O((\frac{T}{\Delta t})M)$. The conclusion is then: if we use a very large M, each error reduction by a factor of 2 requires 2^2 times more computational cost, explaining why the MC method is slow in obtaining an accurate barrier option value. This is the consequence of not trying to reduce ε_T.

Here we also consider the case of not assuming M to be large. As mentioned in Lecture 8, we want the total error to have the same order as ε_T. This require $\varepsilon_T = O(\sqrt{\Delta t})$ and $\varepsilon_S = O(1/\sqrt{M})$ have the same order and lead to the optimal choice of M to be $M = O(\frac{1}{\Delta t})$. By previous discussion, we know that complexity is complexity $= O((\frac{T}{\Delta t})M) = O(1/(\Delta t)^2)$. Thus we have

$$\Delta t = O(\frac{1}{\text{complexity}^{1/2}})$$

And hence

$$\text{error} = O(\varepsilon_T) = O(\sqrt{\Delta t}) = O(\frac{1}{\text{complexity}^{1/4}})$$

Thus, we have the conclusion that: to reduce error by factor of 10, we need to increase computation by a factor of 10^4. [6] This explains why the MC method is slow in obtaining an accurate barrier option value. This is the consequence of not trying to reduce ε_T.

Question: Comment on your observations of the computation results. Compare and discuss the improvement of the results compared to your implementation in previous barrier option pricing implementation.

Answer: We see that by implementing the exceedance probability approach, the global error (which has the same order as time discretization error since our $M = 100000$ is large) has been reduced a lot. The error reduction is often more than 10 times. This can be seen by observing Figure 11, where we plot the error from the usual Monte Carlo and the error from the modified Monte Carlo.

In Figure 12, we also plot the `sqrt(dtlist)`, which represent the time discretization error ε_T. We see that the shape of it is very similar to the shape of the error from the usual Monte Carlo. This is evidence to the claim that $\varepsilon_T = O(\sqrt{\Delta t})$. One interesting observation is that the error from the modified Monte Carlo does not seem to decrease when Δt decreases (or equivalently when N increases). I have used different random seed and observe the same pattern. I guess it is because our exceedance probability approach reduce the time discretization error quite significantly so that our choice of $\Delta t = [0.02, 0.01, 0.005, 0.0025]$ is not small enough to observe the expected behavior of error reduction when Δt is smaller.

We also see that the modified MC price for the largest $\Delta t = 0.02$ already excels the usual MC price using smallest $\Delta t = 0.0025$. This provides great computational efficiency that we can obtain even better results in a much lower cost. Note that the computational cost of modified MC and the usual MC have the same order. After all, the exceedance probability approach only require calculation of `exceedprob` and generation of uniform random numbers, which to my understanding one single such operation is of computational cost $O(M)$.

[6] In general, fixing the sampling error to be $O(1/\sqrt{M})$, if $\varepsilon_T = O((\Delta t)^a)$ for some $a > 0$, we have error $= O(\varepsilon_T) = O(\frac{1}{\text{complexity}^{a/(2a+1)}})$. Note that $\frac{x}{2x+1}$ is a strictly increasing function so that the higher the time discretization error is (i.e. smaller a), the higher the complexity

required to reduce the global error by a certain factor.