Project Document

In all these functions, we have asserted the validity of inputs, mainly for whether scalar or vector, whether the dimensions between inputs match or not.

1. Implementation, output and test[1][2]

(1) getBlackCall.m

Apply the Black-Scholes formula (5). Also deal with zero strike case, in which the premium is simply the forward spot price.

```
%% calculate Black-Scholes call price
d1 = log(f./Ks(Ks>0))./Vs(Ks>0)/sqrt(T) + 0.5*Vs(Ks>0)*sqrt(T);
d2 = d1 - Vs(Ks>0)*sqrt(T);
u(Ks>0) = f*normcdf(d1) - Ks(Ks>0).*normcdf(d2);
u(Ks==0) = f;
```

Code 2. 1: getBlackCall.m[3]

To **test** getBlackCall.m, we plot it out to check whether it satisfies properties we expect[4].

The curve is smooth; the output decreases with strike K and increases with other inputs. Test passed!

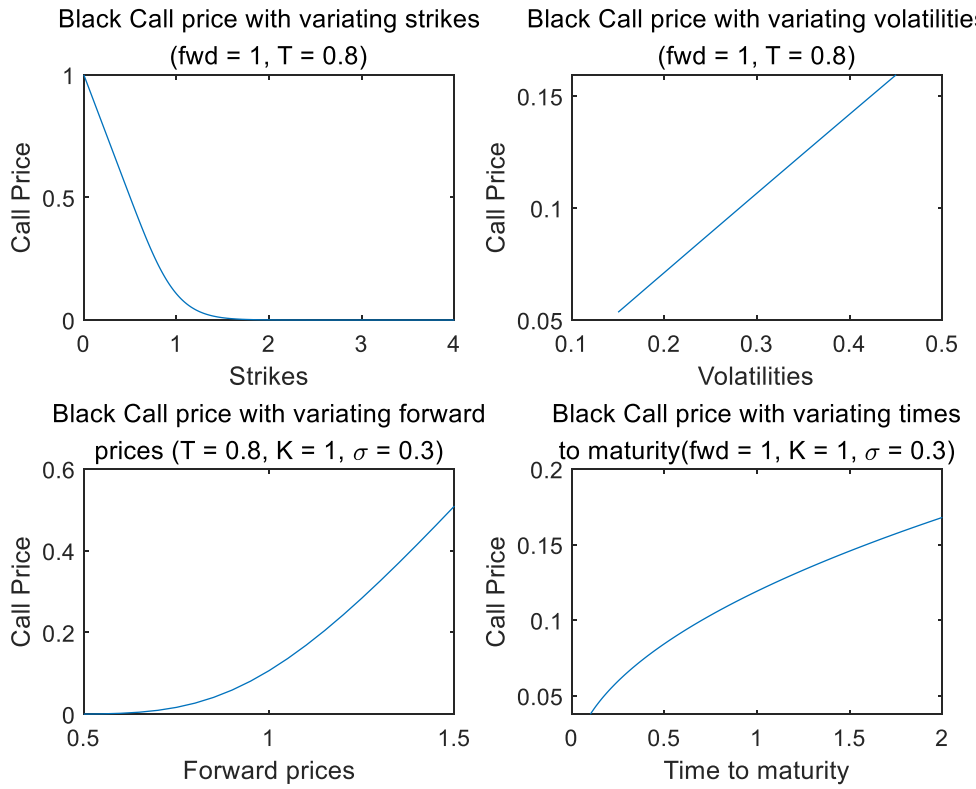
---

[1] In order for the test to run quickly, in some files we don't write multiple inputs, but we have checked for a wide range of them.

[2] It may be easier for a reader to modify the parameters/inputs if writing the test file as a function, but here we don't do this, because we write tests according to each getXX function. That is, we may write multiple tests in a file for a single getXX function.

[3] We only post key part of the code here. Please check the MATLAB file for details.

[4] Illustration of output and test of the function are the same in this part.



Output/Test 2.1: getBlackCall.m

(2) makeDepoCurve.m and <u>its curve output</u>

We process the raw data to generate domestic yield curve domCurve, foreign yield curve forCurve, and use the output in getRateIntegral.m to compute discount factor (as well as forward spot price in 2.3).

We apply bootstrapping to obtain constant/average interest rate in each time sub-interval, mainly use the formula as below:

$$r_{i-1,i} = \frac{r_{T_i} T_i - r_{T_{i-1}} T_i}{T_i - T_{i-1}}$$

```
%% apply bootstrapping
% we apply the bootstrapping to obtain piecewise constant (average)
% interest rate in each time interval
% exp(r1*T1)*exp(r12*(T2-T1)) = exp(r2*T2) ---> r12 = (r2*T2-r1*T1)/(T2-T1)
dts = diff(ts); % calculate difference between time grid point
TR = -log(dfs); % calculate total interest up to each time grid point
IR = TR./ts; % convert discount factor to interest rate (continuously)
numerator = diff(IR.*ts); % calculate the numerator in the formula
IR_boot = numerator./dts; % calculate the piecewise constant IR
curve(:,1) = [0;ts];
curve(:,2) = [0;IR(1);IR_boot];
curve(:,3) = [0;TR];
```

Code 2.2.1: makeDepoCurve.m

The **curve** here includes three columns, the first for time to maturity, the second for constant interest rate between every two grids in first column, and the third for total interest up to each time grid.

domCurve			**forCurve**		
Year **Ti**	Forward Rate **Ti-1** to **Ti**	Log Return **0** to **Ti**	Year **Ti**	Forward Rate **Ti-1** to **Ti**	Log Return **0** to **Ti**
0.00	0.00%	0.00%	0.00	0.00%	0.00%
0.02	1.98%	0.04%	0.02	4.02%	0.08%
0.04	2.09%	0.08%	0.04	4.07%	0.16%
0.06	2.09%	0.12%	0.06	4.13%	0.23%
0.08	2.14%	0.16%	0.08	4.24%	0.32%
0.16	2.20%	0.35%	0.16	4.30%	0.68%
0.24	2.31%	0.54%	0.24	4.45%	1.05%
0.50	2.40%	1.14%	0.50	4.60%	2.21%
1.00	2.60%	2.45%	1.00	4.90%	4.68%
1.50	3.00%	3.94%	1.50	5.50%	7.40%
2.00	3.50%	5.70%	2.00	6.25%	10.55%

Output 2.2.1: domCurve/forCurve

(3) getRateIntegral.m

We have obtained total interest up to each time grid. We only need to locate t and integrate between t and the time grid right before t over the piecewise constant interest rate function. For simplicity of the code, we use ***min()*** to adjust when time input is

beyond our grids:

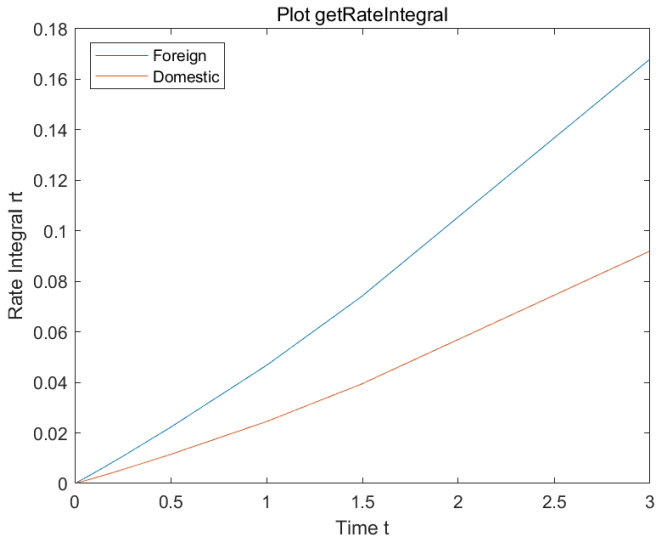
$$r_t = \begin{cases} r_{T_{i-1}} + r_{i-1,i} * (t - T_{i-1}) & t \in [T_{i-1}, T_i)\ i = 1,2 \dots, n \\ r_{T_N} + r_{N-1,N} * (t - T_N) & t \geq T_N \end{cases}$$

```
%% Integrate over a stepwise constant function
if t<0
    error('Time input must be positive')
else
    index = sum(t>=curve(:,1));
    integ = curve(index,3)+(t-curve(index,1))*curve(min(index+1,size(curve,1)),2);
end
```

Code 2.2.2: getRateIntegral.m

We locate t by ***comparing t with our time grid vector*** (first column of curve) and obtain a series of logical 0s and 1s, then sum them up and get an integer number, which reports us location of t.

To **test** getRateIntegral.m, firstly we plot the integral of both foreign and domestic rate. The function is continuous, piecewise linear, and monotonically increases from 0 for both foreign and domestic interest rate as expected.



Output/Test 2.2: getRateIntegral.m

We then do two additional **tests**:

1. If **getRateIntegral** is called with any of the tenor times T_i, given in the market, it should return the integral such that $exp\left(\int_0^{T_i} r(u)du = M_i\right)$. Test passed!

```
%% Test getRateIntegral 2: Show that return the original point
integ_kink_for = zeros(1,length(curve_for));
integ_kink_dom = zeros(1,length(curve_dom));

% Calculate the integral value in the original point and ortanize in a vector
for rr = 1: length(curve_for)
    integ_kink_for(1,rr) = getRateIntegral(curve_for,curve_for(rr,1));
end

for tt = 1: length(curve_dom)
    integ_kink_dom(1,tt) = getRateIntegral(curve_dom,curve_dom(tt,1));
end

% Change the original data into log return and organize in a vector
real_kink_for = [zeros(1,1);-log(fordf)]';
real_kink_dom = [zeros(1,1);-log(domdf)]';

%Test weather the original data fit the integral value, take 1e-10 as threshold
log_test2_for = sum((real_kink_for - integ_kink_for) > 1e-10);
log_test2_dom = sum((real_kink_dom - integ_kink_dom) > 1e-10);

if (log_test2_for==0) & (log_test2_dom == 0)
    disp('Test 2 passed: Original Value is Returned');
else
    disp('Test 2 failed: please check the function');
end
```

Test 2.2: getRateIntegral.m - 1

2. The **getRateIntegral** return rate integral 0 with extreme input t=0, so it is well behaved on extreme value. Test passed!

```
%% Test getRateIntegral 3: Show that return extreme value
log_test3_for = (getRateIntegral(curve_for,0)==0);
log_test3_dom = (getRateIntegral(curve_dom,0)==0);

if (log_test3_for == 1) & (log_test3_dom == 1)
    disp('Test 3 passed: Function Behaves Well in Extreme Value t=0 ');
else
    disp('Test 3 failed: please check the function');
end
```

Test 2.2: getRateIntegral.m - 2

(4) makeFwdCurve.m and its curve object

Exploit the above result and obtain the difference between domestic rate and foreign rate and then save the result. Therefore, the **curve** object here essentially saves the same thing as makeDepoCurve.m, except that it saves the differences between domestic and foreign in the last two columns.

```
%% input check
if size(domCurve,1)~=size(forCurve,1) || size(domCurve,2)~=size(forCurve,2) ||...
        sum(size(spot))~=2 || sum(size(tau))~=2 || sum(domCurve(:,1)~=forCurve(:,1))~=0
    error("Input invalid. Please check!");
end
%% exploit the result from section 2.2
curvedata(:,1) = domCurve(:,1);
curvedata(:,2:3) = domCurve(:,2:3)-forCurve(:,2:3);

%% save result in curve
curve.curvedata = curvedata;
curve.spot = spot;
curve.tau = tau;
```

Code 2.3.1: makeFwdCurve.m

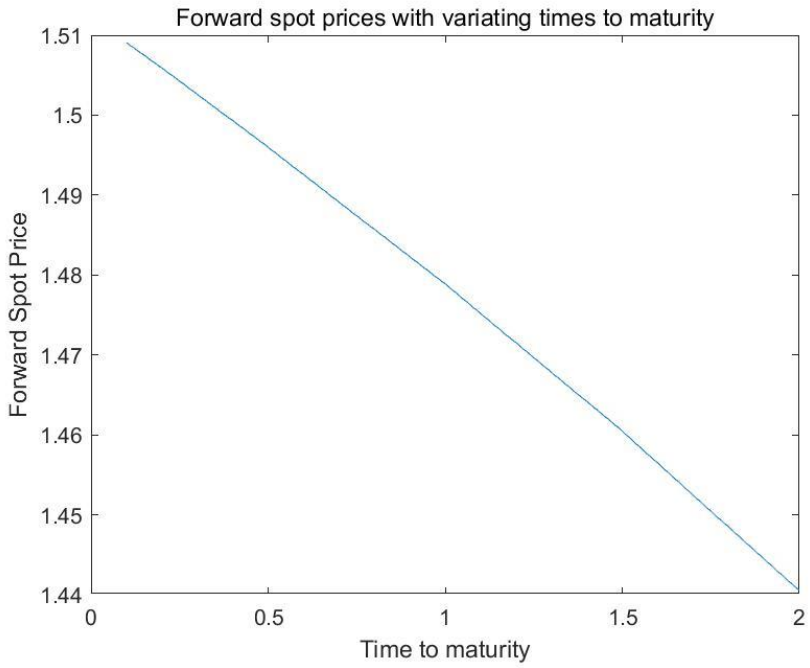
(5) getFwdSpot.m

Exploit the function getRateIntegral to calculate forward spot according to (4).

```
%% exploit the function getRateIntegral
fwdSpot = curve.spot*exp(getRateIntegral(curve.curvedata, T+curve.tau));
```

Code 2.3.2: getFwdSpot.m

To **test** this function, we look at the **output**. We expect it continuously decreasing since foreign interest rate is larger than domestic. We see it actually behaves as we expect, so it passes the test.



Output/Test 2.3.2: getFwdSpot.m

(6) getStrikeFromDelta.m

Secant and Fixed-Point Theorem are not appliable here. We use the simple ***bisection*** method on scalar input since delta as a function of strike is monotonic. Specifically, deltas decrease with strike for call and increase with strike for put. We can unify the expression by multiplying the flag variable ***cp***.

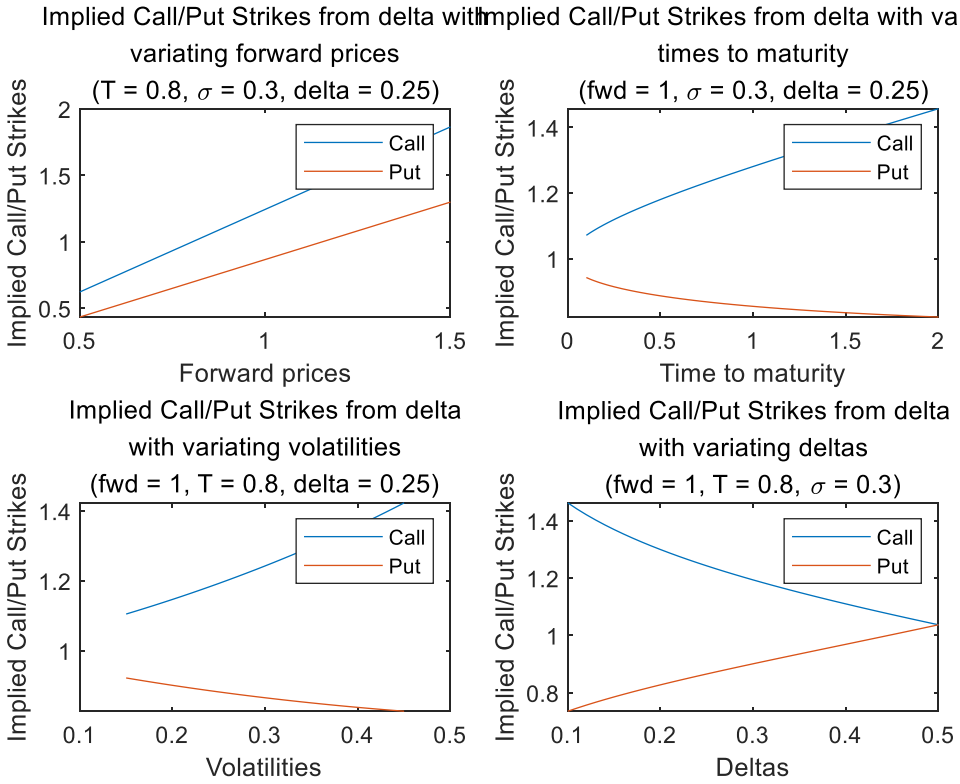
```
%% apply bisection to do root-searching due to monotonicity

K1 = 1e-10; % initial lower bound of guessed K
K2 = 1e+10; % initial upper bound of guessed K
K_tol = 1e-5; % tolerance of guessed K

while abs(K2-K1)>K_tol
    K=(K1+K2)/2;
    d1 = (log(fwd)-log(K))/(sigma*sqrt(T))+1/2*sigma*sqrt(T);
    delta_estimated = normcdf(cp*d1);
    if cp*(delta_estimated-delta)>0
        K1=K;
        % if estimated delta is larger than actual delta (adjusted by cp flag), K is underestimated
    else
        K2=K;
        % if estimated delta is lower than actual delta (adjusted by cp flag), K is overestimated
    end
 end
```

Code 2.4: getStrikeFromDelta.m

To illustrate the **output**, we plot this function according to some inputs. We expect smoothness, and that call/put strikes both increase in forward price; also that call strike increase in time to maturity and volatility (while these two are also positively correlated, so it makes sense) and put is the opposite; that call strike decreases in delta while put increases which also makes sense since call/put deltas are negatively correlated in modulus by (11). As we see, this function also passes **output test**.

We also do two more **tests**:

1. We can calculate the analytical solution for strike. That should give a quite close result. Test passed!
2. If we calculate delta with finite differences of getBlackCall.m taking the strike we obtain from here as input, then the delta should match our original delta. We take 1e-5 as a threshold of the difference since we are dealing with floating point numbers. Test passed!



Output/Test 2.4: getStrikeFromDelta.m

```
%% test getStrikeFromDelta 1 - analytical solution for K
%In this test, we analytically calculate the value of K using predetermined values
%If the function works well, the numerically calculated value should coincide with
%the analytical result
%Analytically, K=fwd*exp(-cp*sigma*sqrt(T)*(invnorm(delta)-1/2*sigma*sqrt(T)))
%Set sigma=T=1, delta=0, then if the function works well, K should be fwd*exp(-1/2)
sigma=2;
T=1;
fwd=1.5;
cp=1;
delta = 0.25;
K_num = getStrikeFromDelta(fwd , T, cp , sigma , delta );
K_ana = fwd*exp(-cp*sigma*sqrt(T)*(norminv(delta)-1/2*sigma*sqrt(T)));
if abs(K_num-K_ana)<1e-5
  disp('the function works well')
else
  disp('error!please check the function!')
end
```

Test 2.4: getStrikeFromDelta.m – match analytical solution

```
%% test getStrikeFromDelta 2 - match getBlackCall
fwd = 1.5;
T = 0.8;
cp = 1;
sigma = 0.3;
delta = 0.25;
h = 1e-7;
K = getStrikeFromDelta(fwd,T,cp,sigma,delta);
delta_fit = (getBlackCall(fwd+h,T,K,sigma)-getBlackCall(fwd,T,K,sigma))/h;
if abs(delta-delta_fit)>1e-5
    error("getStrikeFromDelta does not match getBlackCall");
else
    fprintf("\ngetStrikeFromDelta.m passes test: matches getBlackCall.m.\n Also continuous and smooth, please check output_test!\n\n");
end
```

Test 2.4: getStrikeFromDelta.m – match getBlackCall

(7) makeSmile.m and its curve object

Obtain strikes from deltas first. Here we apply the function ***arrayfun()*** in MATLAB to vectorize and make things faster.

```
%% 1. assert vector dimension match
assert(length(cps)==length(deltas) && length(deltas)==length(vols),...
    'Dimension does not match! Please check!');
```

Code 2.5.1: makeSmile.m-1

```
%% 2. resolve strikes using deltaToStrikes and obtain call price
% deltas_sorted = deltas(vols_index);
fwd = getFwdSpot(fwdCurve, T); % get corresponding forward price
Ks = arrayfun(@(x,y,z) getStrikeFromDelta(fwd, T, x, y, z),...
        cps, vols, deltas,'UniformOutput',false); % get strike price
Ks = cell2mat(Ks);
[Ks_sorted, Ks_index] = sort(Ks); % sort strike price
vols_sorted = vols(Ks_index); % sort volatilities corresponding to strikes
Ks_check = [0 Ks_sorted]; % add K_{0} = 0
```

Code 2.5.1: makeSmile.m-2

Then execute no-arbitrage check for (9). We first ***sort()*** the vectors according to strike first for the purpose of interpolation. We add a $K_0 = 0$ on the left of the strike vector. Since the corresponding volatility is not given, we cannot get price for $K_0 = 0$. However, a call with zero strike is typically not of interest, so according to (8), we just

set the call price as below (i.e., assume no arbitrage just simply holds in $[K_0, K_1]$):

$$\frac{C_1 - C_0}{K_1 - K_0} = -1$$

```
%% 3. check arbitrages
% get call price
us = getBlackCall(fwd, T, Ks ,vols);
us_sorted = us(Ks_index);
% set (c(1)-c(0))/(k(1)-k(0)) = -1
u0 = us_sorted(1)+Ks_sorted(1);
us_check = [u0 us_sorted];

% check no arbitrage (9)
check = (us_check(3:end)-us_check(2:end-1))./...
    (Ks_check(3:end)-Ks_check(2:end-1)) < ...
    (us_check(2:end-1)-us_check(1:end-2))./...
    (Ks_check(2:end-1)-Ks_check(1:end-2));
check = sum(check);
if check > 0
    error('No arbitrage not satisfied!');
end
```

Code 2.5.1: makeSmile.m-3

Finally, compute the interpolation coefficients. We use the ***spline()*** function in MATLAB. Then we still need to calculate coefficients for hyperbolic tangent at the boundary.

We just follow the step suggested in 2.5 and one thing to note is that there can be two possible sets of hyperbolic tangent coefficients. However, they return the same extrapolation result when our input is out of the boundary simply by the fact that

$$\mathrm{a} \cdot \tanh(b) = (-a) \cdot \tanh(-b)$$

We just pick up the positive square root of 0.5 to calculate b_R and b_L.

$$\begin{aligned} |\sigma'(K_L)| &= 0.5\,|\sigma'(K_1)| \\ |\sigma'(K_R)| &= 0.5\,|\sigma'(K_N)| \end{aligned} \quad \Longrightarrow \quad \begin{aligned} 0.5 &= \tanh^2(b_R\,(K_R - K_N)) \\ 0.5 &= \tanh^2(b_L\,(K_1 - K_L)) \end{aligned}$$

Another thing to note is that we use the function ***polyder()*** to calculate coefficients for derivative of polynomials. Since under constant volatility case, we may have

polynomial less than order 3, then it may not match the dimension of ***value***, where we save the power term of the difference between interpolation point and grid point. Therefore, we adjust by extracting corresponding elements in ***value*** (otherwise aR may be of size larger than 1).

```
% 4. compute spline coefficients (for interpolation)
% 5. compute parameters aL , bL , aR and bR (for extrapolation)
coefs = spline(Ks_sorted,vols_sorted).coefs;
KL = Ks_sorted(1)*Ks_sorted(1)/Ks_sorted(2);
KR = Ks_sorted(end)*Ks_sorted(end)/Ks_sorted(end-1);
bR = atanh(sqrt(0.5))/(KR-Ks_sorted(end));
bL = atanh(sqrt(0.5))/(Ks_sorted(1)-KL);
    % bR = atanh(-sqrt(0.5))/(KR-Ks_sorted(end));
    % bL = atanh(-sqrt(0.5))/(Ks_sorted(1)-KL);
    % can use the negative root of 0.5, then we get different/opposite of
    % the a, b coefficients, but the interpolation results are the same
    % since tanh is an odd function in the sense that
    % a*(exp(b)-exp(-b))/(exp(b)+exp(-b)) = (-a)*(exp(-b)-exp(b))/(exp(-b)+exp(b))
derL = coefs(1,3);
dercoef = polyder(coefs(end,:)); % can be vector of size 3,2,or 1
value = [(Ks_sorted(end)-Ks_sorted(end-1))*(Ks_sorted(end)-Ks_sorted(end-1))...
    ;(Ks_sorted(end)-Ks_sorted(end-1));1];
derR = dercoef*value(end-length(dercoef)+1:end);
% say, if length(dercoef) = 2 (parabolic essentially for the spline), then we pick up value(2:3)
aL = -derL/bL;
aR = derR/bR;
```

Code 2.5.1: makeSmile.m-4

We save the sorted strike grids and sorted volatility grids and the extrapolation coefficients in a struct type object **curve**.

```
%% 6. save results
curve.Kvec_sorted = Ks_sorted;
curve.vols_sorted = vols_sorted;
curve.extra = [aL bL aR bR];
```

Code 2.5.1: makeSmile.m-5

(8) getSmileVol.m

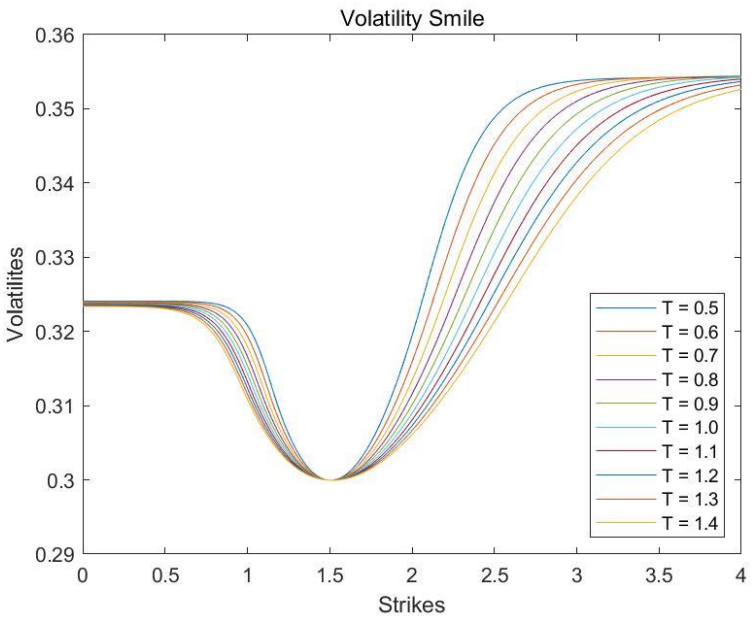
We just use the ***spline()*** function to implement interpolation and when our input is out of the boundary of my grid points, we adjust by coefficients calculated and saved

in the curve object, namely a_L, a_R, b_L and b_R.

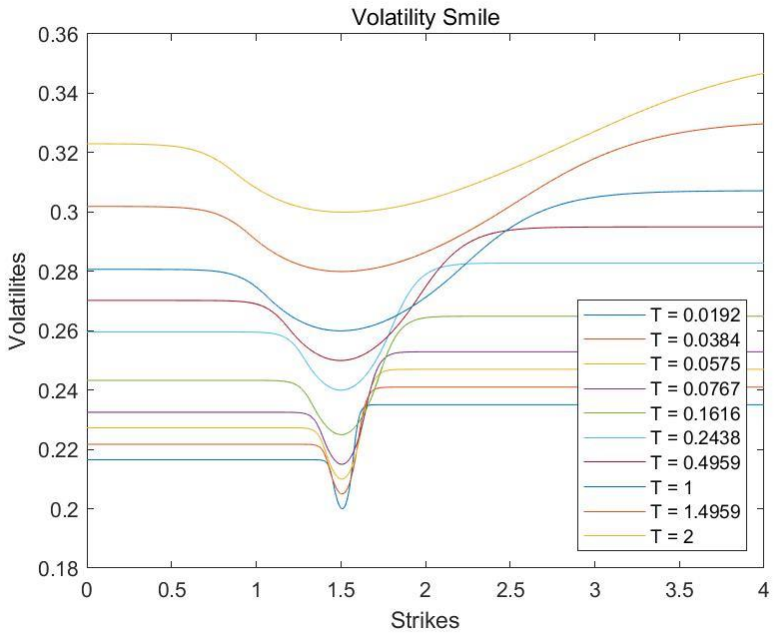
```
%% extract elements from pre-computed curve data
K_grid = curve.Kvec_sorted;
vols = curve.vols_sorted;
extra = curve.extra;

%% implement interpolation and adjust extrapolation
vol_interp = spline(K_grid,vols,Ks);

vol_interp(Ks < K_grid(1)) = vols(1)*ones(size(Ks(Ks < K_grid(1)))) + ...
    extra(1)*tanh(extra(2)*(K_grid(1)-Ks(Ks < K_grid(1))));
vol_interp(Ks > K_grid(end)) = vols(end)*ones(size(Ks(Ks > K_grid(end)))) + ...
    extra(3)*tanh(extra(4)*(Ks(Ks > K_grid(end))-K_grid(end)));

%% save results
vols = vol_interp;
```

Code 2.5.2: getSmileVol.m

The **output** looks as follows. We obtain a smooth smile shape of the curve with different T. Also, when the strike is extremely small or large, the volatility converges to some certain level.



Output/Test 2.5: getSmileVol.m (same vol grids)



Output/Test 2.5: getSmileVol.m (original data)

We also conduct additional **test** whether our function can return the original data (namely the grid points we use). It passes the test.

```
%% test getSmileVol 1 - return original data
fwd = getFwdSpot(fwdCurve,Ts(end));
Ks = cell2mat(arrayfun(@(x,y,z) getStrikeFromDelta(fwd,Ts(end),x,y,z),cps,vols(end,:),deltas,'UniformOutput',false));
vols_fit = getSmileVol(smile,Ks);
if sum(abs(vols_fit-vols(end,:)))>1e-10
    error("getSmileVol does not return original data!");
else
    fprintf("\ngetSmileVol.m passes test: return original volatilities. \n Also smooth and smile-shaped, please check output_test!\n\n");
end
```

Test 2.5: getSmileVol.m – return original data

(9) makeVolSurface.m and its volSurface object

What we actually should do here is essentially repeating the process of **getSmileVol.m** along the dimension of "Ts". We use the function ***cellfun()***, which is similar to ***arrayfun()*** and allow us to vectorize.

```
%% input check
assert(length(cps) == length(deltas) == size(vols,2) ||...
    length(Ts) == size(vols,1),...
    "Input dimension does not match. Please check!");
```

Code 2.6.1: makeVolSurface.m-1

Once obtain volatility smile for each T, we can apply the no arbitrage check suggested by (10). For readability, we just do in a for loop.

```
%% obtain some precomputed data
[Ts_sorted, Ts_index] = sort(Ts); % sort time to maturity
vols_sorted = vols(Ts_index,:); % sort the volatility matrix by rows according to the order of Ts
Ts_cell = num2cell(Ts_sorted);
vols_cell = mat2cell(vols_sorted,ones(1,size(vols,1)),size(vols,2));
% obtain smile curve for each T
smile_curve = cellfun(@(x,y) makeSmile (fwdCurve , x, cps , deltas , y ),...
   Ts_cell, vols_cell, 'UniformOutput',false);
% obtain forward spot price
fwds = cell2mat(arrayfun(@(x) getFwdSpot(fwdCurve, x),Ts_sorted,'UniformOutput',false));
% check no arbitrage
for i=1:length(Ts_sorted)-1
    K1 = fwds(i);
    K2 = fwds(i+1);
    V1 = getSmileVol(smile_curve{i},K1);
    V2 = getSmileVol(smile_curve{i+1},K2);
    C1 = getBlackCall(K1,Ts_sorted(i),K1,V1);
    C2 = getBlackCall(K2,Ts_sorted(i+1),K2,V2);
    assert(C1<C2, "No arbitrage not satisfied!");
end
```

Code 2.6.1: makeVolSurface.m-2

We save the sorted T grids and sorted volatility matrix grids, forward price for each T, forward Curve object obtained above and smile curve for each T in the **volSurface** object.

```
%% save data
volSurf.Ts = Ts_sorted;
volSurf.vols = vols_sorted;
volSurf.fwds = fwds;
volSurf.fwdCurve = fwdCurve;
volSurf.smile = smile_curve;
```

Code 2.6.1: makeVolSurface.m-3

(10) getVol.m

We just apply interpolation following steps in 2.6. The key step again is to locate the T input in the Ts grid vector. What we should pay attention to is the extreme case when T is smaller than our first time grid point and when T is exactly our last time grid point.

```
%% implement interpolation
smile = volSurf.smile;
% locate the input maturity
index = sum(T >= Ts_grid);
% calculate forward spot price
fwdCurve = volSurf.fwdCurve; fwds_grid = volSurf.fwds; fwd = getFwdSpot(fwdCurve,T);
```

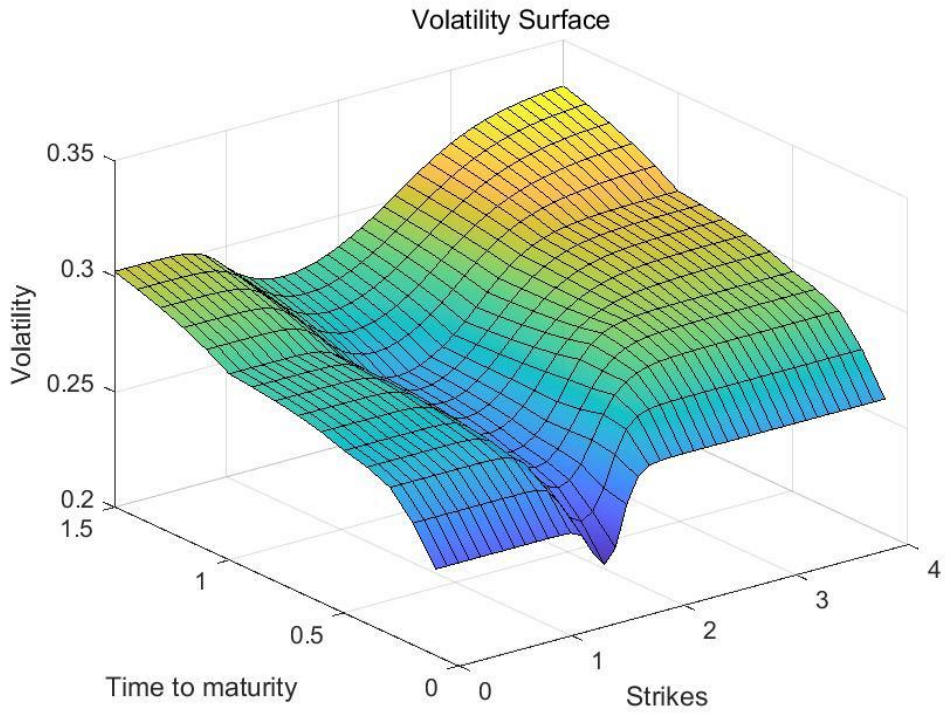
Code 2.6.2: getVol.m-1

```
if index == length(Ts_grid) % T = TN
    fwd_r = fwds_grid(end); K_rs = fwd_r/fwd.*Ks;
    vols = getSmileVol(smile{end},K_rs);
elseif index == 0 % T < T1
    fwd_r = fwds_grid(1); K_rs = fwd_r/fwd.*Ks;
    vols = getSmileVol(smile{1},K_rs);
else
    T_l = Ts_grid(index); T_r = Ts_grid(index+1); dT = T_r-T_l;
    % calculate forward spot price of adjacent time grid
    fwd_l = fwds_grid(index); fwd_r = fwds_grid(index+1);
    % calculate K grid points (finding moneyness equivalent strikes)
    K_l = fwd_l/fwd.*Ks;
    K_r = fwd_r/fwd.*Ks;
    % calculate volatility grid according to smile
    sigma_l = getSmileVol(smile{index},K_l);
    sigma_r = getSmileVol(smile{index+1},K_r);
    % implement interpolation
    vols = (T_r-T)*T_l/dT.*sigma_l.*sigma_l + (T-T_l)*T_r/dT.*sigma_r.*sigma_r;
    vols = sqrt(vols/T);
end
```

Code 2.6.2: getVol.m-2

We then illustrate the **output**. We obtain the so-called volatility surface. Similarly,

we obtain a smooth smile shape along the K dimension. It increases continuously along the T dimension but notice that it's not smooth due to our application of linear interpolation scheme.



Output/Test 2.6: getVol.m

We also conduct additional **test** whether our function can return the original data (namely the grid points we use). It passes the test.

```
%% test getVol 1 - return original data
vols_mat = zeros(size(vols));
for i=1:length(Ts)
    fwd = getFwdSpot(fwdCurve,Ts(i));
    Ks = cell2mat(arrayfun(@(x,y,z) getStrikeFromDelta(fwd,Ts(i),x,y,z),cps,vols(i,:),deltas,'UniformOutput',false));
    vols_mat(i,:) = getVol(volSurface,Ts(i),Ks);
end
if sum(abs(vols_mat-vols))>1e-10
    error("getVol does not return original data!");
else
    fprintf("\ngetVol.m passes test: return original volatilities. \n Also smooth and smile-shaped, please check output_test!\n\n");
end
```

Test 2.6: getVol.m – return original data

(11) getCdf.m and getPdf.m

These two functions are essentially computing first derivative so we use the same method. We apply ***Richardson*** for strikes which are not close to zero. But for those very close to zero, more precisely, ***less than twice of the bump size*** by the nature of Richardson, we simply apply the forward differences.

One thing to note is volatility is also a function of strike. We obtain volatility by getVol.m.

Also, since we exploit getVol.m, so ***if the input T for getCdf.m is larger than our last time grid point***, it will return ***error*** due to the call of getVol.m.

By the way, we select ***a bump size of 1e-4*** by test to reduce oscillations but keep as accurate as possible.

```
%% We are going to apply Richardson
% set bump size
% if h is chosen too small, then it will suffer from oscillations
h = 1e-4;

% obtain forward price
fwdCurve = volSurf.fwdCurve;
fwd = getFwdSpot(fwdCurve,T);

% obtain call price in neighborhood
for i = 1:2
    eval(['us_',num2str(i),'hp = getBlackCall (fwd, T, Ks(Ks>2*h)+i*h , getVol(volSurf,T,Ks(Ks>2*h)+i*h));']);
    eval(['us_',num2str(i),'hm = getBlackCall (fwd, T, Ks(Ks>2*h)-i*h , getVol(volSurf,T,Ks(Ks>2*h)-i*h));']);
end

% apply Richardson for nonzero strikes
cdf(Ks>2*h) = 2/3/h*(us_1hp-us_1hm)-1/12/h*(us_2hp-us_2hm)+1;
% apply forward derivative for close-to-zero strikes
cdf(Ks <= 2*h) = (getBlackCall(fwd,T,h*ones(size(Ks(Ks<=2*h))),getVol(volSurf,T,Ks(Ks<=2*h)+h))...
    -getBlackCall(fwd,T,zeros(size(Ks(Ks<=2*h))),getVol(volSurf,T,Ks(Ks<=2*h))))/h+1;
```

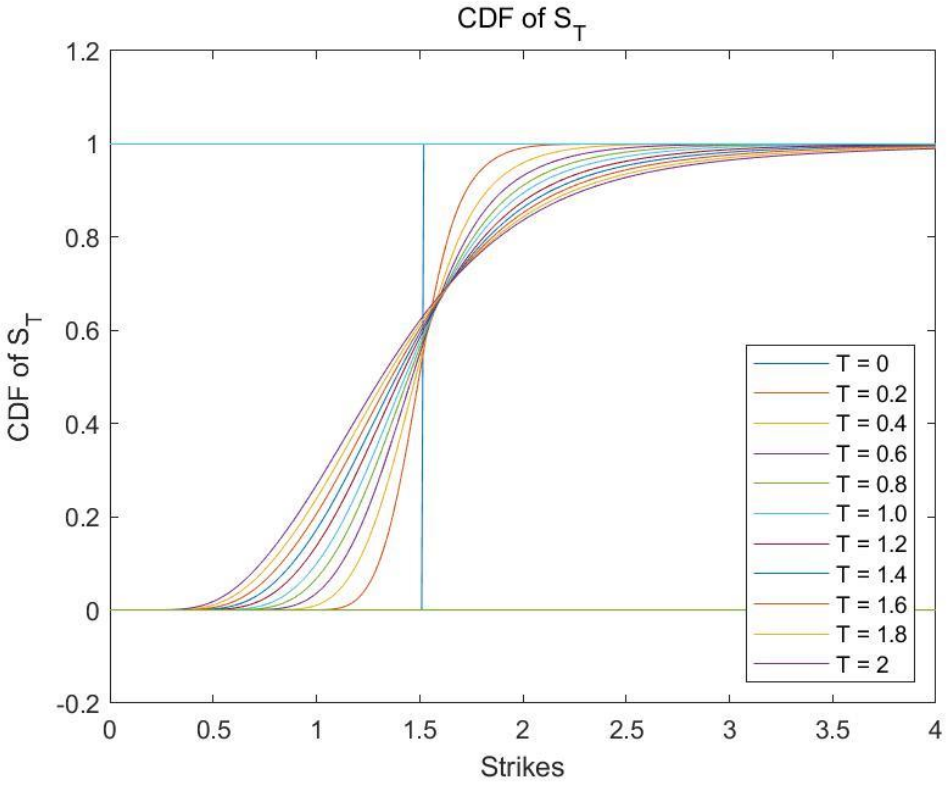
Code 2.7.1: getCdf.m

```
%% We are going to apply Richardson again to cdf
h = 1e-4;
% will oscillate or even overflow once below some certain level
for i = 1:2
    eval(['cdf_',num2str(i),'hp = getCdf (volSurf, T, Ks(Ks>2*h)+i*h);']);
    eval(['cdf_',num2str(i),'hm = getCdf (volSurf, T, Ks(Ks>2*h)-i*h);']);
end
% apply Richardson to nonzero strikes
pdf(Ks>02*h) = 2/3/h*(cdf_1hp-cdf_1hm)-1/12/h*(cdf_2hp-cdf_2hm);
% apply forward derivative to close-to-zero strikes
pdf(Ks<=2*h) = (getCdf (volSurf, T, h*ones(size(Ks(Ks<=2*h))))-...
    (getCdf (volSurf, T, Ks(Ks<=2*h))))/h;
```

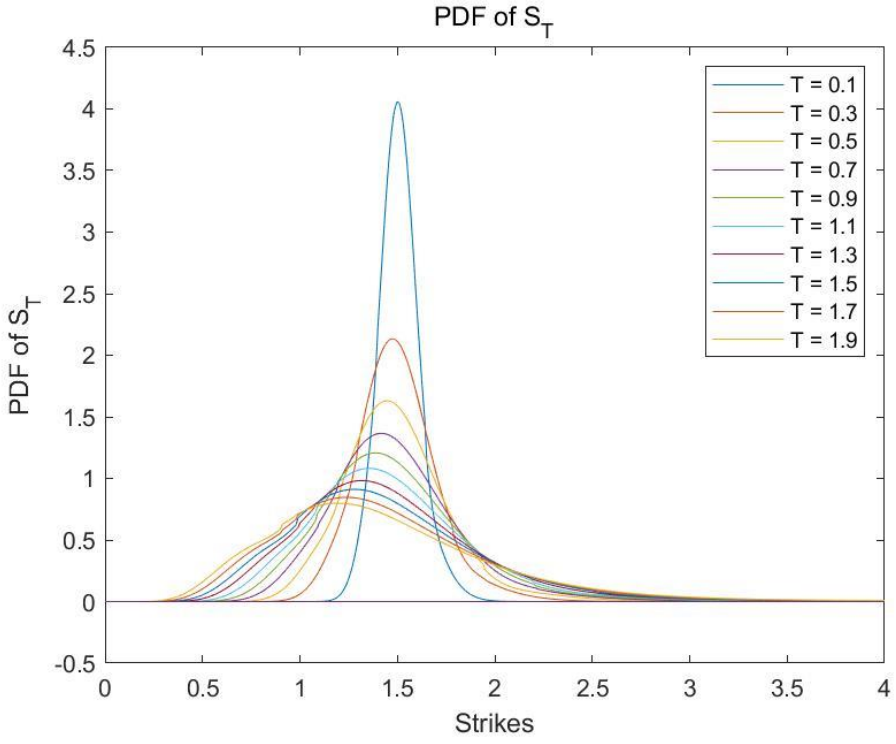
Code 2.7.2: getPdf.m

We plot the **output** as follows. We expect that

1. Both plots are continuous.
2. CDF increases from 0 up to 1, and PDF should be above zero and looks like bell-shaped since we apply the BS formula in (5), which assumes log-normal stock price process, to calculate the derivatives.
3. Peak of PDF shifts to the left as T increases because forward spot price is decreasing in T.
4. Larger T, wider CDF/PDF, namely larger variance (given by $\sigma^2 T$ in BS).
5. In the extreme case that T=0, and CDF becomes a discrete line and not differentiable (no PDF) as expected.
6. When CDF is constant, its derivative, namely PDF is zero.



Output/Test 2.7: getCdf.m



Output/Test 2.7: getPdf.m

We also **test** that PDF integrates to 1 and the mean of PDF is close to the forward price.

```
%% Test getpdf 2: Integration of pdf should be 1
T = linspace(0.001,1.999,100);%Set 100 grid point from 0.001 to 1.999

% Calculate the integral value for each time and ortanize in a vector integTest2
integTest2 = zeros(1,100);
for ii = 1:100
    integTest2(1,ii)=integral(@(k)getPdf(volSurface,T(ii),k),0,Inf);
end

% Test wheather the each value of integTest2 is 1
if sum(abs(integTest2-1)>1e-5) == 0
    disp('Test 2 passed: integration of pdf is 1');
else
    disp('Test 2 failed: please check the function!');
end
```

Test 2.7: getPdf.m – integrate to 1

```
%% Test 3: the mean of the risk neutral probability should be the forward

% Calculate the forwards with each T and ortanize in a vector fwdTest3
fwdTest3 = zeros(1,100);
for ii = 1:100
    fwdTest3(1,ii) = getFwdSpot(fwdCurve, T(ii));
end

% Calculate the mean of risk neutral probability with each T and ortanize in a vector meanTest3
meanTest3 = zeros(1,100);
for jj= 1:100
    meanTest3(1,jj) = integral(@(k)(getPdf(volSurface,T(jj),k).*k),0,+Inf);
end

% Test wheather each element of fwdTest3 equals to each element of meanTest3
if sum(abs(fwdTest3 - meanTest3)>1e-5) == 0
    disp('Test 3 passed: the mean of the risk neutral probability is the forward');
else
    disp('Test 3 failed: please check the function!');
end
```

Test 2.7: getPdf.m – mean forward price

(12) getEuropean.m

We first set default value for the interval variable “int”.

```
%% set default value for the 4th input
if nargin < 4
    ints = [0,+Inf];
end
```

Code 2.8: getEuropean.m-1

We won't do integration over the whole interval from 0 to infinity. We search for an upper bound recursively where CDF is already 1. Then we obtain PDF and apply the mid-point rule[5] to do numerical integration over the interval from 0 to the upper bound.

```
%% get upper bound for integral
ub = 3;
while getCdf(volSurface,T,ub)<1
    ub = ub+1;
end
```

Code 2.8: getEuropean.m-2

We use the same bump size of 1e-4[6]. If there exists discontinuity, we first locate the discontinuity point using the same method as above. Then we assume the payoff is ***left-continuous***[7] at this point and do integration in two separated intervals. We allow ***multiple discontinuities*** here.

---

[5] We have also tried Simpson and Simpson-Richardson, and found that there is actually no improvement. We also tried to drop the zero-payoff region when integration, since we aim to value a European vanilla option in this function and any nonvanilla can be represented as combination of vanilla. But again, it does not improve much for speed. Please check the corresponding function!

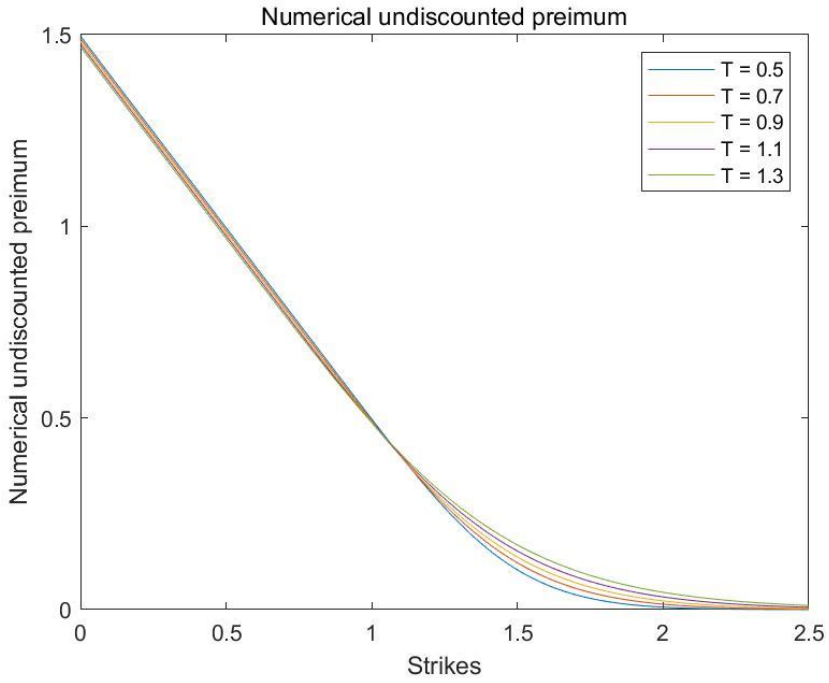
[6] However, it makes this function run too slow (same result for 1e-5 but much slower, 1e-4 is like a cutoff). If we choose a bump size of 1e-3, it will return close result faster (***0.24 sec versus 1.4 sec*** on average; if choose 0.5e-3, the result is different at the 4th decimal point and the time is ***0.3sec***; if choose something between 1e-4 and 0.5e-3, the result is the same as 0.5e-3 with longer running time, like ***0.5 sec*** for 0.2e-3), but at the meantime will lower the accuracy of some of our tests below. There is always a tradeoff between accuracy and efficiency.

[7] One can easily change into right continuous by modifying the code.

```
%% obtain pdf
h = 1e-4; % if choose too small bump size, the function will be very slow!
interval = ints(1):h:ub;
pdf = getPdf(volSurface, T, interval);
%% implement integration
% we apply the midpoint rule and allow multiple breaking points
loc = sum(ub >= ints);
u = 0;
lb = 1;
for i = 1:loc
    if i == length(ints)
        break;
    else
        index = sum(ints(i+1) >= interval);
    end
    y1 = payoff(interval(lb:index)+0.5*h);
    I1 = sum(h*y1.*pdf(lb:index));
    u = u + I1;
    lb = index+1;
end
```

Code 2.8: getEuropean.m-3

We again plot the **output** using a call option payoff function. The output is decreasing in K, as expected. One thing to note is that when strike is low, the payoff actually is same across all T. But when strike is high enough, typically larger than those forward prices, then ***payoff with low T is also lower***, this is because we have thinner tails for lower T in our PDFs plot.



Output/Test 2.8: getEuropean.m

We do the following **tests**:

Test 1: check the performance with a wide range of inputs, particularly multiple discontinuities indicated by the last input. It works.

```
%% test getEuropean 1 - multiple discontinuities
Ts = 0:0.1:2;
arrayfun(@(y) getEuropean(volSurface,y,@(x) max(x-1,0)),Ts,'UniformOutput',false);
arrayfun(@(y) getEuropean(volSurface,y,@(x) max(x-1,0),[0,+Inf]),Ts,'UniformOutput',false);
arrayfun(@(y) getEuropean(volSurface,y,@(x) max(x-1,0),[0,1,+Inf]),Ts,'UniformOutput',false);
arrayfun(@(y) getEuropean(volSurface,y,@(x) max(x-1,0),[0,0.3,0.5,0.7,+Inf]),Ts,'UniformOutput',false);
arrayfun(@(y) getEuropean(volSurface,y,@(x) max(x-1,0),[0,0.9]),Ts,'UniformOutput',false);
arrayfun(@(y) getEuropean(volSurface,y,@(x) max(x-1,0),[0.1,0.9]),Ts,'UniformOutput',false);
```

Test 2.8: getEuropean.m – multiple discontinuities

Test 2-1: put-call parity holds with 4 digits accuracy.

Test 2-2: same result under degenerate volatility surface.

```
%% test getEuropean 2-1 - put-call parity
K = 1;
T = 0.8;
call = getEuropean(volSurface,T,@(x) max(x-1,0));
put = getEuropean(volSurface,T,@(x) max(1-x,0));
I1 = call - put;
I2 = getFwdSpot(fwdCurve,T) - K;
if abs(I1-I2) > 1e-4
    disp("Put-call parity not satisfied!");
else
    disp("Put-call parity satisfied!");
end
```

Test 2.8: getEuropean.m – put-call parity

```
%% test getEuropean 2-2 - put-call parity under degenerate volatility surface
vols_d = 0.3*ones(size(vols));
volSurface_d = makeVolSurface(fwdCurve, Ts, cps, deltas, vols_d);
K = 1;
T = 0.8;
call = getEuropean(volSurface_d,T,@(x) max(x-1,0));
put = getEuropean(volSurface_d,T,@(x) max(1-x,0));
I1 = call - put;
I2 = getFwdSpot(fwdCurve,T) - K;
if abs(I1-I2) > 1e-4
    disp("Put-call parity not satisfied!");
else
    disp("Put-call parity satisfied!");
end
```

Test 2.8: getEuropean.m – put-call parity under degenerate vol

Test 3-1: compare the numerical premium for call with the analytical one from getBlackCall.m. Obviously their results are close. It passes this result match test but only with 2 digits accuracy since we assume constant volatility in BS.

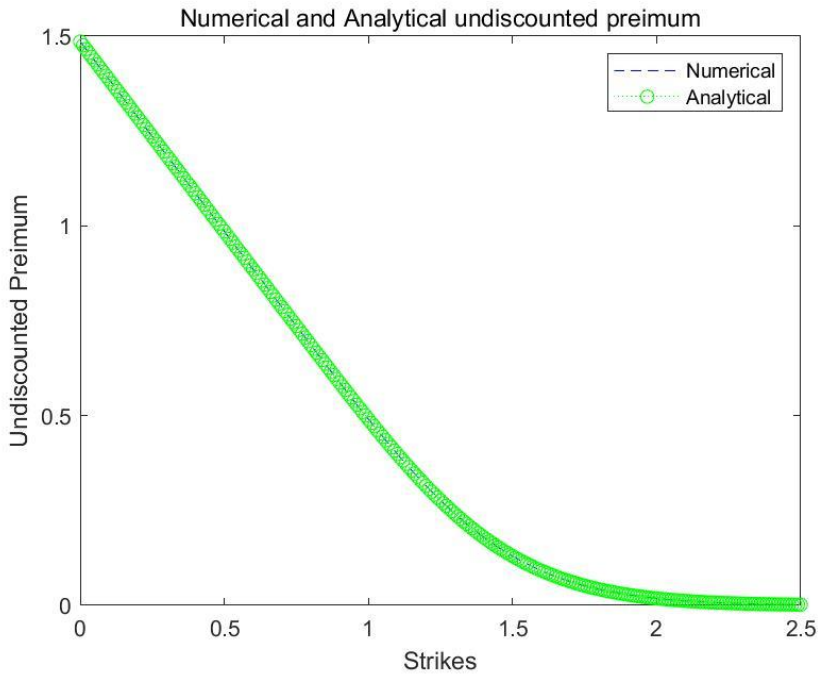
Test 3-2: under degenerate volatility surface, the accuracy raises up to 4 digits.

```
%% test getEuropean 3-1 - match getBlackCall
K = 1;
T = 0.8;
European = getEuropean(volSurface,T,@(x) max(x-K,0));
fwd = getFwdSpot(fwdCurve,T);
vol = getVol(volSurface,0.8,fwd);
vol = vol*ones(size(K));
Black = getBlackCall(fwd,0.8,K,vol);
if abs(European-Black) > 1e-2
    disp("Not match getBlackCall!");
else
    disp("Match getBlackCall!");
end
```

Test 2.8: getEuropean.m – match getBlackCall

```
%% test getEuropean 3-2 - match getBlackCall under degenerate volatility surface
vols_d = 0.3*ones(size(vols));
volSurface_d = makeVolSurface(fwdCurve, Ts, cps, deltas, vols_d);
K = 1;
T = 0.8;
European = getEuropean(volSurface_d,T,@(x) max(x-K,0));
fwd = getFwdSpot(fwdCurve,T);
vol = getVol(volSurface_d,0.8,fwd);
vol = vol*ones(size(K));
Black = getBlackCall(fwd,0.8,K,vol);
if abs(European-Black) > 1e-4
    disp("Not match getBlackCall under degenerate vol!");
else
    disp("Match getBlackCall under degenerate vol!");
end
```

Test 2.8: getEuropean.m – match getBlackCall under degenerate vol



Output/Test 2.8: getEuropean.m and getBlackCall.m

Test 4-1: compare the numerical premium for binary option with the analytical one. They match with 2 digits accuracy since we assume constant volatility in BS.

Test 4-2: under degenerate volatility surface, the accuracy raises up to 4 digits.

```
%% test getEuropean 4-1 - nonvanilla option (binary option)
B1 = 1.5;
B2 = 1;
K = 1.4;
T = 0.8;
payoff = @(x) B1*(x>K)+B2*(x<=K);
num_pre = getEuropean(volSurface, T, payoff);
% we know how to value a binary option
vol = getVol(volSurface,T,K);
fwd = getFwdSpot(fwdCurve,T);
d2 = log(fwd/K)/vol/sqrt(T) - 0.5*vol*sqrt(T);
ana_pre = B1*normcdf(d2)+B2*normcdf(-d2);
if abs(num_pre-ana_pre) > 1e-2
    disp("Binary option numerical value does not match analytical value!");
else
    disp("Binary option numerical value matches analytical value!");
end
```

Test 2.8: getEuropean.m – binary option

```
%% test getEuropean 4-2 - nonvanilla option (binary option) under degenerate volatility surface
vols_d = 0.3*ones(size(vols));
volSurface_d = makeVolSurface(fwdCurve, Ts, cps, deltas, vols_d);
smile_d = makeSmile(fwdCurve, Ts(end), cps, deltas, vols_d(end,:));
B1 = 1.5;
B2 = 1;
K = 1.4;
T = 0.8;
payoff = @(x) B1*(x>K)+B2*(x<=K);
num_pre = getEuropean(volSurface_d, T, payoff);
% we know how to value a binary option
vol = getVol(volSurface_d,T,K);
fwd = getFwdSpot(fwdCurve,T);
d2 = log(fwd/K)/vol/sqrt(T) - 0.5*vol*sqrt(T);
ana_pre = B1*normcdf(d2)+B2*normcdf(-d2);
if abs(num_pre-ana_pre) > 1e-4
    disp("Binary option cannot be approximated under degenerated volatility surface!");
else
    disp("Binary option can be approximated under degenerated volatility surface!");
end
```

Test 2.8: getEuropean.m – binary option under degenerate vol

Test 5: we price an option allowing holder to choose between call and put at maturity and compare with a combination of call and put. Results are close up to at least 10 digits accuracy.

```
%% test getEuropean 5 - nonvanilla option represented by vanilla
K = 1.4; T = 0.8;
nonvan_pay = @(x) max(x-K,K-x);
% an option that allows holder to choose between call and put
vancall_pay = @(x) max(x-K,0);
vanput_pay = @(x) max(K-x,0);
nonvan_pre = getEuropean(volSurface, T, nonvan_pay);
vancall_pre = getEuropean(volSurface, T, vancall_pay);
vanput_pre = getEuropean(volSurface, T, vanput_pay);
if abs(nonvan_pre-vancall_pre-vanput_pre) > 1e-10
    disp("Nonvanilla option cannot be approximated by vanilla!");
else
    disp("Nonvanilla option can be approximated by vanilla!");
end
```

Test 2.8: getEuropean.m – nonvanilla approximated by vanilla

In addition, we can use this function to **test getPdf.m** in the sense that integral of

PDF should be equal to 1 (we ***round*** the result and see whether it is exactly 1) and the mean of PDF should be the forward price (we set a ***threshold*** of 1e-10). It passes both tests for a wide range of inputs.

2. Other tests

We write a quick test for convenience as well as a specific test for each function. The quick test aims to confirm some properties as follows:

function	Property/purpose
Running time	Not too slow (1.4 sec on average)
getBlackCall	Zero strike
getRateIntegral getSmileVol getVol	Return original data (10 digits accuracy)
getFwdSpot	Decreasing
GetStrikeFromDelta	Match getBlackCall (5 digits accuracy)
getCdf	Increasing up to 1 (0.5% relative error)
getPdf	Positive (0.5% relative error); Bell-shaped; Mean is forward (10 digits accuracy)
getEuropean	Allow multiple discontinuities

3. Personal reflections

(1) Huang Kenghua (A0212105A)

I work hard on section ***2.5 - 2.8***, as well as the file ***test_getBlackCall.m***, ***test_getFwdSpot.m***, ***test_European.m***, ***quick_test.m***, and ***output_test.m***. I also go through the whole task from 2.1 to 2.8 to keep consistency with my teammates and try to write efficient and simple code. I have tried to write vectorized code everywhere to

achieve efficiency, but in some place, we still need to maintain readability. There is a tradeoff.

I encounter some issues in programming - sometimes we come up with a method, write it down, but we cannot make sure that it works for all situations. There can always be some corner situations that your function cannot work, or more precisely, you have not considered, then you need to modify and test again and again. For example, there is some out-of-range error of getVol.m initially. We fix by test.

Another problem I met is oscillations when I compute PDF. There can be oscillations when we use finite differences. It will behave weirdly especially when PDF is around 0, namely that PDF may be negative. The way I address this problem is just ignore it since it's not important when PDF is around 0. I also set a relative error for the proportion of negative PDF for test.

I'm comfortable with the syntax and the theoretical part. But I always agree that you can never understand the key of numerical methods unless you realize them. Programming is different from theory we learn. In terms of programming, we need to keep balance between efficiency, simplicity and readability always in addition to realize our algorithm. Then it needs experience and multiple trials and errors. Therefore, the most struggling part in this project is always the debugging process! I think this course gives me a lot of thinking about programming for numerical methods.

(2) Dang Yifei (A0212116Y)

I tried to write the ***makeFwdCurve.m***, ***getBlackCall.m*** and ***getFwdSpot.m*** and test ***getEuropean.m***. But my unfamiliarity with MATLAB caused many problems. Eventually, Huang Kenghua (A0212105A) helped me to successfully implement and test these functions.

Since I have never used MATLAB before, the biggest problem I encountered is that I don't understand the grammar of this language very well. In order to solve this problem, I read the MATLAB document and search the specific grammatical problems I have on stack overflow. I also asked Ding Xiangyu (A0212112H) and Huang Kenghua (A0212105A) for help. They gave me enormous amount of help.

To be more specific, my teammates asked me to write the getBlackCall.m in a way that can process metrics as variables. At first, I tried to write for loops. But that algorithm can only process vectors and is very slow. After asked Kenghua, I used dot product to allow the getBlackCall.m function to process metrics as inputs.

When testing the getEuropean.m, I wrote a function that calculate the difference between the call option and put option to check the put-call parity. But I met problems when trying to construct the volSurface data. Again, Huang Kenghua helped me out and help to successfully test the put-call parity of this function.

Through this project, I realized my limitation in programming skills. I will try to hone my programming skills in order to finish a project independently. Again, I need to thank my smart and patient teammates, especially Huang Kenghua. This is the best team I have ever cooperated with. They helped me to solve problems I met when doing this project and to have further understanding of the content of this course.

(3) Ding Xiangyu (A0212112H)

In this project, I was responsible for writing the function ***makeDepoCurve.m***, ***getRateIntegral.m***, and I also participated in the testing, optimizing and debugging of ***makeDepoCurve.m***, ***getRateIntegral.m***, ***getCdf.m***, ***getPdf.m*** with Huang Kenghua. In order to test the performance of our functions, I wrote 7 tests in our test files: ***test_getRateIntegral.m***, ***test_getCdf.m*** and ***test_getPdf.m***.

One of the problems I faced during the project was the algorithm of ***getRateIntegral.m***, in which we needed to use discrete forward rate to generate continuous log return by integral.

In order to determine the interval in which a given time occurs, I initially used a while loop, but then I found that a Boolean Operation gave me a much faster result. In addition, in the first version of the program, the curve generated by makeDepoCurve.m only held data of the forward rate, so the we needed to integrate the forward rate from 0 to T. To make the function faster, I then saved the interest rates between 0 and the interval point in the third column of the variable curve, which can be called directly

during integration. Then, the function became faster.

Beyond that, In the process of testing getCdf.m and getPdf.m, I chose 10,000 test points between 0 and 2 years to conduct a test, but the results showed that the function failed near the extreme values t=0, and t=2. We found that the cause of the error was that the function called getVol.m, while it ran into an out of range error. To solve this problem, we added an if judgment at the end of the getvol.m function to identify the extreme input value. In the end, we solved the problem.

(4) Du Bowen (A0212103H)

List What functions I implemented:

I am mainly in charge of ***2.4*** conversion of deltas to strikes part, and conduct tests for 3 get* functions, which are ***2.4 getStrikeFromDelta.m***, ***2.5 getSmileVol.m***, and ***2.6 getVol.m***. In the functions and subscripts above, I mainly applied while loop and root search to get the numerical value of the strikes in delta function. In the test part, I mainly applied plots and graphs to qualitatively determine the validity of these functions.

What issues I faced and how I addressed them:

In the 2.4 getStrikeFromDelta part, the problem I met is that I need to consider the requirements of other teammates. For example, at first, my team decided that the inputs and outputs of 2.4 getStrikeFromDelta should be vectors, however, it turned out that scalar inputs and outputs are what we need, thus I create 2 versions of getStrikeFromDelta, one is scalar version, and the other is vector version.

During the test parts, the problem is mainly on coordination. To test 2.5 and 2.6, I need to use the functions in 2.1, 2.2 and 2.3. If any parts consist of a tiny bug, the results in 2.4, 2.5 and 2.6 might be wrong. Therefore, I need to read and understand the codes in all parts before 2.6 to implement the tests.

(5) Liu Yiming (A0212149M)

List what functions you implemented.

Firstly, I implemented the function of interpolation of implied volatility in strike

direction with my teammate Kenghua Huang. We apply the function arrayfun in MATLAB to vectorize and make things faster. What's more, we execute no-arbitrage check. We first sort the vectors according to strike first for the purpose of interpolation and compute the interpolation coefficients. We used the spline function in MATLAB. Then we still need to calculate coefficients for hyperbolic tangent at the boundary. We just follow the step suggested in ***2.5*** and one thing to note is that there can be two possible sets of hyperbolic tangent coefficients, and pick up the positive square root of 0.5 to calculate bR and bL.

Secondly, I operated the test about A call option forward price obtained with the Black formula and the result is matched.

Describe what issues you faced and how you addressed them.

The first issue that I faced was that my skill in MATLAB was limited at first, with the help of my teammates, I gradually get more familiar with MATLAB.

Another issue that I faced was how to test the data in MATLAB. After discussing with my teammates, I simulate some data and practice for few times, then I know how to operate the function and the result is proper.