computational_hw2

May 11, 2024

It is written by C++

0.1 Question 1

Compare the convergence rates of the four methods below by doing the following: Use the Binomial Method to price a 6-month American Put option with the following information: the risk-free interest rate is 5.5% per annum; the volatility is 25% per annum; the current stock price is $180; and the strike price is $170. Divide the time interval into parts to estimate the price of this option. Use = 20, 40, 80, 100, 200, 500, to estimate the price and draw all resulting prices in one graph, where the horizontal axis measures , and the vertical one the price of the option. (a) Use the binomial method in which u=1/d, d=c− √c2 − 1, c = 1/2 (− + (+ 2)), = (−)/(−) (b) Use the binomial method in which = [(− 2/2) + √], = [(− 2/2) − √], = 1/2

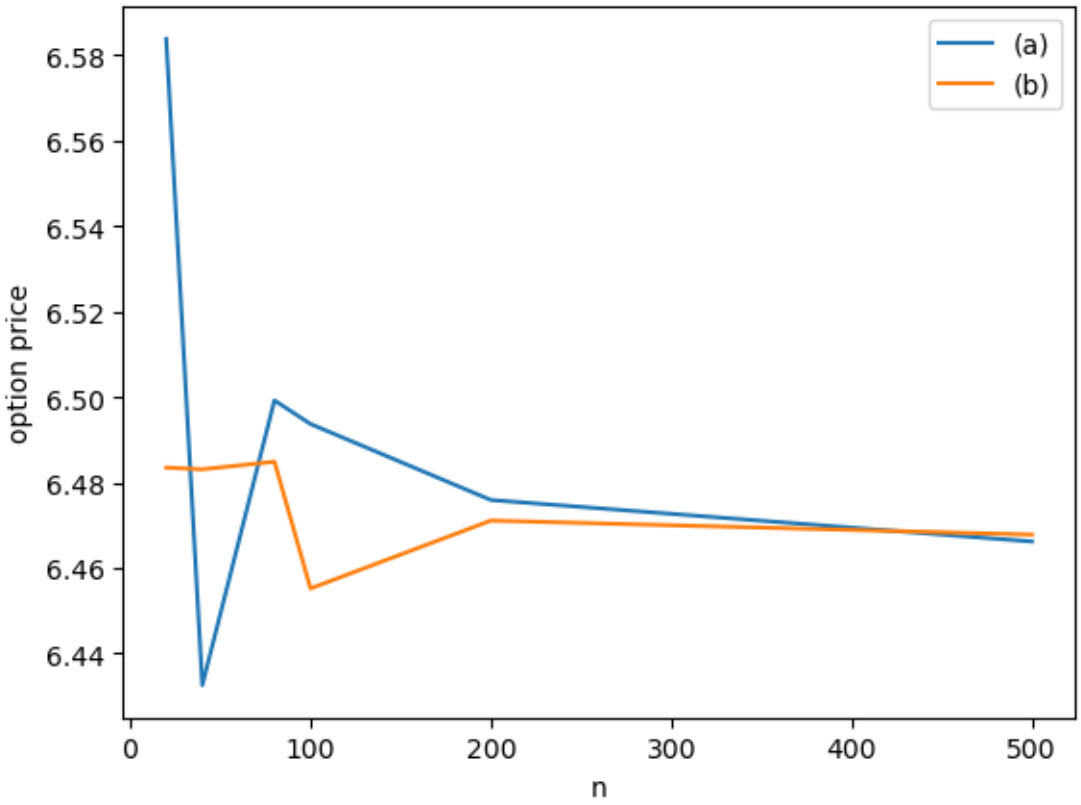
[ ]:
```
void binomial_1(int n, float T, float r, float sigma, float S0, float K)
{
    float S[n+1],C[n+1];
    float c = (exp(-r*T/n)+exp((r+pow(sigma,2))*T/n))/2;
    float d = c-sqrt(pow(c,2)-1), u = 1/d;
    float p = (exp(r*T/n)-d)/(u-d);
    for(int i = 0; i <= n; i++)
    {
        for(int j = 0; j <= n-i; j++)
        {
            S[j] = S0 * pow(u,n-i-j) * pow(d,j);
            if(i == 0)
                C[j] = max(K-S[j],0);
            else
                C[j] = max(max(K-S[j],0),exp(-r*T/n)*(p*C[j]+(1-p)*C[j+1]));
        }
    }
    cout << C[0];
}
```

[ ]:
```
void binomial_2(int n, float T, float r, float sigma, float S0, float K)
{
    float S[n+1],C[n+1];
    float u = exp((r-pow(sigma,2)/2)*T/n+sigma*sqrt(T/n)), d =␣
 ↪exp((r-pow(sigma,2)/2)*T/n-sigma*sqrt(T/n));
```

```
    float p = 0.5; //cannot write 1/2
    for(int i = 0; i <= n; i++)
    {
        for(int j = 0; j <= n-i; j++)
        {
            S[j] = S0 * pow(u,n-i-j) * pow(d,j);
            if(i == 0)
                C[j] = max(K-S[j],0);
            else
                C[j] = max(max(K-S[j],0),exp(-r*T/n)*(p*C[j]+(1-p)*C[j+1]));
        }
    }
    cout << C[0];
}
```

[27]:
```
import matplotlib.pyplot as plt
n_list = [20, 40, 80, 100, 200, 500]
plt.plot(n_list,[6.58381, 6.43258, 6.49921, 6.4937, 6.47591, 6.4662],␣
 ↪label='(a)')
plt.plot(n_list,[6.4835, 6.48309, 6.48488, 6.45521, 6.4711, 6.46781],␣
 ↪label='(b)')
plt.legend()
plt.xlabel('n')
plt.ylabel('option price')
```

[27]:
```
Text(0, 0.5, 'option price')
```



0.2 Question 2

Consider the following information on the stock of a company and American put options on it: $0 = \$180$, $= \$170$, $= 0.055$, $= 0.25$, $= 6$ months, $= 0.15$. Using the CRR Binomial tree method, estimate the following and draw their graphs: * (i) Delta of the put option as a function of S_0, for S_0 ranging from \$170 to \$190, in increments of \$2.

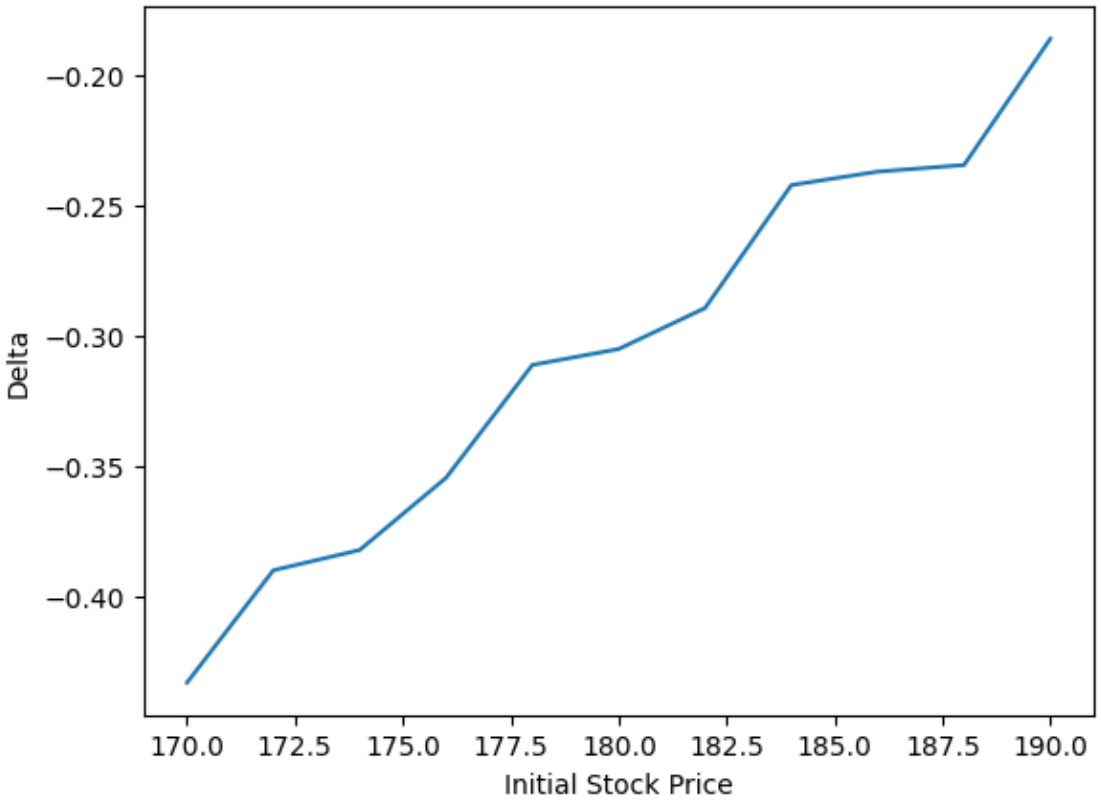
[ ]:
```
float CRR(float T, float r, float sigma, float S0, float K, int n = 100)
{
    # use this function to calculate the CRR binomial tree method put option␣
 ↪price
    float S[n+1],C[n+1];
    float u = exp(sigma*sqrt(T/n)), d = exp(-sigma*sqrt(T/n));
    float p = (exp(r*T/n)-d)/(u-d);
    for(int i = 0; i <= n; i++)
    {
        for(int j = 0; j <= n-i; j++)
        {
            S[j] = S0 * pow(u,n-i-j) * pow(d,j);
            if(i == 0)
                C[j] = max(K-S[j],0);
```

```
            else
                C[j] = max(max(K-S[j],0),exp(-r*T/n)*(p*C[j]+(1-p)*C[j+1]));
        }
    }
    return C[0];
}
```

[ ]:
```
void delta_S(float T, float r, float sigma, float S1, float S2, float K, float␣
 ↪increments)
{
    int N = (S2-S1)/increments;
    float delta[N];
    for(int i = 1; i <= N; i++)
    {
        delta[i-1] = (CRR(T,r,sigma,S1+increments*i,K) -␣
 ↪CRR(T,r,sigma,S1+increments*(i-1),K))/2;
        cout << delta[i-1] << ", ";
    }
    cout << endl;
}
```

[20]:
```
import numpy as np
S = np.arange(170,190+2,2)
delta = [-0.43296, -0.389886, -0.382119, -0.354444, -0.311186, -0.305076, -0.
 ↪28938, -0.242235, -0.2371, -0.234561, -0.186136
]
plt.plot(S, delta)
plt.xlabel('Initial Stock Price')
plt.ylabel('Delta')
```

[20]:
```
Text(0, 0.5, 'Delta')
```



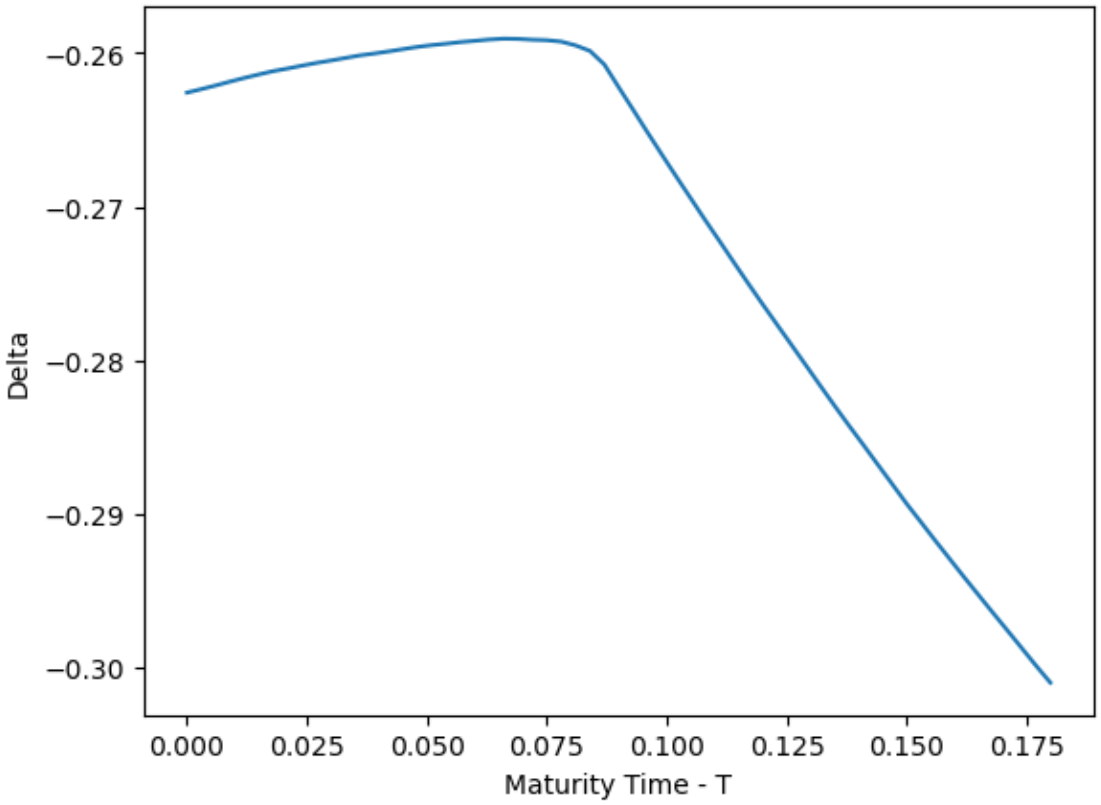
- (ii) Delta of the put option, as a function of T (time to expiration), T ranging from 0 to 0.18 in increments of 0.003.

[ ]:
```
void delta_T(float T, float r, float sigma, float S0, float K, float increments)
{
    int N = T/increments;
    float delta[N+1];
    for(int i = 0; i <= N; i++)
    {
        delta[i] = (CRR(T+increments*i,r,sigma,S0+1,K) -␣
 ↪CRR(T+increments*i,r,sigma,S0-1,K))/2;
        cout << delta[i] << ", ";
    }
    cout << endl;
}
```

[21]:
```
T = np.arange(0,0.18+0.003,0.003)
delta_T = [-0.262576, -0.262349, -0.26211, -0.261857, -0.261616, -0.26139, -0.
 ↪261175, -0.261005, -0.260818, -0.260641,
           -0.260486, -0.26032, -0.260153, -0.26003, -0.259892, -0.259746, -0.
 ↪2596, -0.259483, -0.259391, -0.259284,
```

```
          -0.259203, -0.259118, -0.259062, -0.259077, -0.25913, -0.259161, -0.
↪259252, -0.259498, -0.259858, -0.260742,
          -0.262225, -0.263687, -0.265164, -0.266612, -0.268044, -0.269453, -0.
↪270865, -0.272236, -0.273629, -0.275007,
          -0.276357, -0.277681, -0.279001, -0.280324, -0.281655, -0.282972, -0.
↪284261, -0.285518, -0.286799, -0.288065,
          -0.289345, -0.290546, -0.291758, -0.292945, -0.294132, -0.295293, -0.
↪296452, -0.297594, -0.298748, -0.2999,
          -0.301011]
plt.plot(T,delta_T)
plt.xlabel('Maturity Time - T')
plt.ylabel('Delta')
```

[21]: `Text(0, 0.5, 'Delta')`



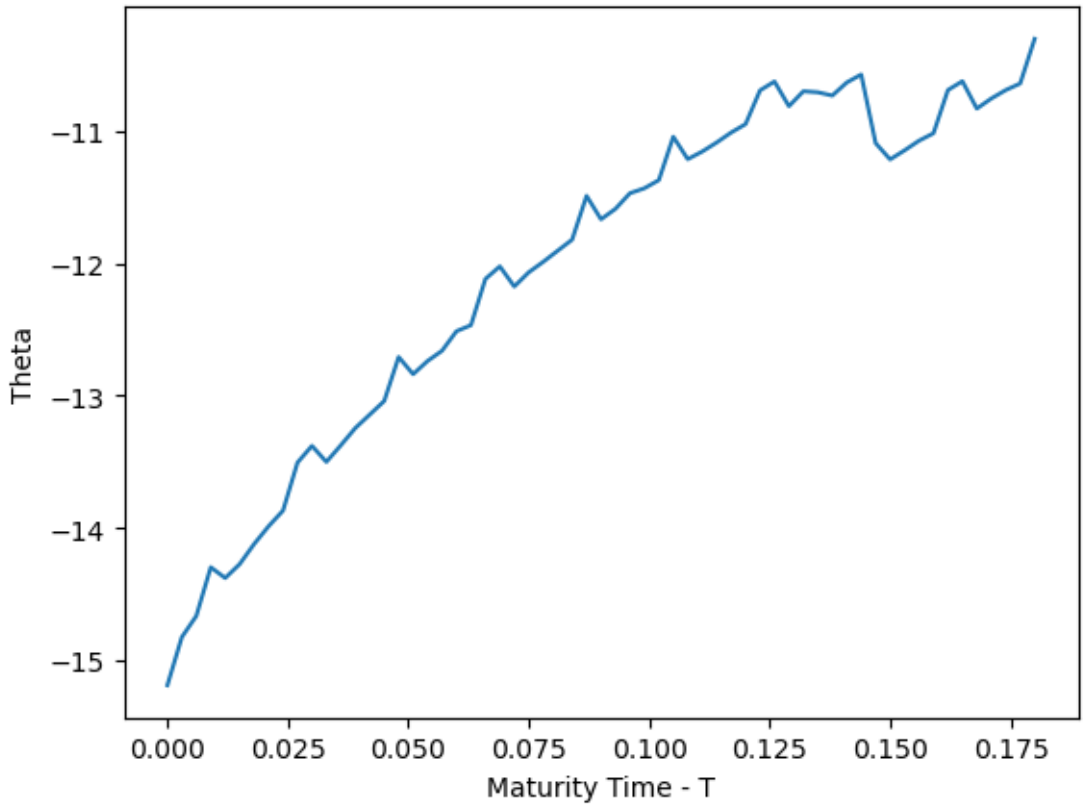
- (iii) Theta of the put option, as a function of T (time to expiration), T ranging from 0 to 0.18 in increments of 0.003.

[ ]:
```
void theta_T(float T, float r, float sigma, float S0, float K, float increments)
{
    int N = T/increments;
```

```
    float theta[N+1];
    for(int i = 0; i <= N; i++)
    {
        # if i == 0 we can only calculate theta on one side
        if(i == 0)
            theta[i] = (CRR(T+increments*i,r,sigma,S0,K) - CRR(T+increments*i+0.
 ↪001,r,sigma,S0,K))/0.001;
        else
            theta[i] = (CRR(T+increments*i-0.001,r,sigma,S0,K) -␣
 ↪CRR(T+increments*i+0.001,r,sigma,S0,K))/0.002;
        cout << theta[i] << ", ";
    }
    cout << endl;
}
```

[22]:
```
T = np.arange(0,0.18+0.003,0.003)
theta_T = [-15.1882, -14.8247, -14.665, -14.297, -14.3783, -14.273, -14.1213,␣
 ↪-13.9872, -13.8683, -13.5057, -13.3795,
           -13.5005, -13.3748, -13.2451, -13.1429, -13.0405, -12.7077, -12.
 ↪8378, -12.738, -12.6606, -12.5129, -12.4675,
           -12.118, -12.0229, -12.1751, -12.0678, -11.9886, -11.9047, -11.821,␣
 ↪-11.4901, -11.667, -11.5879, -11.4701,
           -11.4326, -11.3709, -11.0431, -11.2126, -11.1551, -11.0867, -11.
 ↪0114, -10.9477, -10.6928, -10.6256, -10.8125,
           -10.6993, -10.7071, -10.7319, -10.6332, -10.5729, -11.0915, -11.
 ↪2143, -11.1461, -11.0731, -11.0168, -10.6902,
           -10.6235, -10.8304, -10.7543, -10.6905, -10.6406, -10.3033]
plt.plot(T, theta_T)
plt.xlabel('Maturity Time - T')
plt.ylabel('Theta')
```

[22]:
```
Text(0, 0.5, 'Theta')
```



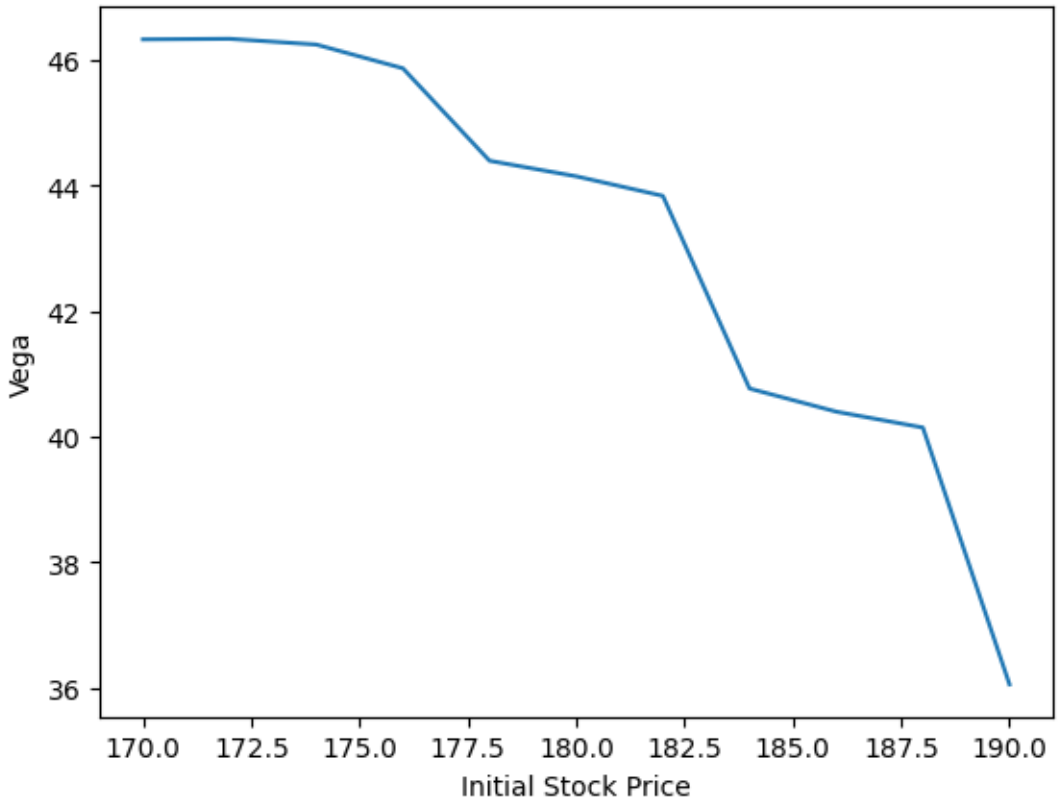
- (iv) Vega of the put option, as a function of S_0, for S_0 ranging from \$170 to \$190, in increments of \$2.

[ ]:
```
void vega_S(float T, float r, float sigma, float S1, float S2, float K, float␣
 ↪increments)
{
    int N = (S2-S1)/increments;
    float vega[N+1];
    for(int i = 0; i <= N; i++)
    {
        vega[i] = (CRR(T,r,sigma+0.01,S1+increments*i,K) - CRR(T,r,sigma-0.
 ↪01,S1+increments*i,K))/0.02;
        cout << vega[i] << ", ";
    }
    cout << endl;
}
```

[23]:
```
S = np.arange(170,190+2,2)
vega_S = [46.3335, 46.3428, 46.251, 45.8724, 44.3977, 44.1501, 43.8379, 40.
 ↪7657, 40.3971, 40.1438, 36.0485]
plt.plot(S, vega_S)
```

```
plt.xlabel('Initial Stock Price')
plt.ylabel('Vega')
```

[23]:
```
Text(0, 0.5, 'Vega')
```



0.3 Question 3

Compare the convergence rates of the two methods, (a) and (b), described below, by doing the following: Use the Trinomial-tree method to price a 6-month American put option with the following information: the risk-free interest rate is 5.5% per annum, the volatility is 25% per annum, the current stock price is \$180, and the strike price is \$170. Divide the time interval into equal parts to estimate the option price. Use n = 20, 40, 70, 80, 100, 200, 500; to estimate option prices and draw them all in one graph, where the horizontal axis measures , and the vertical one measures option price. The two methods are in (a) and (b) below

- (a) Use the Trinomial-tree method applied to the stock price-process ( ) in which = 1/ , = − √3 , = ((1−)+( )2+ 2)/(−)(1−) , = ((1−)+( )2+ 2)(−)(−1) , = 1 − −

[ ]:
```
void trinomial_1(int n, float T, float r, float sigma, float S0, float K)
{
    float S[2*n+1], P[2*n+1];
```

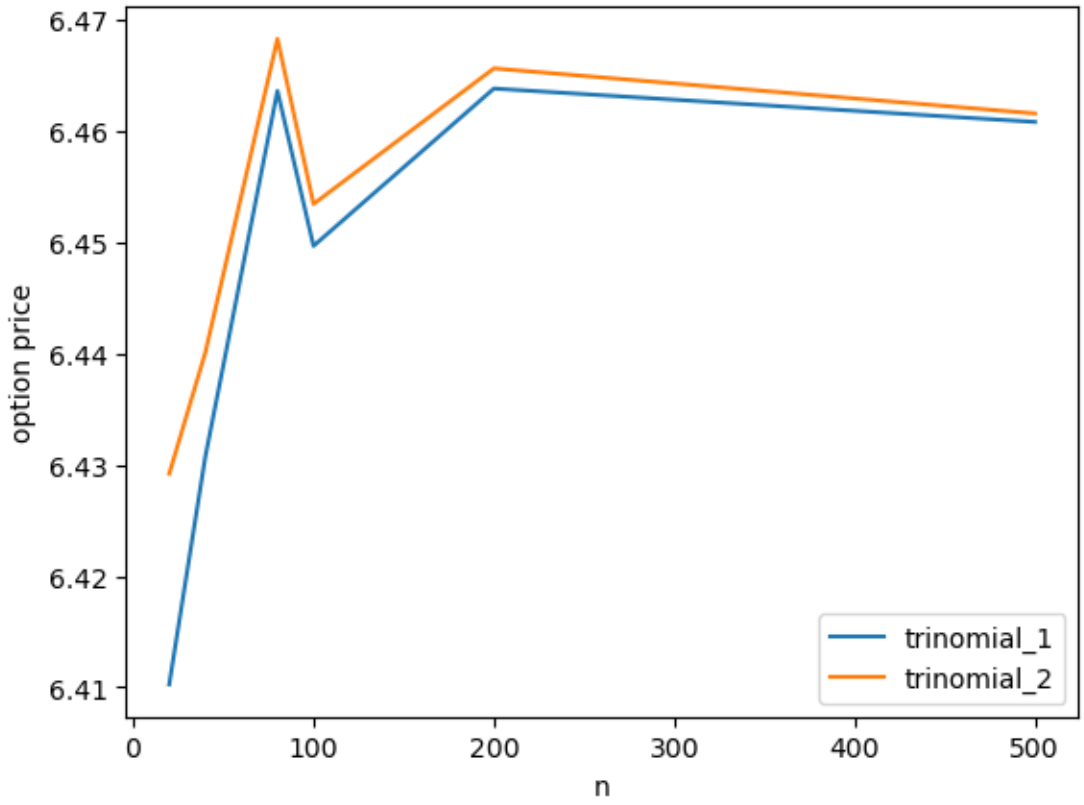
```
    float d = exp(-sigma*sqrt(3*T/n)), u = 1/d;
    float pd = (r*T/n*(1-u)+pow(r*T/n,2)+pow(sigma,2)*T/n)/(u-d)/(1-d);
    float pu = (r*T/n*(1-d)+pow(r*T/n,2)+pow(sigma,2)*T/n)/(u-d)/(u-1), pm =␣
 ↪1-pu-pd;
    for(int i = 0; i <= n; i++)
    {
        for(int j = 0; j <= 2*(n-i); j++)
        {
            S[j] = S0 * pow(u,n-i-j);
            if(i == 0)
                P[j] = max(K-S[j],0);
            else
                P[j] = max(max(K-S[j],0),exp(-r*T/
 ↪n)*(pu*P[j]+pm*P[j+1]+pd*P[j+2]));
        }
    }
    cout << P[0] << ", ";
}
```

- (b) Use the Trinomial-tree method applied to the Log-stock price-process ( ) in which = √3 , = − √3 , = 1/2 ((2 +(− 2/2)² 2)/ 2 − (− 2/2) /), = ((2 +(− 2/2)² 2)/ 2 + (− 2/2) /), = 1 − −

[ ]:
```
void trinomial_2(int n, float T, float r, float sigma, float S0, float K)
{
    float X[2*n+1], P[2*n+1];
    float u = sigma*sqrt(3*T/n);
    float pd = ((pow(sigma,2)*T/n+pow(r-pow(sigma,2)/2,2)*pow(T/n,2))/
 ↪pow(u,2)-(r-pow(sigma,2)/2)*T/n/u)/2;
    float pu = ((pow(sigma,2)*T/n+pow(r-pow(sigma,2)/2,2)*pow(T/n,2))/
 ↪pow(u,2)+(r-pow(sigma,2)/2)*T/n/u)/2;
    float pm = 1-pu-pd;
    for(int i = 0; i <= n; i++)
    {
        for(int j = 0; j <= 2*(n-i); j++)
        {
            X[j] = log(S0) + (n-i-j) * u;
            if(i == 0)
                P[j] = max(K-exp(X[j]),0);
            else
                P[j] = max(max(K-exp(X[j]),0),exp(-r*T/
 ↪n)*(pu*P[j]+pm*P[j+1]+pd*P[j+2]));
        }
    }
    cout << P[0] << ", ";
}
```

[26]:
```
plt.plot(n_list, [6.41027, 6.43071, 6.46359, 6.44966, 6.4638, 6.4608], label =␣
↪'trinomial_1')
plt.plot(n_list, [6.42919, 6.44007, 6.46827, 6.45343, 6.46561, 6.46156], label␣
↪= 'trinomial_2')
plt.legend()
plt.xlabel('n')
plt.ylabel('option price')
```

[26]:
```
Text(0, 0.5, 'option price')
```



0.4 Q4

Consider the following information on the stock of company XYZ: The current stock price is \$180, and the volatility of the stock price is = 25% per annum. Assume the prevailing risk-free rate is = 5.5% per annum. Use the following method to price the specified option:

- (a) Use the LSMC method with N=100,000 paths simulations (50,000 plus 50,000 antithetic variates) and a time step of = 1/√ to price an American Put option with strike price of = \$170 and maturity of 0.5-years and 1.5-years. Use the first of the Laguerre Polynomials for = 2, 3, 4, 5. (That is, you will compute 8 prices here). Compare the prices for the 4 cases, = 2, 3, 4, 5 and comment on the choice of k.

- (b) Use the LSMC method with N=100,000 paths simulations (50,000 plus 50,000 antithetic variates) and a time step of $= 1/\sqrt{}$ to price an American Put option with strike price of $= \$170$ and maturity of 0.5-years and 1.5-years. Use the first $$ of the Hermite Polynomials for $= 2, 3, 4, 5$. (That is, you will compute 8 prices here). Compare the prices for the 4 cases, $= 2, 3, 4, 5$ and comment on the choice of k.
- (c) Use the LSMC method with N=100,000 paths simulations (50,000 plus 50,000 antithetic variates) and a time step of $= 1/\sqrt{}$ to price an American Put option with strike price of $= \$170$ and maturity of 0.5-years and 1.5-years. Use the first $$ of the Simple Monomials for $= 2, 3, 4, 5$. (That is, you will compute 8 prices here). Compare the prices for the 4 cases, $= 2, 3, 4, 5$ and comment on the choice of k.

[ ]:
```
SparseMat simulate_path(vector<vector<double>>& S, float S0, int N, int n,␣
 ↪float r, float sigma, float dt, float K)
{
    # antithetic variates
    float w;
    for(int i = 0; i < N/2; i++){
        S[0][2*i] = S0;
        S[0][2*i+1] = S0;
        for(int j = 1; j < n; j++){
            w = sqrt(dt) * dis(gen);
            S[j][2*i] = S[j-1][2*i] + r*S[j-1][2*i]*dt + sigma*S[j-1][2*i]*w;
            S[j][2*i+1] = S[j-1][2*i+1] + r*S[j-1][2*i+1]*dt -␣
 ↪sigma*S[j-1][2*i+1]*w;
        }
    }
    SparseMat EV(N,n);
    EV.setZero();
    for(int i = 0; i < N; i++){
        for(int j = 0; j < n; j++){
            if(K-S[j][i]>0){EV.coeffRef(i,j) = K-S[j][i];}
        }
    }
    return EV;
}
```

[ ]:
```
float LSMC(float T, float r, float sigma, double S0, double K, int k,␣
 ↪double(*base_func)(int,double), int N = 10000)
{
    # Compute LSMC for American options using regression
    float dt = 1/sqrt(N);
    int n = int(T/dt)+1; // calculate how many timestamp

    # use Sparse Matrix to save space
    SparseMat Index(n, N);
    SparseMat P(n, N);
    MatrixXd A(k, k);
```

```
    # initiate as 0
    Index.setZero();
    P.setZero();
    A.setZero();
    vector<vector<double>> S(n, vector<double>(N, 0.0));
    VectorXd b = VectorXd::Zero(k), a = VectorXd::Zero(k);

    # 1. simulate N paths
    SparseMat EV(N, n);
    EV = simulate_path(S, S0, N, n, r, sigma, dt, K);
    EV.makeCompressed();

    # set the initial index
    int count = 0;
    for (SparseMatrix<double>::InnerIterator it(EV, n-1); it; ++it) {
        if(EV.coeffRef(it.row(),n-1)<=0){cout << "error!!!" << endl;}
        Index.coeffRef(n-1,it.row()) = 1;
    }

    # 2. Calculate continuation values and make exercise decisions
    int j = 0;
    for(int i = n-1; i >= 1; i--)
    {
        A.setZero();
        b.setZero();
        # calculate parameter a s.t. A'a=b
        for (SparseMatrix<double>::InnerIterator it(EV, i-1); it; ++it) {
            j = int(it.row());
            for(int m = 0; m < k; m++){
                for(int q = 0; q < k; q++){A(m,q) += base_func(m,S[i-1][j]/K) *␣
↪base_func(q,S[i-1][j]/K);}
                for(int q = i; q < n; q++){b(m) += Index.
↪coeffRef(q,j)*exp(-r*(q-i+1)*dt)*max(1-S[q][j]/K,0) * base_func(m,S[i-1][j]/
↪K);}
            }
        }
        a = A.lu().solve(b);

        # compare EV and CV
        for (SparseMatrix<double>::InnerIterator it(EV, i-1); it; ++it) {
            j = int(it.row());
            float CV = 0;
            for(int m = 0; m < k; m++){CV += a[m]*base_func(m,S[i-1][j]/K);}
            if(EV.coeffRef(j,i-1)/K > CV){
                Index.coeffRef(i-1,j) = 1;
                for(int q = i; q < n; q++){Index.coeffRef(q,j) = 0;}
                P.coeffRef(i-1,j) = exp(-r*(i-1)*dt) * EV.coeffRef(j,i-1);
```

```
                count += 1;
            }
        }
    }

    # Last Calculate and return the average payoff
    float mean = 0;
    Index.makeCompressed();
    for (int k = 0; k < Index.outerSize(); ++k) {
        for (SparseMatrix<double>::InnerIterator it(Index, k); it; ++it) {
            j = int(it.row());
            if(Index.coeffRef(j,k) == 1)
            {
                mean += P.coeffRef(j,k);
            }
        }
    }
    return mean/N;
}
```

[ ]:
```
# a
float Laguerre(int k, float x)
{
    # Compute the Laguerre polynomial value for a given x based on order k
    switch(k)
    {
        case 0: return exp(-x/2);
        case 1: return exp(-x/2)*(1-x);
        case 2: return exp(-x/2)*(1-2*x+pow(x,2)/2);
        case 3: return exp(-x/2)*(1-3*x+3*pow(x,2)/2-pow(x,3)/6);
        case 4: return exp(-x/2)*(1-4*x+3*pow(x,2)-2*pow(x,3)/3+pow(x,4)/24);
    }
    return 0;
}
```

[ ]:
```
float Hermite(int k, float x)
{
    # Compute the Hermite polynomial value for a given x based on order k
    switch(k)
    {
        case 0: return 1;
        case 1: return 2*x;
        case 2: return 4*pow(x,2)-2;
        case 3: return 8*pow(x,3)-12*x;
        case 4: return 16*pow(x,4)-56*pow(x,2)+16;
    }
    return 0;
```

```
}
```

[ ]:
```
float Monomials(int k, float x)
{
    # Compute the Monomials polynomial value for a given x based on order k
    switch(k)
    {
        case 0: return 1;
        case 1: return x;
        case 2: return pow(x,2);
        case 3: return pow(x,3);
        case 4: return pow(x,4);
    }
    return 0;
}
```

[ ]:
```
# in the int main() part:
    cout << "4(a): ";
    for(float T = 0.5; T < 2; T+=1)
    {
        for(int n = 2; n <= 5; n++)
        {
            cout << T  << ", " << n << ": " << LSMC(T, r, sigma, S0, K, n,␣
 ↪Laguerre)<<endl;
        }
    }
    cout << "4(b): ";
    for(float T = 0.5; T < 2; T+=1)
    {
        for(int n = 2; n <= 5; n++)
        {
            cout << T  << ", " << n << ": " << LSMC(T, r, sigma, S0, K, n,␣
 ↪Hermite)<<endl;
        }
    }
    cout << "4(c): ";
    for(float T = 0.5; T < 2; T+=1)
    {
        for(int n = 2; n <= 5; n++)
        {
            cout << T  << ", " << n << ": " << LSMC(T, r, sigma, S0, K, n,␣
 ↪Monomials)<<endl;
        }
    }
```

0.4.1 OUTPUT_6

T=0.5/k	Laguerre	Hermite	Monomials
2	5.9053	4.77032	4.76563
3	5.55796	5.71456	5.64286
4	5.91483	5.78762	6.03062
5	6.01208	6.16137	6.24969

T=1.5/k	Laguerre	Hermite	Monomials
2	11.4652	10.0108	9.1008
3	11.1773	11.159	11.0475
4	11.5082	11.6783	11.6055
5	11.7865	11.6865	11.909

Analysis of Data:

At T = 0.5 years:

- Laguerre Polynomials: Prices generally increase as the degree of the polynomial increases from k=2 to k=5. Laguerre polynomials appear to provide the highest option pricing across all k values compared to Hermite and Monomials.
- Hermite Polynomials: Starting with the lowest price at k=2, Hermite prices increase as the polynomial degree increases, showing higher sensitivity to changes in polynomial degree.
- Monomials: These also show an increase in price with the degree, but generally provide the lowest pricing across similar degrees when compared to the other two methods.

At T = 1.5 years:

- Laguerre Polynomials: Consistently provide higher prices as k increases, following a similar trend to the T = 0.5-year scenario, suggesting stable performance across different maturities.
- Hermite Polynomials: Prices again increase with the degree but are more competitive with Laguerre at this maturity. Notably, Hermite at k=5 almost aligns with Laguerre, suggesting improved reliability or fit at higher polynomial degrees for longer maturities.
- Monomials: Show a steady increase with the polynomial degree and actually exceed the pricing given by Hermite and Laguerre at k=5, suggesting that simpler polynomial bases might eventually converge to a more "true" value, or reflect certain dynamics better at higher degrees and longer maturities.

Comments: * Impact of Polynomial Type: Different polynomial bases affect the LSMC method's efficiency and accuracy. Laguerre and Hermite, being orthogonal polynomials, might be capturing the dynamics of the option's value better than simple monomials, especially at lower degrees or shorter maturities.

- Degree Sensitivity: Increasing the polynomial degree generally results in higher option prices, likely due to a better fit of the continuation value approximation. This is more evident in monomials, where higher degrees sharply increase the estimated option prices.
- Maturity Influence: Longer maturities show an interesting interaction with polynomial type and degree. For instance, Monomials, despite being simple, provide competitive or even higher pricing at k=5 for T = 1.5 years, possibly indicating that for longer maturities, the fitting

of the model becomes critical and simpler models might sometimes capture the long-term dynamics adequately.

- Choosing Polynomial Degree: The choice of k needs careful consideration. Higher degrees might offer better accuracy but at the cost of computational complexity and potential overfitting. The trade-off between accuracy and computational efficiency must be managed, especially in real-world applications.
- Practical Implications: For practical implementations, choosing between these polynomial bases would depend on the specific characteristics of the option being priced, including its maturity, underlying asset dynamics, and market conditions. Extensive testing and validation are advised to select the most suitable method and degree.

We also need to pay attension to the paper that mentions when using Laguerre Polynomials, which mentions that directly applying the weighted Laguerre polynomials to the problem could result in computational underflows and we need to renormalized the American put example by dividing all cash flows and prices by the strike price, and estimating the conditional expectation function in the renormalized space;

0.5 Question 5

Consider the following information on the stock of company XYZ: The volatility of the stock price is = 25% per annum. Assume the prevailing risk-free rate is = 5.5% per annum. Use the = ( ) transformation of the Black-Scholes PDE, and = 0.002, with = √ ; then with = √3 ; then with = √4 , and a uniform grid (on X) to price an American Put option with strike price of = $170, expiration of 6 months and current stock prices ranging from $170 to $190; using the specified methods below:

- (a) Explicit Finite-Difference method,
- (b) Implicit Finite-Difference method,
- (c) Crank-Nicolson Finite-Difference method.

[ ]:
```
MatrixXd EFD(float dt, float sigma, float r, float dx, float S1, float S2,␣
 ↪float K, float T)
{
    int n = T/dt, N = int(log(S2/S1)/dx+1);
    float Pu = dt/2*(pow(sigma/dx,2)+(r-pow(sigma,2)/2)/dx);
    float Pd = dt/2*(pow(sigma/dx,2)-(r-pow(sigma,2)/2)/dx);
    float Pm = 1-dt*pow(sigma/dx,2)-r*dt;
    MatrixXd A(N+1,N+1);
    VectorXd P(N+1), B(N+1);

    # initiate A
    VectorXd p(3);
    p << Pu,Pm,Pd;
    A.row(0).segment(0,3) = p;
    A.row(N).segment(N-2,3) = p;
    for(int i = 1; i <= N-1; i++){A.row(i).segment(i-1,3) = p;}
```

```
    # initiate the boudary value
    for(int i = 0; i <= N; i++){P(i) = max(K - S1*exp((N-i)*dx),0);}
    //printAsPythonList(P);
    # iterate to get Put price(t=0)
    B(N) = - S1*(1-exp(dx));
    for(int i = n-1; i >= 0; i--)
    {
        P = A * P + B;
        # American condition
        for(int j = 0; j <= N; j++){P(j) = max(P(j), max(K -␣
 ↪S1*exp((N-j)*dx),0));}
    }
    return P;
}
```

[ ]:
```
MatrixXd IFD(float dt, float sigma, float r, float dx, float S1, float S2,␣
 ↪float K, float T)
{
    int n = T/dt, N = int(log(S2/S1)/dx+1);
    float Pu = -dt/2*(pow(sigma/dx,2)+(r-pow(sigma,2)/2)/dx);
    float Pd = -dt/2*(pow(sigma/dx,2)-(r-pow(sigma,2)/2)/dx);
    float Pm = 1+dt*pow(sigma/dx,2)+r*dt;
    MatrixXd A(N+1,N+1);
    VectorXd P(N+1), B(N+1);

    # initiate A
    VectorXd p(3);
    p << Pu,Pm,Pd;
    A(0,0) = 1;
    A(0,1) = -1;
    A(N,N-1) = 1;
    A(N,N) = -1;
    for(int i = 1; i <= N-1; i++){A.row(i).segment(i-1,3) = p;}

    # initiate the boudary value
    for(int i = 0; i <= N; i++){P(i) = max(K - S1*exp((N-i)*dx),0);}

    printAsPythonList(P.col(1));
    # iterate to get Put price(t=0)
    B(N) = - S1*(exp(dx)-1);
    for(int i = n-1; i >= 0; i--)
    {
        B.segment(1,N-1) = P.segment(1,N-1);
        P = A.colPivHouseholderQr().solve(B);
        # American condition
        for(int j = 0; j <= N; j++){P(j) = max(P(j), K - S1*exp((N-j)*dx));}
    }
```

```
    return P;
}
```

[ ]:
```
MatrixXd CN_FD(float dt, float sigma, float r, float dx, float S1, float S2,␣
 ↪float K, float T)
{
    int n = T/dt, N = int(log(S2/S1)/dx+1);
    float Pu = -dt*(pow(sigma/dx,2)+(r-pow(sigma,2)/2)/dx)/4;
    float Pd = -dt*(pow(sigma/dx,2)-(r-pow(sigma,2)/2)/dx)/4;
    float Pm = 1+dt*pow(sigma/dx,2)/2+r*dt/2;
    MatrixXd A(N+1,N+1);
    VectorXd P(N+1), B(N+1);

    # initiate A
    VectorXd p(3);
    p << Pu,Pm,Pd;
    A(0,0) = 1;
    A(0,1) = -1;
    A(N,N-1) = 1;
    A(N,N) = -1;
    for(int i = 1; i <= N-1; i++){A.row(i).segment(i-1,3) = p;}

    # initiate the boudary value
    for(int i = 0; i <= N; i++){P(i) = max(K - S1*exp((N-i)*dx),0);}

    # iterate to get Put price(t=0)
    B(N) = - S1*(exp(dx)-1);
    for(int i = n-1; i >= 0; i--)
    {
        B.segment(1,N-1) = - Pu * P.segment(0,N-1);
        B.segment(1,N-1) += - (Pm-2) * P.segment(1,N-1);
        B.segment(1,N-1) += - Pd * P.segment(2,N-1);
        P = A.colPivHouseholderQr().solve(B);
        # American condition
        for(int j = 0; j <= N; j++){P(j) = max(P(j), K - S1*exp((N-j)*dx));}
    }
    return P;
}
```

0.5.1 OUTPUT_5

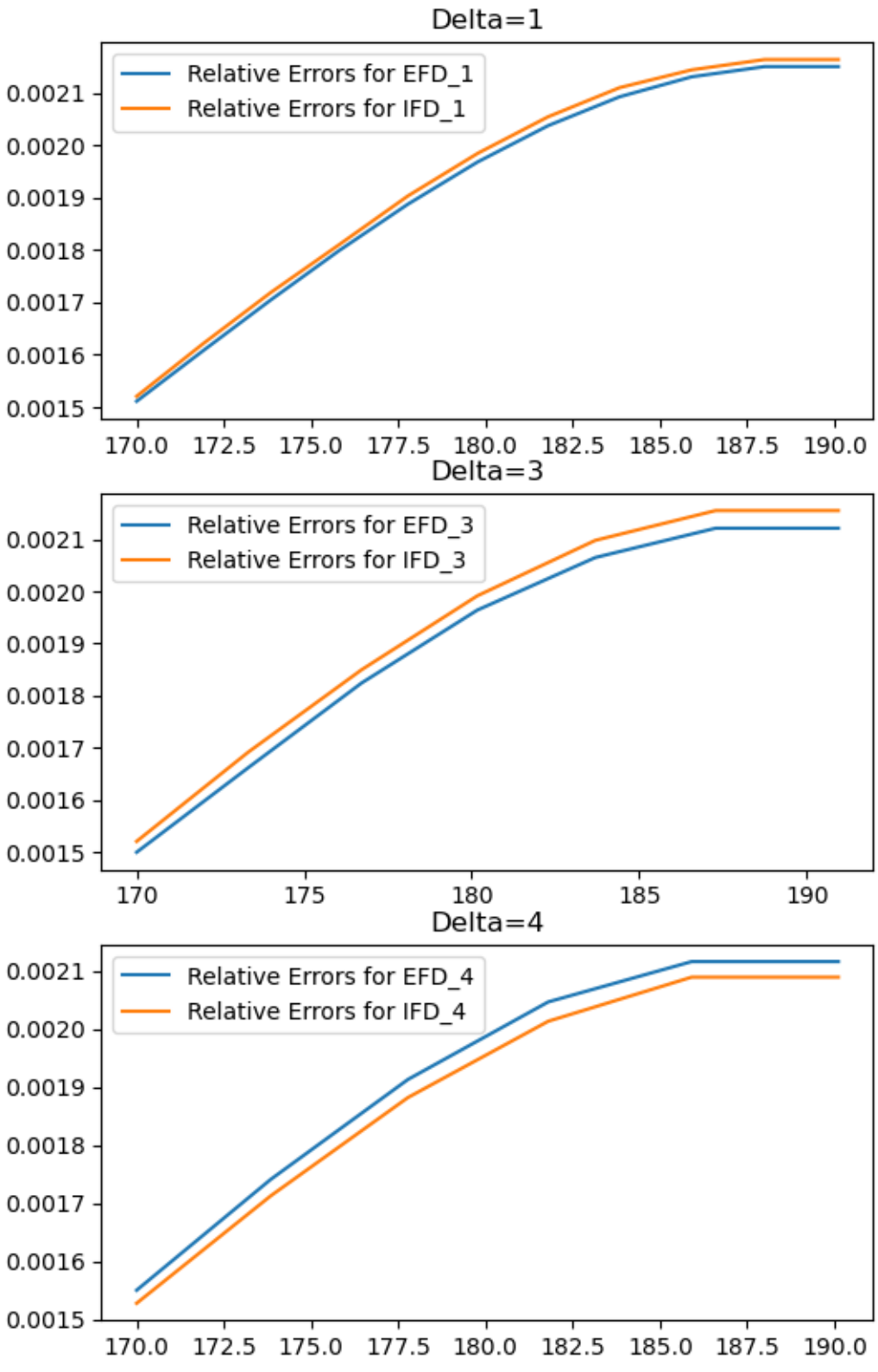
[34]:
```
# EFD
EFD_1 = [22.2798, 22.2798, 22.4844, 22.8954, 23.5147, 24.3442, 25.3858, 26.
 ↪6417, 28.1138, 29.8044, 31.7158]
EFD_3 = [23.5664, 23.5664, 24.2101, 25.5074, 27.469, 30.106, 33.4301]
EFD_4 = [26.0848, 26.0848, 27.0194, 28.9055, 31.761, 35.6052]
# IFD
```

```
IFD_1 = [22.3761, 22.3761, 22.5807, 22.9918, 23.6111, 24.4406, 25.4822, 26.738,␣
 ↪28.2102, 29.9008, 31.8121]
IFD_3 = [23.6674, 23.6674, 24.3111, 25.6085, 27.5701, 30.2071, 33.5312]
IFD_4 = [26.1947, 26.1947, 27.1293, 29.0154, 31.8709, 35.715]
#CN_FD
CN_FD_1 = [22.3278, 22.3278, 22.5324, 22.9434, 23.5627, 24.3922, 25.4338, 26.
 ↪6897, 28.1618, 29.8524, 31.7638]
CN_FD_3 = [23.6165, 23.6165, 24.2602, 25.5576, 27.5192, 30.1561, 33.4803]
CN_FD_4 = [26.1401, 26.1401, 27.0748, 28.9609, 31.8164, 35.6605]
```

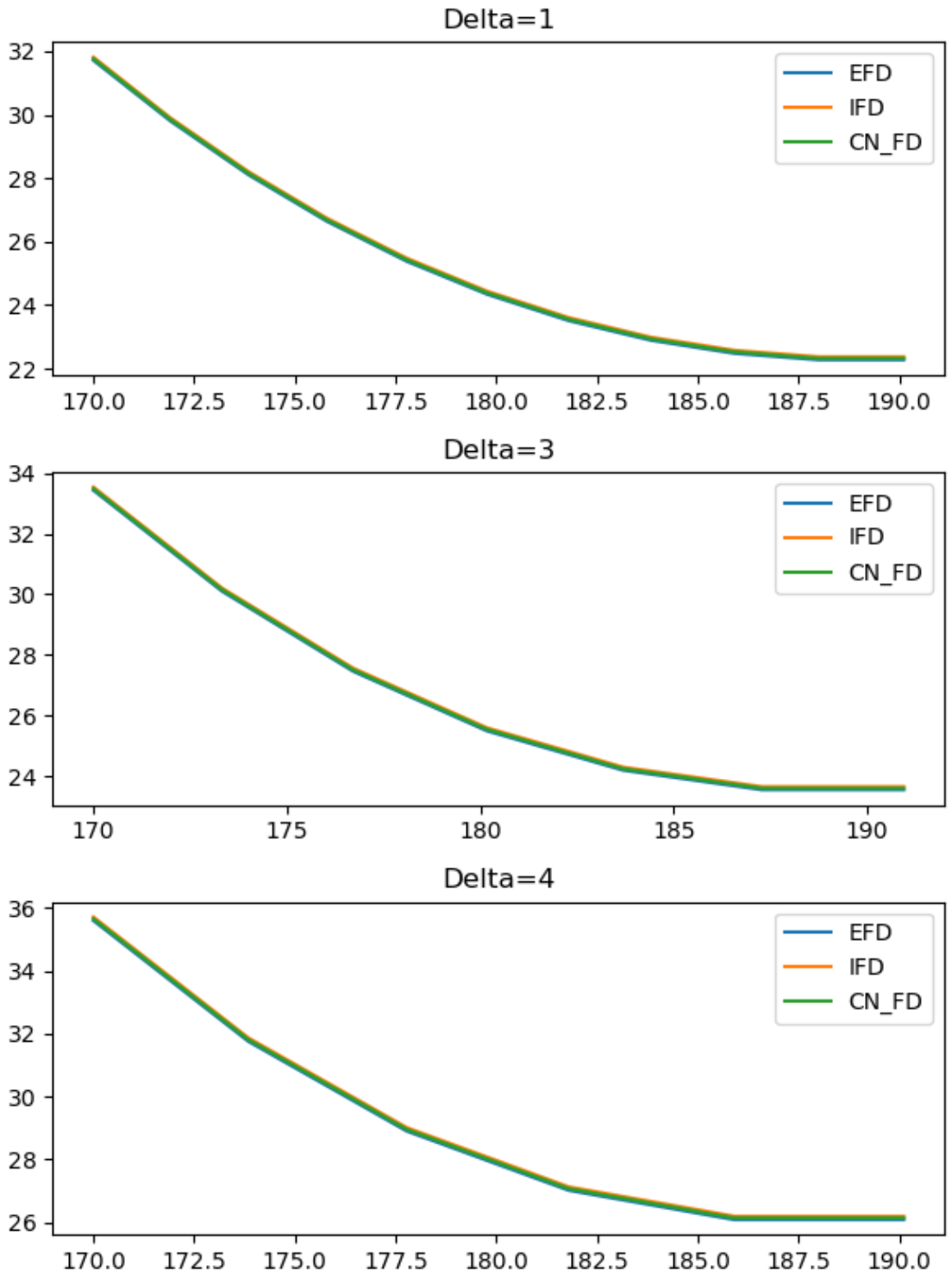
[38]:
```
from math import *
# Calculate x values for each series
x_1 = [170 * exp(0.25 * np.sqrt(0.002) * i)for i in np.
 ↪arange(len(EFD_1)-1,-1,-1)]
x_3 = [170 * exp(0.25 * np.sqrt(3*0.002) * i)for i in np.
 ↪arange(len(EFD_3)-1,-1,-1)]
x_4 = [170 * exp(0.25 * np.sqrt(4*0.002) * i)for i in np.
 ↪arange(len(EFD_4)-1,-1,-1)]

# Calculate relative errors
def calculate_relative_error(method_values, reference_values):
    return np.abs((method_values - reference_values) / reference_values)

# Plotting
fig, axes = plt.subplots(3, 1, figsize=(6, 10))

# Plot for EFD, IFD, CN_FD with i=1
axes[0].plot(x_1, calculate_relative_error(np.array(EFD_1), np.array(CN_FD_1)),␣
 ↪label='Relative Errors for EFD_1')
axes[0].plot(x_1, calculate_relative_error(np.array(IFD_1), np.array(CN_FD_1)),␣
 ↪label='Relative Errors for IFD_1')
axes[0].set_title('Delta=1')
axes[0].legend()

# Plot for EFD, IFD, CN_FD with i=3
axes[1].plot(x_3, calculate_relative_error(np.array(EFD_3), np.array(CN_FD_3)),␣
 ↪label='Relative Errors for EFD_3')
axes[1].plot(x_3, calculate_relative_error(np.array(IFD_3), np.array(CN_FD_3)),␣
 ↪label='Relative Errors for IFD_3')
axes[1].set_title('Delta=3')
axes[1].legend()

# Plot for EFD, IFD, CN_FD with i=4
axes[2].plot(x_4, calculate_relative_error(np.array(EFD_4), np.array(CN_FD_4)),␣
 ↪label='Relative Errors for EFD_4')
```

```
axes[2].plot(x_4, calculate_relative_error(np.array(IFD_4), np.array(CN_FD_4)),␣
 ↪label='Relative Errors for IFD_4')
axes[2].set_title('Delta=4')
axes[2].legend()
```

[38]: <matplotlib.legend.Legend at 0x10d76d150>


Delta=1
Relative Errors for EFD_1
Relative Errors for IFD_1
0.0021
0.0020
0.0019
0.0018
0.0017
0.0016
0.0015
170.0
172.5
175.0
177.5
180.0
182.5
185.0
187.5
190.0
Delta=3
Relative Errors for EFD_3
Relative Errors for IFD_3
0.0021
0.0020
0.0019
0.0018
0.0017
0.0016
0.0015
170
175
180
185
190
Delta=4
Relative Errors for EFD_4
Relative Errors for IFD_4
0.0021
0.0020
0.0019
0.0018
0.0017
0.0016
0.0015
170.0
172.5
175.0
177.5
180.0
182.5
185.0
187.5
190.0

[54]:
```
from math import *
# Calculate x values for each series
x_1 = [170 * exp(0.25 * np.sqrt(0.002) * i)for i in np.
 ↪arange(len(EFD_1)-1,-1,-1)]
x_3 = [170 * exp(0.25 * np.sqrt(3*0.002) * i)for i in np.
 ↪arange(len(EFD_3)-1,-1,-1)]
x_4 = [170 * exp(0.25 * np.sqrt(4*0.002) * i)for i in np.
 ↪arange(len(EFD_4)-1,-1,-1)]

# Plotting
fig, axes = plt.subplots(3, 1, figsize=(6, 8))

# Plot for EFD, IFD, CN_FD with i=1
axes[0].plot(x_1, EFD_1, label='EFD')
axes[0].plot(x_1, IFD_1, label='IFD')
axes[0].plot(x_1, CN_FD_1, label='CN_FD')
axes[0].set_title('Delta=1')
axes[0].legend()

# Plot for EFD, IFD, CN_FD with i=3
axes[1].plot(x_3, EFD_3, label='EFD')
axes[1].plot(x_3, IFD_3, label='IFD')
axes[1].plot(x_3, CN_FD_3, label='CN_FD')
axes[1].set_title('Delta=3')
axes[1].legend()

# Plot for EFD, IFD, CN_FD with i=4
axes[2].plot(x_4, EFD_4, label='EFD')
axes[2].plot(x_4, IFD_4, label='IFD')
axes[2].plot(x_4, CN_FD_4, label='CN_FD')
axes[2].set_title('Delta=4')
axes[2].legend()

# Adjust plot spacing
plt.tight_layout()

# Show plot
plt.show()
```



Analysis of the Option Prices Graphs: * Trend Observation: Across all graphs (Delta=1, Delta=3, Delta=4), it is observed that the option prices decrease as the underlying asset price increases. This trend is typical in put option pricing, where the option's value generally decreases

as the stock price moves further from the strike price in a favorable direction. * Method Comparison: The CN_FD method consistently results in the highest option prices, followed by IFD and then EFD. This ordering could suggest that the CN_FD method is the most conservative in estimating the price declines, possibly due to its better stability and convergence properties (balancing between EFD and IFD).

Analysis of the Relative Errors Graphs: * Error Patterns: All methods exhibit increasing relative errors as the underlying asset price increases. This could be indicative of growing inaccuracies in the numerical methods at higher asset price levels, where the intrinsic value of the put options becomes very low, and the relative error hence becomes more sensitive to small absolute errors. * Comparison of Methods: In general, the relative error lines for EFD and IFD are very close to each other across all scenarios, suggesting similar error behaviors between these methods. However, EFD seems to have slightly higher relative errors compared to IFD, which might be due to the less stable nature of explicit methods in certain settings.

Key Takeaways: * Price Sensitivity: The decrease in option prices with increasing asset prices highlights the inherent nature of put options, which gain in value as the market declines. The steepness of these curves could be analyzed further to understand the sensitivity (Delta) of the option prices to changes in the underlying asset's price.

- Numerical Method Efficiency: The Crank-Nicolson method, showing the highest option prices and possibly lower relative errors (not fully discernible from the provided data), might be considered the most reliable for balancing accuracy and numerical stability in pricing.
- Relative Error Implications: The increase in relative errors with the asset price suggests that all numerical methods might face challenges in accurately pricing deep out-of-the-money options where the absolute prices are very low. This underscores the importance of method selection based on the option's moneyness and the underlying asset's volatility.
- Practical Considerations: For real-world applications, choosing between these methods would depend on factors such as computational resources, required accuracy, and the specific characteristics of the financial instrument being modeled. The CN_FD method, despite potentially higher computational overhead due to its implicit nature, might offer a better balance between accuracy and stability.

0.6 Question 6

Consider the following information on the stock of company XYZ: The volatility of the stock price is = 25% per annum. Assume the prevailing risk-free rate is = 5.5% per annum. Use the Black-Scholes PDE (for S) to price American Put options with strike prices of = \$170, expiration of 6 months and current stock prices for a range from \$170 to \$190; using the specified methods below: Choose = 0.002, with = 0.5, or with = 1.

- (a) Explicit Finite-Difference method,
- (b) Implicit Finite-Difference method,
- (c) Crank-Nicolson Finite-Difference method.

[ ]:
```
# follow the second version of notes and TA notes
MatrixXd EFD_S(float dt, float sigma, float r, float dS, float S1, float S2,␣
 ↪float K, float T)
```

```
{
    int n = T/dt, N = int((S2-S1)/dS/2);
    float S0 = (S1+S2)/2;
    float Pu, Pm, Pd;
    MatrixXd Pumd(2*N+1,3);
    VectorXd P(2*N+1), B(2*N+1);

    # initiate A
    MatrixXd A = MatrixXd::Identity(2*N+1, 2*N+1);
    A(0,1) = -1;
    A(2*N,2*N-1) = 1;
    A(2*N,2*N) = -1;

    # initiate Pumd
    for(int i = N-1; i >= -N+1; i--)
    {
        Pumd.row(N-i) << dt/2*(r*(S0+i*dS)/dS+pow(sigma,2)*pow((S0+i*dS)/
 ↪dS,2)),1-dt*(pow(sigma*(S0+i*dS)/dS,2)+r),  dt/2*(-r*(S0+i*dS)/
 ↪dS+pow(sigma,2)*pow((S0+i*dS)/dS,2));
    }

    # initiate the boudary value
    for(int i = 0; i <= 2*N; i++){P(i) = max(K - (S0+(N-i)*dS),0);}


    # iterate to get Put price(t=0)
    B(2*N) = - dS;
    for(int i = n-1; i >= 0; i--)
    {
        B.segment(1,2*N-1) = (Pumd.col(0).segment(1,2*N-1).array() * P.
 ↪segment(0,2*N-1).array()).matrix();
        B.segment(1,2*N-1) += (Pumd.col(1).segment(1,2*N-1).array() * P.
 ↪segment(1,2*N-1).array()).matrix();
        B.segment(1,2*N-1) += (Pumd.col(2).segment(1,2*N-1).array() * P.
 ↪segment(2,2*N-1).array()).matrix();
        P = A.lu().solve(B);
        # American condition
        for(int j = 0; j <= 2*N; j++){P(j) = max(P(j), max(K -␣
 ↪(S0+(N-j)*dS),0));}
    }
    return P;
}
```

[ ]:
```
# follow the first version of notes
MatrixXd EFD_S(float dt, float sigma, float r, float dS, float S1, float S2,␣
 ↪float K, float T)
```

```
{
    int n = T/dt, N = int((S2-S1)/dS/2);
    float S0 = (S1+S2)/2;
    MatrixXd Pumd(2*N+1,3);
    VectorXd P(2*N+1), B(2*N+1);
    Pumd.setZero();
    P.setZero();
    B.setZero();

    # initiate A
    MatrixXd A = MatrixXd::Identity(2*N+1, 2*N+1);

    # initiate Pumd
    for(int i = N; i >= -N; i--)
    {
        Pumd.row(N-i) << dt/2*(pow(sigma*(S0+i*dS)/dS,2)+r*(S0+i*dS)/dS),␣
 ↪1-dt*pow(sigma*(S0+i*dS)/dS,2)+r*dt, dt/2*(pow(sigma*(S0+i*dS)/
 ↪dS,2)-r*(S0+i*dS)/dS);
        if(i!=N and i!=-N){A.row(N-i).segment(N-i-1,3) = Pumd.row(N-i);}
    }
    A.row(0).segment(0,3) = Pumd.row(1);
    A.row(2*N-1).segment(2*N-2,3) = Pumd.row(2*N-1);

    # initiate the boudary value
    for(int i = 0; i <= 2*N; i++){P(i) = max(K - (S0+(N-i)*dS),0);}

    # iterate to get Put price(t=0)
    B(2*N) = dS;
    for(int i = n-1; i >= 0; i--)
    {
        P = A.lu().solve(B);
        # American condition
        for(int j = 0; j <= 2*N; j++){P(j) = max(P(j), max(K -␣
 ↪(S0+(N-j)*dS),0));}
    }
    return P;
}
```

[ ]:
```
MatrixXd IFD_S(float dt, float sigma, float r, float dS, float S1, float S2,␣
 ↪float K, float T)
{
    int n = T/dt, N = int((S2-S1)/dS/2);
    float S0 = (S1+S2)/2;
    MatrixXd Pumd(2*N+1,3);
    MatrixXd A(2*N+1,2*N+1);
    VectorXd P(2*N+1), B(2*N+1);
    Pumd.setZero();
```

```
    A.setZero();
    P.setZero();
    B.setZero();

    # initiate A
    VectorXd p(3);
    A(0,0) = 1;
    A(0,1) = -1;
    A(2*N,2*N-1) = 1;
    A(2*N,2*N) = -1;
    for(int i = N-1; i >= -N+1; i--)
    {
        Pumd.row(N-i) << -dt/2*(pow(sigma*(S0+i*dS)/dS,2)+r*(S0+i*dS)/dS),␣
 ↪1+dt*pow(sigma*(S0+i*dS)/dS,2)+r*dt, -dt/2*(pow(sigma*(S0+i*dS)/
 ↪dS,2)-r*(S0+i*dS)/dS);
        A.row(N-i).segment(N-i-1,3) = Pumd.row(N-i);
    }

    # initiate the boudary value
    for(int i = 0; i <= 2*N; i++){P(i) = max(K - (S0+(N-i)*dS),0);}

    # iterate to get Put price(t=0)
    B(2*N) = - dS;
    for(int i = n-1; i >= 0; i--)
    {
        B.segment(1,2*N-1) = P.segment(1,2*N-1);
        P = A.lu().solve(B);
        # American condition
        for(int j = 0; j <= 2*N; j++){P(j) = max(P(j), max(K -␣
 ↪(S0+(N-j)*dS),0));}
    }
    return P;
}
```

[ ]:
```
MatrixXd CN_FD_S(float dt, float sigma, float r, float dS, float S1, float S2,␣
 ↪float K, float T)
{
    int n = T/dt, N = int((S2-S1)/dS/2);
    float S0 = (S1+S2)/2;

    MatrixXd Pumd(2*N+1,3);
    MatrixXd A(2*N+1,2*N+1);
    VectorXd B(2*N+1), P(2*N+1);
    Pumd.setZero();
    A.setZero();
    P.setZero();
    B.setZero();
```

```
    # initiate A
    VectorXd p(3);
    A(0,0) = 1;
    A(0,1) = -1;
    A(2*N,2*N-1) = 1;
    A(2*N,2*N) = -1;

    for(int i = N; i >= -N; i--)
    {
        Pumd.row(N-i) << -dt/4*(pow(sigma*(S0+i*dS)/dS,2)+r*(S0+i*dS)/dS), 1+dt/
↪2*(pow(sigma*(S0+i*dS)/dS,2)+r), -dt/4*(pow(sigma*(S0+i*dS)/
↪dS,2)-r*(S0+i*dS)/dS);
        if(i!=N and i!=-N){A.row(N-i).segment(N-i-1,3) = Pumd.row(N-i);}
    }

    # initiate the boudary value
    for(int i = 0; i <= 2*N; i++){P(i) = max(K - (S0+(N-i)*dS),0);}

    # iterate to get Put price(t=0)
    B(2*N) = - dS;
    Pumd.col(1).array() -= 2;
    for(int i = n-1; i >= 0; i--)
    {
        B.segment(1,2*N-1) = - (Pumd.col(0).segment(1,2*N-1).array() * P.
↪segment(0,2*N-1).array()).matrix();
        B.segment(1,2*N-1) += - (Pumd.col(1).segment(1,2*N-1).array() * P.
↪segment(1,2*N-1).array()).matrix();
        B.segment(1,2*N-1) += - (Pumd.col(2).segment(1,2*N-1).array() * P.
↪segment(2,2*N-1).array()).matrix();
        P = A.lu().solve(B);
        # American condition
        for(int j = 0; j <= 2*N; j++){P(j) = max(P(j), max(K -␣
↪(S0+(N-j)*dS),0));}
    }
    return P;
}
```

0.6.1 OUTPUT_6

[52]:
```
# EFD
EFD_5 = [0, 0, 4.48529e+298, 0, 7.46133e+298, 0, 8.42578e+298, 0, 7.75282e+298,␣
↪0, 6.18607e+298, 0, 4.41288e+298, 0, 2.86532e+298, 0, 1.71354e+298, 0, 9.
↪51733e+297, 0, 4.94033e+297, 0, 2.4084e+297, 0, 1.10696e+297, 0, 4.
↪81242e+296, 0, 1.98424e+296, 0, 7.77713e+295, 0, 2.9031e+295, 0, 1.
↪03299e+295, 0, 3.48228e+294, 0, 1.03153e+294, 0, 0]
```

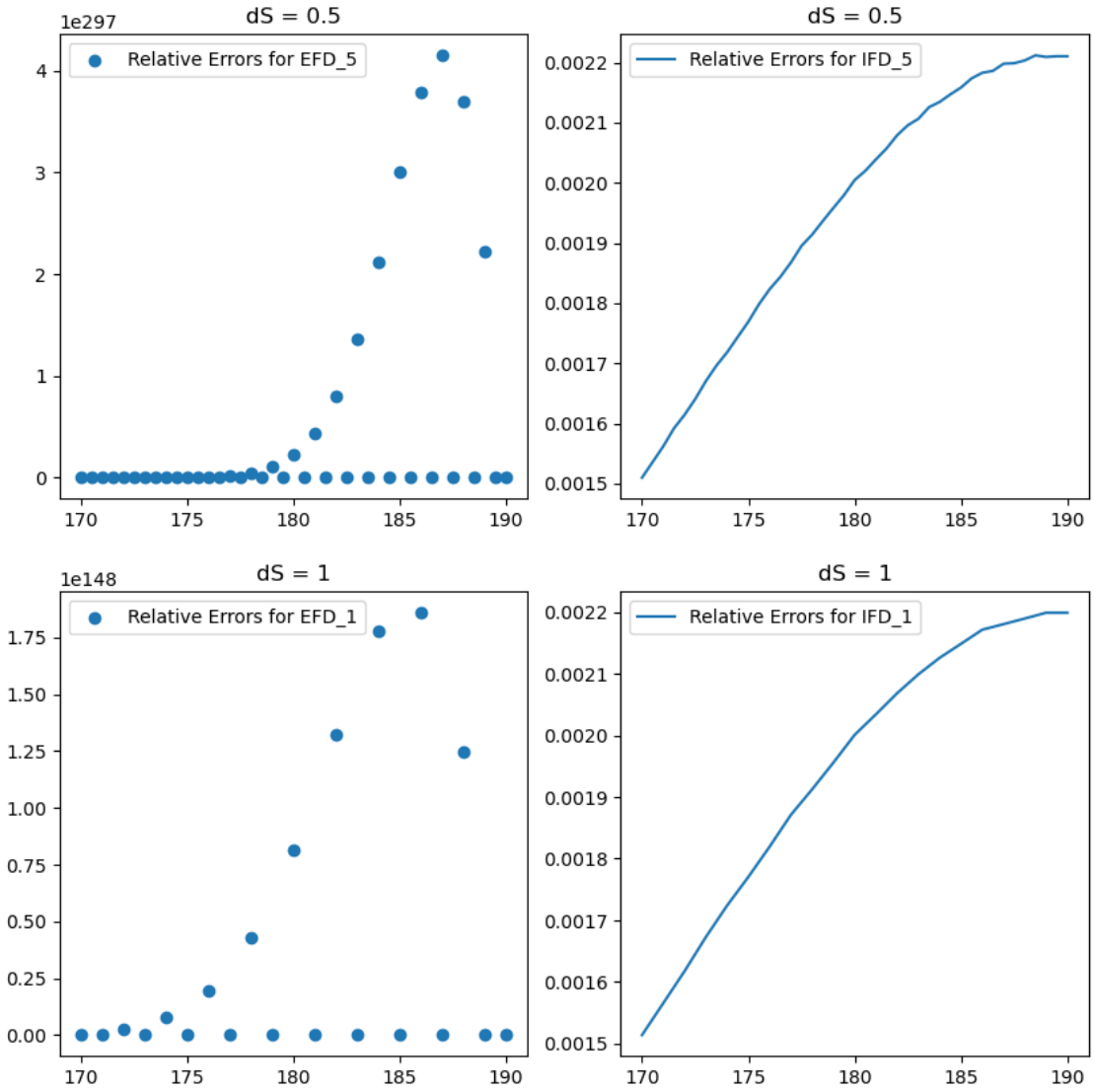
```
EFD_1 = [0, 0, 2.60293e+149, 0, 3.93074e+149, 0, 3.83703e+149, 0, 2.92664e+149,␣
 ↪0, 1.86417e+149, 0, 1.0248e+149, 0, 4.9494e+148, 0, 2.10214e+148, 0, 7.
 ↪24054e+147, 0, 0]
# IFD
IFD_5 = [20.1782, 20.1782, 20.1887, 20.2098, 20.2415, 20.284, 20.3375, 20.4019,␣
 ↪20.4775, 20.5643, 20.6624, 20.772, 20.8932, 21.0261, 21.1708, 21.3275, 21.
 ↪4962, 21.6771, 21.8703, 22.076, 22.2943, 22.5252, 22.769, 23.0258, 23.2957,␣
 ↪23.5789, 23.8754, 24.1855, 24.5093, 24.8469, 25.1984, 25.5641, 25.9441, 26.
 ↪3386, 26.7476, 27.1713, 27.61, 28.0638, 28.5328, 29.0172, 29.5172]
IFD_1 = [20.9222, 20.9222, 20.9656, 21.0534, 21.1864, 21.3655, 21.5917, 21.866,␣
 ↪22.1893, 22.5627, 22.9873, 23.464, 23.994, 24.5785, 25.2185, 25.9154, 26.
 ↪6703, 27.4845, 28.3592, 29.2959, 30.2959]
# CN_FD
CN_FD_5 = [20.1337, 20.1337, 20.1442, 20.1652, 20.197, 20.2395, 20.2929, 20.
 ↪3574, 20.4329, 20.5197, 20.6179, 20.7275, 20.8487, 20.9815, 21.1263, 21.
 ↪2829, 21.4516, 21.6326, 21.8258, 22.0315, 22.2497, 22.4807, 22.7245, 22.
 ↪9813, 23.2512, 23.5343, 23.8309, 24.141, 24.4647, 24.8023, 25.1539, 25.5196,␣
 ↪25.8996, 26.294, 26.703, 27.1268, 27.5655, 28.0192, 28.4883, 28.9727, 29.
 ↪4727]
CN_FD_1 = [20.8763, 20.8763, 20.9198, 21.0076, 21.1405, 21.3197, 21.5459, 21.
 ↪8202, 22.1435, 22.5169, 22.9414, 23.4182, 23.9482, 24.5326, 25.1727, 25.
 ↪8696, 26.6244, 27.4386, 28.3134, 29.2501, 30.2501]
```

[50]:
```
EFD_5 = [0, 0, 0, 0, 0.185525, 0.608327, 0.963734, 1.21031, 1.31851, 1.27441, 1.
 ↪08158, 0.761114, 0.349452, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.127146, 0.585076, 0.
 ↪971628, 1.23711, 1.34645, 1.28397, 1.05589, 0.689952, 0.23234, 0, 0, 0, 0,␣
 ↪0, 0, 0, 0, 0, 0.5]
EFD_1 = [0, 0, 0.427049, 1.17067, 1.3831, 0.954775, 0.0738221, 0, 0, 0, 0, 0.
 ↪506535, 1.26, 1.38144, 0.794972, 0, 0, 0, 0, 0, 1]
```

[53]:
```
from math import *
# Calculate x values for each series
x_5 = np.arange(190,169.5,-0.5)
x_1 = np.arange(190,169,-1)

# Calculate relative errors
def calculate_relative_error(method_values, reference_values):
    return np.abs((method_values - reference_values) / reference_values)

# Plotting
fig, axes = plt.subplots(2, 2, figsize=(10, 10))

# Plot for EFD, IFD, CN_FD with dS = 0.5
axes[0][0].scatter(x_5, calculate_relative_error(np.array(EFD_5), np.
 ↪array(CN_FD_5)), label='Relative Errors for EFD_5')
axes[0][0].set_title('dS = 0.5')
```

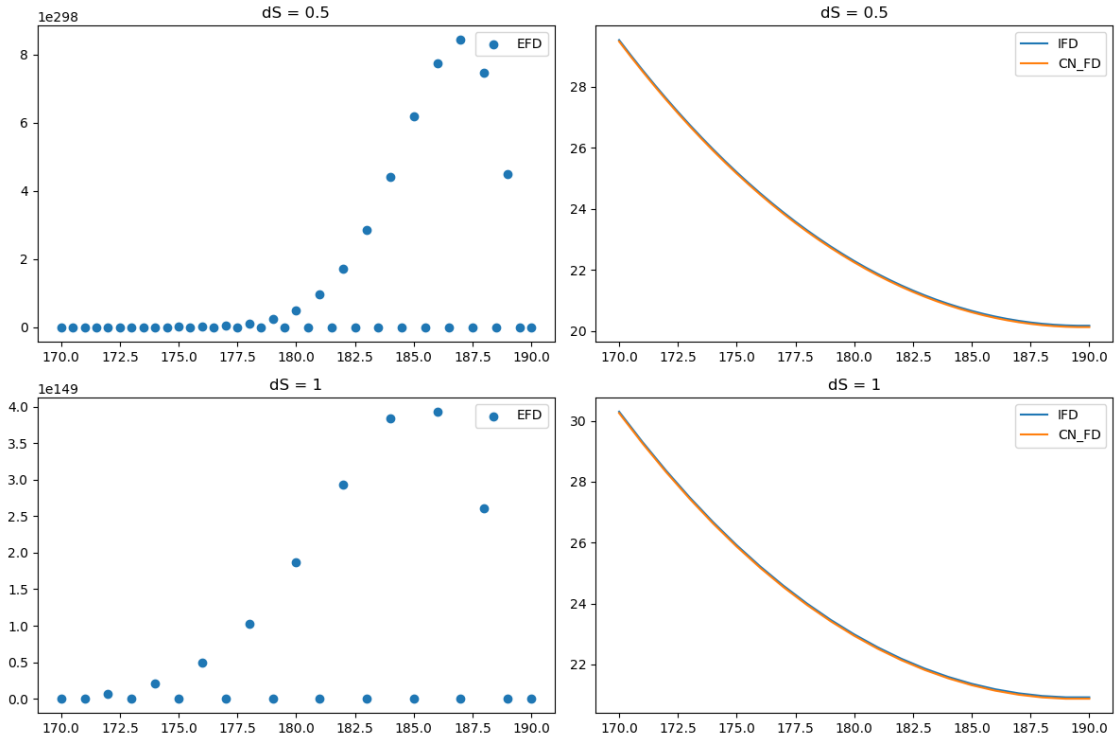
```
axes[0][0].legend()

axes[0][1].plot(x_5, calculate_relative_error(np.array(IFD_5), np.
 ↪array(CN_FD_5)), label='Relative Errors for IFD_5')
axes[0][1].set_title('dS = 0.5')
axes[0][1].legend()

# Plot for EFD, IFD, CN_FD with dS = 1
axes[1][0].scatter(x_1, calculate_relative_error(np.array(EFD_1), np.
 ↪array(CN_FD_1)), label='Relative Errors for EFD_1')
axes[1][0].set_title('dS = 1')
axes[1][0].legend()

axes[1][1].plot(x_1, calculate_relative_error(np.array(IFD_1), np.
 ↪array(CN_FD_1)), label='Relative Errors for IFD_1')
axes[1][1].set_title('dS = 1')
axes[1][1].legend()
```

[53]: <matplotlib.legend.Legend at 0x134c8a2d0>



[14]:
```
from math import *
import numpy as np
import matplotlib.pyplot as plt
# Calculate x values for each series
x_5 = np.arange(190,169.5,-0.5)
x_1 = np.arange(190,169,-1)

# Plotting
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

# Plot for EFD, IFD, CN_FD with dS=0.5
axes[0][0].scatter(x_5, EFD_5, label='EFD')
axes[0][0].set_title('dS = 0.5')
```

```
axes[0][0].legend()

axes[0][1].plot(x_5, IFD_5, label='IFD')
axes[0][1].plot(x_5, CN_FD_5, label='CN_FD')
axes[0][1].set_title('dS = 0.5')
axes[0][1].legend()

# Plot for EFD, IFD, CN_FD with dS=1
axes[1][0].scatter(x_1, EFD_1, label='EFD')
axes[1][0].set_title('dS = 1')
axes[1][0].legend()

axes[1][1].plot(x_1, IFD_1, label='IFD')
axes[1][1].plot(x_1, CN_FD_1, label='CN_FD')
axes[1][1].set_title('dS = 1')
axes[1][1].legend()

# Adjust plot spacing
plt.tight_layout()

# Show plot
plt.show()
```



Relative Errors Analysis:

EFD Errors ($\Delta S = 0.5$ and $\Delta S = 1$):

The EFD method shows extremely high variability in relative errors as depicted by the scattered plot. The relative errors increase dramatically as the stock price increases, indicating a significant loss of numerical stability or accuracy. This variability is especially pronounced at smaller steps ($\Delta S = 0.5$), which may indicate numerical instability typical of explicit methods which are conditionally stable depending on the size of the time and space steps used.

IFD Errors ($\Delta S = 0.5$ and $\Delta S = 1$):

The relative errors for IFD show a smooth and continuous increase as the stock price increases. This method displays higher stability compared to EFD but still suffers from increasing errors with higher stock prices. This trend is consistent for both step sizes, though the growth in error is less abrupt than in EFD, suggesting better handling of numerical diffusion.

Option Prices Comparison:

IFD vs. CN_FD ($\Delta S = 0.5$ and $\Delta S = 1$):

Both IFD and CN_FD methods show a decrease in option prices as the stock price increases, which is expected behavior. The two methods appear to converge closely, with CN_FD slightly underpricing compared to IFD, especially as the stock price increases. This could be due to CN_FD's averaging nature (balancing between implicit and explicit characteristics), which might slightly dampen the response to changes in the underlying price dynamics.

Key Insights:

- Stability and Accuracy: EFD, while straightforward to implement, shows potential numerical instability as indicated by the highly variable and extreme values of relative errors. Both IFD and CN_FD provide more stable error behaviors, with CN_FD potentially offering the best stability due to its centered time-stepping approach.
- Method Suitability: The choice between IFD and CN_FD might depend on specific requirements for accuracy and stability, particularly under conditions of rapidly changing market dynamics. CN_FD's method, due to its inherent averaging of time steps, could be preferred in environments where smoother price transitions are critical.
- Practical Implementation Considerations: When implementing these numerical methods in practical scenarios, attention must be paid to step sizing. Smaller step sizes, while potentially increasing accuracy, can lead to increased computational cost and numerical instability in methods like EFD. Adjustments in step size should consider the trade-offs between computational efficiency, accuracy, and stability.

[ ]: