1 Question 1

1.1 Deriving the Crank-Nicolson Scheme

Introduction: Let \( V = V(S_t, t) \) be the value of an American put option at time \( t \) when the underlying stock price is \( S_t \). Per definition, the one-time pay-off acquired upon exercising the put is \([K - S(t)]^+\) for any \( t \) up to and including the maturity, where \( K \) is the predetermined strike.

Using a simple replication argument it can be shown that \( V \) must satisfy the Black Scholes equation

\[
\frac{\partial V}{\partial t} + rS \frac{\partial V}{\partial S} + \frac{1}{2} \sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} = rV
\]

where \( r \) is the risk free rate, and \( \sigma \) is the volatility of the stock, assuming that no dividend is paid. One way to go about solving this PDE is to implement finite difference methods, i.e. to discretize the stock and time dimensions and then solve for \( V \) at any one of the resulting grid points. Explicitly, if the option exists between \( t = 0 \) and \( T \), then we can divide this interval into \( N \) equally spaced sub-intervals of length \( \delta t = T/N \), giving us a total of \( N + 1 \) time steps: \{0, \delta t, 2\delta t, ..., T\}. Likewise, suppose \( S_{\text{max}} \) is the stock price at which the value of the put for all practical purposes is zero. By dividing the range of possible stock values into \( M \) subintervals, each of width \( \delta S = S_{\text{max}}/M \), we obtain \( M + 1 \) discrete stock prices: \{0, \delta S, 2\delta S, ..., S_{\text{max}}\}. Jointly, discrete time and stock space thus forms a grid of \((N + 1) \cdot (M + 1)\) points, where a given 2-tuple \((t_n, S_j) \equiv (n, j)\) represents time \( n \cdot \delta t \) and stock value \( j \cdot \delta S \), \( n \in \{0, 1, ..., N\}, j \in \{0, 1, ..., M\} \).

In order to derive the Crank-Nicolson (CN) scheme for the Black Scholes equation, it is incumbent that we elucidate the implicit and explicit finite difference methods of which CN is an equally weighted average.

The implicit method: Defining \( V^n_j \equiv V(S_j, t_n) \) let us work systematically through equation (1) to obtain the implicit finite difference scheme. Firstly, the temporal derivative \( \partial V/\partial t \) at \((n, j)\) is reasonably approximated by the forward difference

\[
\frac{\partial V(S_j, t_n)}{\partial t} \approx \frac{V^n_{j+1} - V^n_j}{\delta t}.
\]

Secondly, the first stock derivative \( \partial V/\partial S \) at \((n, j)\) is approximated by a central difference quotient across \((n, j - 1)\) to \((n, j + 1)\):

\[
\frac{\partial V(S_j, t_n)}{\partial S} \approx \frac{V^n_{j+1} - V^n_{j-1}}{2\delta S}.
\]

Finally, the second order stock derivative is taken to be the

\[
\frac{\partial^2 V(S_j, t_n)}{\partial S^2} \approx \frac{V^n_{j+1} - V^n_{j-1}}{\delta S} = \frac{V^n_{j+1} - 2V^n_j + V^n_{j-1}}{\delta S^2}.
\]

whence the discretized Black Scholes equation becomes (using \( S = j\delta S \)):

\[
\frac{V^n_{j+1} - V^n_j}{\delta t} + rj\delta S \frac{V^n_{j+1} - V^n_{j-1}}{2\delta S} + \frac{1}{2} \sigma^2 j^2 \delta S^2 \frac{V^n_{j+1} - 2V^n_j + V^n_{j-1}}{\delta S^2} = rV^n_j
\]

where \( j \in [1, M - 1] \) and \( n \in [0, N - 1] \). Equation (2) is trivially rearranged such as to relate \( V(S_j) \) at time \( t_{n+1} \) to three different put values \( \{V(S_i)\}_{i=j-j-1}^{j+1} \) at time \( t_n \). In practice this implies that upon working our way
where the coefficients are defined as

\[
\frac{\partial V(S_j, t_n)}{\partial t} + \frac{1}{2} \sigma^2 j^2 \delta S^2 \left( \frac{V_{j+1}^{n+1} - 2V_j^{n+1} + V_{j-1}^{n+1}}{2\delta S} \right) \approx \frac{V_j^{n+1} - V_j^n}{\delta t} + \frac{1}{2} \sigma^2 j^2 \delta S^2 \left( \frac{V_{j+1}^{n+1} - 2V_j^{n+1} + V_{j-1}^{n+1}}{2\delta S} \right) - rV_j^n
\]

which evidently relates three different put values \( \{V(S_i)\}_{i=j-1}^{j+1} \) at time \( n+1 \) to one put value \( V(S_j) \) at time \( n \).

In other words, the explicit scheme does not give rise to a situation in which we have to solve any simultaneous equations. Whilst this significantly reduces the algorithmic complexity, the approximations inherent to the explicit scheme will inevitably lead to reduced accuracy of the results, \( \text{[2]} \).

**The Crank-Nicholson method** Taking the equally weighted average of the implicit scheme (2) and the explicit scheme (3) we obtain

\[
\frac{V_j^{n+1} - V_j^n}{\delta t} + \frac{1}{2} \sigma^2 j^2 \delta S^2 \left( \frac{V_{j+1}^{n+1} - 2V_j^{n+1} + V_{j-1}^{n+1}}{2\delta S} \right) \approx \frac{V_j^{n+1} - V_j^n}{\delta t} + \frac{1}{2} \sigma^2 j^2 \delta S^2 \left( \frac{V_{j+1}^{n+1} - 2V_j^{n+1} + V_{j-1}^{n+1}}{2\delta S} \right) - rV_j^n
\]

or simply

\[
a_j V_{j-1}^{n+1} + b_j V_j^{n+1} + c_j V_{j+1}^{n+1} = a_j V_{j-1}^n + b_j V_j^n - c_j V_{j+1}^n
\]

where the coefficients are defined as

\[
a_j \equiv \frac{1}{2} \delta t (\sigma^2 j - r), \\
b_j \equiv (1 - \frac{1}{2} \sigma^2 j^2 \delta t), \\
c_j \equiv \frac{1}{2} \delta t (\sigma^2 j + r), \\
d_j \equiv (1 + (r + \frac{1}{2} \sigma^2 j^2 \delta t)).
\]

In matrix form this system of equations take on the form.
or more succinctly

\[ T_1 V^{(n+1)} + k^{(n+1)} = T_2 V^{(n)} \]  

where \( T_1 \) and \( T_2 \) are the two tridiagonal \( \mathbb{R}^{(M-1)\times(M-1)} \) matrices, \( V^{(n+1)} \), \( V^{(n)} \) are the \( \mathbb{R}^{(M-1)} \) put value vectors and \( k^{(n+1)} \) is the boundary time \( \mathbb{R}^{(M-1)} \) vector.

Since we know the terminal condition \([K - S(T)]^+\) (i.e. \( V^{(N)} \)) and are working our way backwards in time, we must compute

\[ V^{(n)} = T_2^{-1} \left( T_1 V^{(n+1)} + k^{(n+1)} \right) \]  

for \( n = N - 1, N - 2, ..., 1, 0 \).

Figure 1: Time and stock space has been discretized into a grid of \((N + 1) \cdot (M + 1)\) points. The time axis runs from 0 to maturity \( T \) and the incremental time step is \( \delta t \); stock prices run from 0 to \( S_{\text{max}} \) and the incremental price step is \( \delta S \). Three boundary conditions (blue) have been imposed for an American put on the grid: (i) the terminal condition is \( V(T,j \cdot \delta S) = \left[ K - j \cdot \delta S \right]^+ \), (ii) the put has zero value when \( S = S_{\text{max}} \) and (iii) the put has value \( K \) when \( S = 0 \). Finally, the red arrows highlight the mechanics of the Crank-Nicolson method: by relating three adjacent put values at time \( t_n+1 \) to three adjacent put values at time \( t_n \), and solving \( M - 1 \) simultaneous equations, we can find explicit values for the put option at time \( t_n \). This is done iteratively from maturity, \( T \), to the present, \( t = 0 \).
1.2 Implementation of the Scheme in C++

The boundary conditions: Upon implementing equation \((6)\) in a C++ a number of aspects need to be elucidated. First and foremost, there is the issue of boundary conditions (cf. e.g. the presence of the \(k^{(n)}\) vector). Till this end we observe that the \(t = T\) boundary is given by the well defined put option pay-off: whichever is greater of \(K - S(T)\) and 0. Accordingly, we stipulate that the value of the put is zero at all times when \(S = S_{\text{max}}\) (this certainly makes good sense when \(S_{\text{max}} \gg K\) although it is open to debate) and, analogously, that the value of the the put is \(K\) at all times when \(S = 0\). Formally

\[
\text{Boundaries} = \begin{cases} 
V^n_j = [K - j \cdot \delta S]^+, & \forall j \in [0, M] \\
V^n_M = 0, & \forall n \in [0, N] \\
V^n_0 = K, & \forall n \in [0, N]. 
\end{cases}
\]

Observe that our boundary conditions guarantee that \(k^{(n)}\) is a constant vector which we only have to compute once. Also notice that the American put offers the opportunity of early exercise whence \(V^n_j = [V^n_j, (CN), K - j \cdot \delta S]^+\).

The Crout algorithm: The next question that arises is how one should go about computing \(V^{(n)}\) from \(V^{(n+1)}\). A popular choice in the literature is to deploy iterative methods for linear equations such as the Successive Over-Relaxation (SOR) technique. Whilst this might engender a dramatic run time reduction for complex systems, it is of some academic interest to try to implement an “exact” procedure, where \(T_2\) is inverted as in \((7)\).

Since \(T_2\) is manifestly tridiagonal, our inversion algorithm is aptly chosen to be Croutian: i.e. we shall perform a decomposition of \(T_2\) into \(LU\) form, where \(L\) is a lower triangular matrix and \(U\) is an upper triangular matrix of \(\mathbb{R}^{(M-1)\times(M-1)}\) and subsequently solve

\[
LU^{(n)} = W^{(n+1)}
\]

where \(W^{(n+1)} = T_1V^{(n+1)} + k^{(n+1)} \in \mathbb{R}^{M-1}\). For notational convenience, let us relabel \(M - 1\) as \(n\) and the entries of \(T_2\) as \(a_{ij}\) then the first step in Crout’s algorithm is to compute the entries in the factorisation. This is most easily done by writing out a portion of the system explicitly.

\[
\begin{pmatrix}
    a_{11} & a_{12} & 0 & \cdots & 0 \\
    a_{21} & a_{22} & a_{23} & \vdots & \\
    0 & \ddots & \ddots & \ddots & 0 \\
    \vdots & & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\
    0 & \cdots & 0 & a_{n,n-1} & a_{n,n}
\end{pmatrix}
= 
\begin{pmatrix}
    l_{11} & 0 & 0 & \cdots & 0 \\
    l_{21} & l_{22} & 0 & \vdots & \\
    0 & \ddots & \ddots & \ddots & 0 \\
    \vdots & & l_{n-1,n-2} & l_{n-1,n-1} & 0 \\
    0 & \cdots & 0 & l_{n,n-1} & l_{nn}
\end{pmatrix}
\times 
\begin{pmatrix}
    1 & u_{12} & 0 & \cdots & 0 \\
    0 & 1 & u_{23} & \vdots & \\
    0 & \ddots & \ddots & \ddots & 0 \\
    \vdots & & 0 & 1 & u_{n-1,n} \\
    0 & \cdots & 0 & 0 & 1
\end{pmatrix}
\]

\[
\begin{pmatrix}
l_{11} & l_{11}u_{12} & 0 & \cdots & 0 \\
l_{21} & l_{21}u_{12} + l_{22} & l_{22}u_{23} & \vdots & \\
0 & \ddots & \ddots & \ddots & 0 \\
\vdots & & l_{n-1,n-2} & l_{n-1,n-1} + l_{n-1,n} & l_{n-1,n}u_{n-1,n} \\
0 & \cdots & 0 & l_{n,n-1} & l_{nn}
\end{pmatrix}
\]

where we have assumed that the leading diagonal of \(U\) is composed of 1s to match the degrees of freedom on each side of the equality. Generalizing from \((9)\), \(a_{11} = l_{11}, a_{i,i-1} = l_{i,i-1}\) (for \(i \in [2, n]\)), \(a_{ii} = l_{i-1,i-1}u_{i-1,i} + l_{ii}\) (for \(i \in [2, n]\)) and \(a_{i,i+1} = l_{ij}u_{i+1,j}\) (for \(i \in [1, n-1]\)). Hence, since we are interested in the \(u_{ij}\) and \(l_{ij}\), we must work our way systematically through each row of the matrix as follows:
Having obtained the entries of the lower and upper triangular matrices, our next task is to solve for $V^{(n)}$ in (8). Defining $Z^{(n)} = UV^{(n)}$ we first solve for $Z^{(n)}$ in $LZ^{(n)} = W^{(n+1)}$, and subsequently for $V^{(n)}$ in $UV^{(n)} = Z^{(n)}$. I.e.

\[
\begin{align*}
\text{Set} & & l_{11} = a_{11}; & & u_{12} = a_{12}/l_{11}; \\
\text{For } i = 2, ..., n-1 & & l_{i,i-1} = a_{i,i-1}; & & l_{ii} = a_{ii} - l_{i,i-1}u_{i-1,i}; & & u_{i,i+1} = a_{i,i+1}/l_{ii}; \\
\text{Set} & & l_{n,n-1} = a_{n,n-1}; & & l_{nn} = a_{nn} - l_{n,n-1}u_{n-1,n}; \\
\end{align*}
\]

This completes the Crout algorithm: by factorising $T_2$ into $LU$ and isolating $V^{(n)}$ step-wise in (8), we’ve computed the put prices at $t_n$ using the put prices at $t_{n+1}$. Notice a mere $(5n - 4)$ multiplications/divisions, and $(3n - 3)$ additions/subtractions are needed to solve a tridiagonal system of $n$ unknowns. Contrast this to a Gaussian elimination with backward substitution scheme in which $n^3/3 + n^2 - n/3$ multiplications/divisions and $n^3/3 + n^2/2 - 5n/6$ additions/subtractions are needed, [1].

**The code:** Having outlined the nature of the boundary conditions and the numerical procedure by which we move between temporally adjacent put value vectors, $V^{(n+1)} \rightarrow V^{(n)}$, we now only need to make a few remarks about the overall structure and output of the code (see Appendix). In accordance with good programming practice, a reasonably high degree of modularity has been upheld: the main computation of the put values is done within the class function `DiffMethod`, which in turn makes use of the matrix-vector multiplication function, `matrixvector`, the Crout algorithm function, `crout`, and the maximum of two numbers function, `max`. This implies that it is fairly straightforward to implement, say, an iterative linear equation solver without making radical changes to the code.

As for the input/output data, the user can enter the maturity $T$, maximum stock value $S_{\text{max}}$, interest rate $r$, volatility $\sigma$, strike price $K$ and the number of time and stock intervals $(N, M)$ in the `main` function. The `DiffMethod` will then output a table in the terminal window which lists all possible (discrete) stock prices in the first row and all possible (discrete) time steps in the first column. Inside the table we can track the evolution of the put prices backwards in time (see figure 2). Naturally, the top row of the put values simply gives us the defined pay-off of an American put, but already in the second row of the put values we see that some figures have changed. For example, in the example provided $(T, S_{\text{max}}, r, \sigma, K, N, M) = (0.4167, 100, 0.10, 0.40, 50, 10, 20)$, the value of $V(S = 55)$ at maturity is clearly zero (as $S > K = 50$). However, just half a month before that $V(S = 55) = 0.16$ as there still is a nonzero chance that the stock price fill fall below the strike before maturity. If we are interested in the value of the put *today*, five months before expiration, given that $S = 55$, we look at the bottom row $(t = 0)$ under the appropriate stock value column to find $V(S = 55) = 2.50$ - a price noticeably above zero.

The only question that remains pertains to the performance of the Crank Nicolson scheme vis-a-vis the implicit and explicit finite difference methods. The parameter specifications listed above are deliberately matching an example by Hull, [2], who provides similar tables for both procedures: as it happens, the implicit method yields $V(S = 55) = 2.43$ and the explicit method yields $V(S = 55) = 2.59$ five months before maturity. Hence, we see that the Crank Nicolson scheme generates a put value (2.50) which falls almost exactly in between these figures. Intuitively, this makes good sense as CN is defined as the equally weighted average of the implicit and the explicit procedures. Indeed, upon scrutinizing put value figures around the $S = 50$ mark, similar conclusions can be drawn.

I conclude that the Crank Nicolson method with Croutian matrix inversion constitutes a more than adequate pricing algorithm for American put options.
Figure 2: Screenshot of the terminal output. In this particular case we operate with the parameter specifications: \( T = 0.4167 \) years (5 months), \( S_{\text{max}} = 100, r = 0.10, \sigma = 0.40, K = 50, N = 10 \) and \( M = 20 \). Put prices are therefore computed on a bimonthly basis starting at maturity \( T \) and finishing in the present \( t = 0 \). The table above prints all discrete stock prices in the first row, and all discrete time steps in the first column. Thus, if we wish to find the price of an American put today, if the stock price is \( S = 55 \), we look under row \( 0.00 \) and column \( 55.0 \): the answer is \( V(S = 55) = 2.50 \) dollars.

References

A Appendix

Listing 1: American Put Option

```cpp
#include <iostream.h>
#include <math.h>
#include <iomanip.h>

//=================================== Prototypes ====================================
void matrixvector(float **T1, float *V, float *k, float *vec1, int M);
void crout(float **A, float *b, float *x, int n);
float max(float a, float b);

//=================================== Put Class ====================================
class DiffMethod{
private:
  float dt, dS;
public:
  DiffMethod(float T, float Smax, float r, float sd, float K, int N, int M);
  ~DiffMethod(){};
};

DiffMethod::DiffMethod(float T, float Smax, float r, float sd, float K, int N, int M){
  //specify increments of time and stock price
  dt = (float) T/N;
  dS = (float) Smax/M;
  //allocate memory to coefficients
  float *a = new float[M-1];
  float *b = new float[M-1];
```
float *c = new float[M-1];
float *d = new float[M-1];

// compute all a, b, c and d coefficients
for (int j = 0; j < (M-1); j++){
a[j] = 0.25*(j+1)*dt*(pow(sd,2)*(j+1)-r);
b[j] = (1-0.5*pow(sd*(j+1),2)*dt);
c[j] = 0.25*(j+1)*dt*(pow(sd,2)*(j+1)+r);
d[j] = (1+(r+0.5*pow(sd*(j+1),2))*dt);
}

// allocate memory to the T matrices
float **T1 = new float*[M-1];
float **T2 = new float*[M-1];
for (int j = 0; j < (M-1); j++){
    T1[j] = new float[M-1];
    T2[j] = new float[M-1];
}

// fill out the T1 and T2 matrices
for (int j = 0; j < (M-1); j++){
    for (int i = 0; i < (M-1); i++){
        T1[j][i] = 0.0;
        T2[j][i] = 0.0;
    }
}

// allocate memory to the V and K vectors
float *V = new float[M-1];
float *k = new float[M-1];

// fill out the K vector
k[0]=a[0]*(2.0*K);
for (int j = 1; j < (M-1); j++){
    k[j] = 0.0;
}

// print all stock prices in the table
cout.setf(ios::fixed);
cout << setw(7) << "Stock:";
for (int i = 0; i < (M+1); i++){
    cout << setw(6) << setprecision(1) << i*dS;
}
cout << endl;
cout << setw(7) << "Time:";
for (int i = 0; i < (M+1); i++){
    cout << setw(6) << "-----";
}
cout << endl;

// fill out the terminal values for V and print them
cout << setw(6) << setprecision(2) << N*dt << "|";
cout << setw(6) << K;
for (int j = 0; j < (M-1); j++){
    V[j] = max(K-dS*(j+1),0);
    cout << setw(6) << V[j];
}
cout << setw(6) << "0.00" << endl;

// allocate memory for vec1 = T1*V + k
float *vec1 = new float[M-1];

// this loop solves the system T1*V(n+1) + k = T2*V(n).
// first the LHS is computed using the matrixvector function
// then V(n) is isolated and computed by inverting T2 using the
// crout algorithm. All values are printed
for(int j = N; j > 0; j--){
    matrixvector(T1, V, k, vec1, M);
    crout(T2, vec1, V, M);
    cout << setw(6) << setprecision(2) << (j-1)*dt << "|" <<
    cout << setw(6) << K;
    for(int i = 0; i < (M-1); i++){
        V[i] = max(V[i], K-dS*(i+1));
        cout << setw(6) << V[i];
    }
    cout << setw(6) << "0.00" << endl;
}

// clean up
delete [] a;
delete [] b;
delete [] c;
delete [] d;
for(int j = 0; j < (M-1); j++){
    delete T1[j];
delete T2[j];
delete [] V;
delete [] k;
}

//=======================================================================

int main(){
    DiffMethod example(0.4167, 100.0, 0.10, 0.40, 50.0, 10, 20);
    return 0;
}

//=======================================================================

// this function returns the maximum of two numbers
float max(float a, float b){
    return ((a>b)?a:b);
}

//=======================================================================

// this function multiplies a square matrix with a vector
void matrixvector(float **T1, float *V, float *k, float *vec1, int M){
    for(int i = 0; i < (M-1); i++){
        float temp = 0;
        for(int j = 0; j < (M-1); j++)
            temp = temp + T1[i][j]*V[j];
        vec1[i] = temp + k[i];
    }
}

//=======================================================================

// this generic function solves the system Ax=b where A is tridiagonal using the Crout algorithm
void crout(float **A, float *b, float *x, int n) {
    n = n-1; // adapt size as we pass M but want dimension M-1

    // reserve memory
    float *z = new float[n];
    float **L = new float*[n];
    for(int i=0; i<n; i++) {
        L[i] = new float[n];
    }
    float **U = new float*[n];
    for(int i=0; i<n; i++) {
        U[i] = new float[n];
    }

    // This bit does the factorization A = LU
    L[0][0] = A[0][0];
    U[0][1] = A[0][1]/L[0][0];
    for(int i=1; i<=(n-2); i++) {
        L[i][i-1] = A[i][i-1];
        L[i][i] = A[i][i] - L[i][i-1]*U[i-1][i];
        U[i][i+1] = A[i][i+1]/L[i][i];
    }
    L[n-1][n-2] = A[n-1][n-2];
    L[n-1][n-1] = A[n-1][n-1] - L[n-1][n-2]*U[n-2][n-1];

    // This bit solves for z in Lz = b where z = Ux
    z[0] = b[0]/L[0][0];
    for(int i=1; i<=(n-1); i++) {
        z[i] = (b[i] - L[i][i-1]*z[i-1])/L[i][i];
    }

    // This bit solves for x in Ux = z
    x[n-1] = z[n-1];
    for(int i=(n-2); i>=0; i--){
        x[i] = z[i] - U[i][i+1]*x[i+1];
    }

    // clean up
    delete []z;
    for(int j=0; j<n; j++) {
        delete L[j];
        delete U[j];
    }
    delete []L;
    delete []U;
}