Accurate and Efficient Numerical Methods for Pricing and Hedging American Put Options

Jun Ouyang*,
supervised by Professor Kenneth Jackson,[1]

[1]Department of Computer Science, University of Toronto,
10 King's College Rd, Toronto, ON, M5S 3G4, Canada
*To whom correspondence should be addressed; E-mail: jun.ouyang@www.utoronto.ca.

Accurate and efficient methods for pricing and hedging financial derivatives is critical in many financial applications. Many derivatives have well developed mathematical models, but the models do not have a closed form solution. The pricing of American put options is a case in point.

1 Introduction

This report explores the different methods for solving European, and, most importantly, American put options derived from the well-known *Black-Scholes* equation. For European put options, we transform the problem to a diffusion equation and develop mesh approximation to numerically solve the partial differential equation. All methods are based on numerical algorithms and the results are compared and analyzed to the explicit solution of the *Black-Scholes* formula. For American put options, the idea of the linear complementary problem is used to solve the free boundary condition in the partial differential equation. Mainly, we focus on both accuracy and efficiency of each algorithm.

2 Background

2.1 European and American Put Option

A put option is a fundamental financial instrument. The value of a put option at the termination of the contract is given by the following equation:

$$P(S,T) = max(E - S, 0),$$

where E is the exercise price and S is the value of underlying assets. The put option is called European if we can only exercise the option on maturity T. In contrast, an American put option can be exercised at any time up to maturity.

2.2 Black-Scholes Equation

In the *Black-Scholes* model, we treat a financial instrument $V(S,t)$ as a function of time t and the value of the underlying asset S. For the underlying asset S, we assume:

1. S follows a the lognormal random walk process, namely:

$$\frac{dS}{S} = \sigma dX + \mu dt$$

2. The risk free interest rate r and asset volatility σ are known constants.

3. There are no transaction costs, no dividends, and no arbitrage opportunity.

Ito's Lemma: If $f(S,t)$ is a function of a random variable S and time t, df satisfies:

$$df = \sigma S \frac{\partial f}{\partial S} dX + (\mu S \frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} + \frac{\partial f}{\partial t})dt$$

We construct a well-designed portfolio Π consisting of the put option V and the underlying asset S such that $\Pi = V - \Delta S$. By choosing $\Delta = \frac{\partial V}{\partial S}$ and applying Ito's Lemma, $d\Pi$ can be

expressed as:

$$d\Pi = (\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2})dt$$

Since $d\Pi$ is risk-free, by our arbitrage-free assumption, it must satisfy:

$$d\Pi = r\Pi dt$$

Therefore,

$$(\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2})dt = r\Pi dt$$

Then we plug in the assumed portfolio $\Pi = V - \frac{\partial V}{\partial S}S$ and re-arrange the equation to get:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

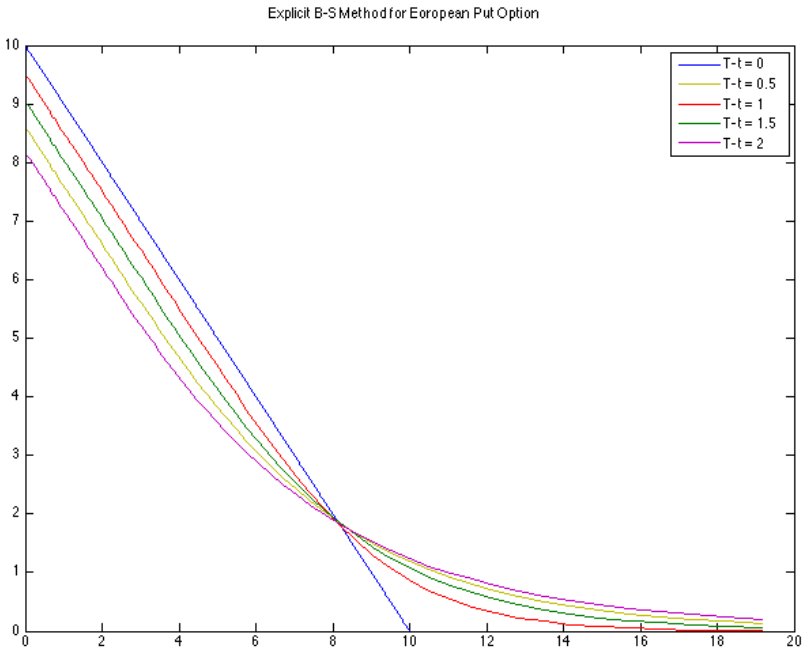
This is the well-known *Black-Scholes* equation. Note this can be transformed to a diffusion equation after simply change of variables. The *Black-Scholes* formula for European put options is given by:

$$P(S,t) = Ee^{-r(T-t)}N(-d_2) - SN(-d_1),$$

where

$$d_1 = \frac{log(S/E) + (r + \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{(T-t)}}$$
$$d_2 = \frac{log(S/E) + (r - \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{(T-t)}}$$

The following plot is based on the *Black-Scholes* formula with $E = 10, r = 0.1, \sigma = 0.4$:



3 Numerical Approach to European Put Option

3.1 Variable Transformation

As mentioned in the last part, we can easily transform the *Black-Scholes* equation into a diffusion equation by change of variables. Take

$$S = Ee^x, t = T - \tau/1/2\sigma^2, V = Ev(x,\tau), k = r/\frac{1}{2}\sigma^2$$

Also, let

$$v = e^{1\frac{1}{2}(k-1)x - \frac{1}{4}(k+1)^2\tau}u(x,\tau)$$

The *Black-Scholes* equation now becomes

$$\frac{\partial u}{\partial \tau} = \frac{\partial^2 u}{\partial x^2}\ for -\infty < x < \infty, \tau > 0,$$

with initial condition:

$$u(x,0) = max(e^{\frac{1}{2}(k+1)x} - e^{\frac{1}{2}(k-1)x}, 0).$$

Let V be a put option P, the boundary conditions are:

$$\lim_{x\to-\infty} u(x,\tau) = e^{-\frac{1}{2}(k-1)x-\frac{1}{4}(k+1)^2\tau}(e^{-k\tau}-e^{x}),\ \lim_{x\to\infty} u(x,\tau) = 0$$

3.2 Finite Difference Mesh

In this method, we cut the (x,τ) domain into a finite mesh with tiny space. Denote $u_n^m = u(n\delta x, m\delta\tau)$. By forward difference approximation of $\frac{\partial u}{\partial x}$ and $\frac{\partial u}{\partial \tau}$, we have

$$\frac{u_n^{m+1}-u_n^m}{\delta\tau} + O(\delta\tau) = \frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{(\delta x)^2} + O((\delta x)^2)$$

and backward difference approximation:

$$\frac{u_n^{m}-u_n^{m-1}}{\delta\tau} + O(\delta\tau) = \frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{(\delta x)^2} + O((\delta x)^2)$$

If we take $\alpha = \frac{\delta\tau}{(\delta x)^2}$, the forward and backward difference approximation will be:

$$u_n^{m+1} = \alpha u_{n+1}^m + (1-2\alpha)u_n^m + \alpha u_{n-1}^m \ (1)$$

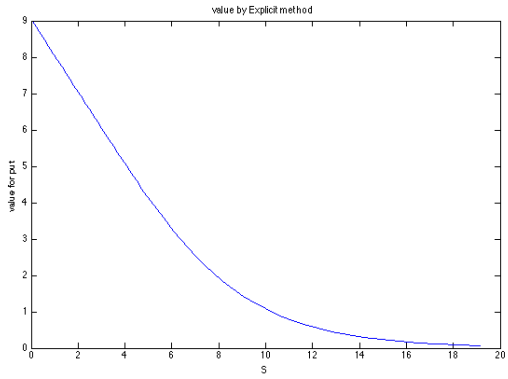
and

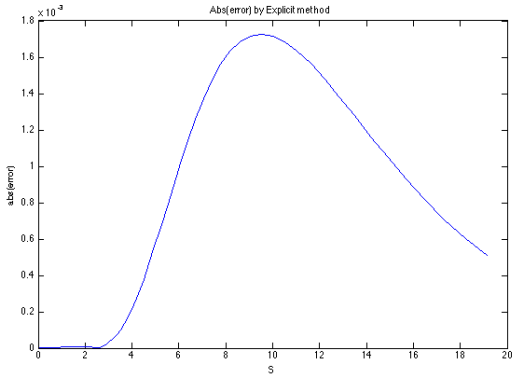
$$u_n^{m-1} = -\alpha u_{n-1}^m + (1+2\alpha)u_n^m - \alpha u_{n-1}^m. \ (2)$$

The following numerical algorithms solving these equations comes from [1]. In the textbook, algorithms are listed in Pseudocode format whose implementation is done by Matlab function listed in the Appendix. We focus on the accuracy and efficiency of each method.

3.3 Explicit Forward Method

The Matlab function is listed in Appendix (a). Briefly, the method is solving each entry of the mesh with a forward direction in τ. The iterating process is conducted on equation (1). However, this method restricts $\alpha < 1/2$. Value and error of European($T-t=1$) put option computed by this method is plotted here:





Note that in general, the magnitude of error compared to the *Black-Scholes* formula is less than 1.8e-3, which is sufficiently small for most applications. Also, the error increases as S approaches the exercise price $E = 10$. This is due to the fact that around $S = 10$ is a saddle point. When S is much smaller than 10, the result is approximately linear. If S is much larger than 10, the value is convergent to 0. In both cases, they leave small room for the approximation. Thus, the induced error is quite small. However, around $S = 10$, we have a transition from one linear line to another. In this case, the approximation is used heavily. Therefore, we observe larger error around $S = 10$.

3.4 Implicit LU Method

In fully implicit methods, the backward difference approximation instead of forward difference approximation. Note if we write (2) with different m together as a system of equations:

$$Mu^m = b^m$$

where

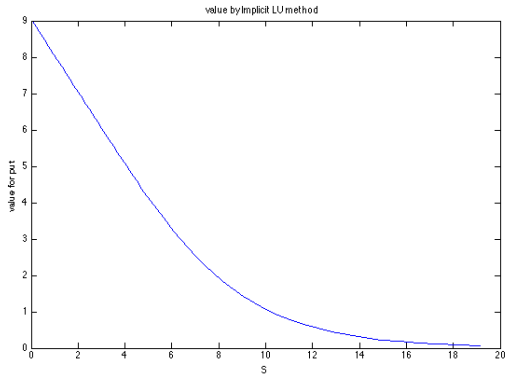
$$M = \begin{bmatrix} 1+2\alpha & -\alpha & 0 & \dots & 0 \\ -\alpha & 1+2\alpha & -\alpha & \dots & \vdots \\ \vdots & \vdots & \vdots & \ddots & -\alpha \\ 0 & 0 & \dots & -\alpha & 1+2\alpha \end{bmatrix}$$

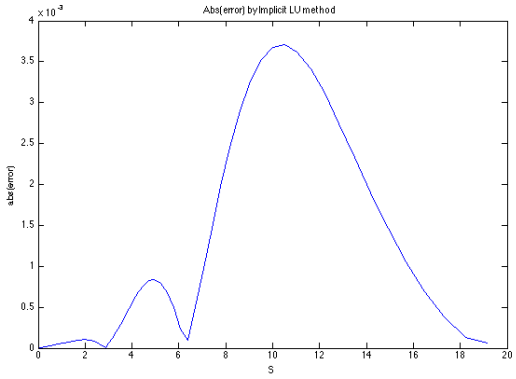
$$u^m = (u^m_{N^-+1}, \dots, u^m_{N^+-1}),\ b^m = u^{m-1} + \alpha(u^m_{N^-}, 0, \dots, 0, u^m_{N^+})$$

The LU method utilizes the unique trait of M that M is a tridiagonal matrix. Therefore, we can efficiently take the LU decomposition of M such that $M = LU$. Then the problem is transformed to two simpler problems that

$$Lq^m = b^m,\ Uu^m = q^m.$$

The algorithm is realized by Matlab function listed in Appendix (b, c, d). Value and error of this method computing same European put option is plotted here:





Note the magnitude of the absolute value of the error is less than 4e-3, which is less accurate than the explicit method. However, the plot shows a similar trend of error. When S approaches to 10, the error also increases and then decreases after 10. The reason for this behavior is the same analyzed as above.

3.5 Implicit SOR Method

In this method, we face the same problem as in implicit LU method, solving

$$Mu^m = b^m$$

However, in this time the **Gauss-Seidel** method is used. Let $u_n^{m,k}$ be the k-th iteration of u^m, where the initial guess is denoted by $u_n^{m,0}$. Then $u_n^{m,k+1}$ can be calculated by

$$u_n^{m,k+1} = \frac{1}{1+2\alpha}(b_n^m + \alpha(u_{n-1}^{m,k+1} + u_{n+1}^{m,k})),\ N^- < n < N^+$$

where kth-error is measured by

$$||u^{m,k+1} - u^{m,k}||^2 = \sum_n (u^{m,k+1} - u^{m,k})^2.$$

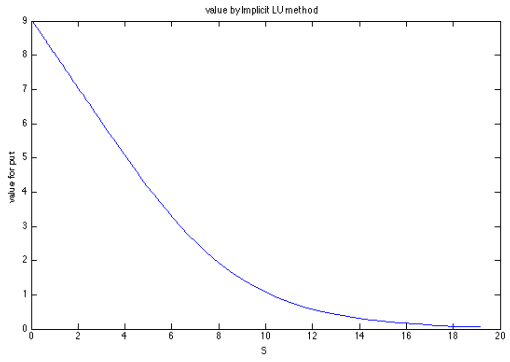
During the calculation of each u_m, the algorithm is iterated until convergence, which means the error is bounded by some small tolerance. **SOR** method is the refined method of **Gauss-Seidel** method. Basically, the speed of convergence can be controlled by a new parameter ω:

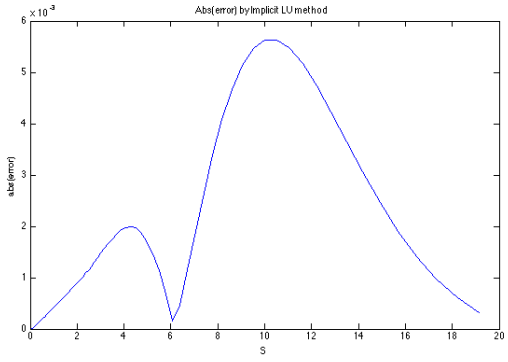
$$y_n^{m,k+1} = \frac{1}{1+2\alpha}(b_n^m + \alpha(u_{n-1}^{m,k+1} + u_{n+1}^{m,k}))$$

$$u_n^{m,k+1} = u_n^{m,k} + \omega(y_n^{m,k+1} - u_n^{m,k})$$

If $\omega > 1$, it is called over-relaxation. The algorithm is very aggressive to pursuit convergence. Else $\omega < 1$, it is called under-relaxation. The algorithm will take slower but more careful moves to the convergence. In the **SOR** method, the parameter ω is auto-adjusted based on the last iteration's error. The algorithm is realized by Matlab function listed in Appendix (e,f).

Value and error of this method computing the same European put option are plotted here:





The magnitude of error here is 6e-3, which is also comparably small to the magnitude of value. In the error plot, there is also one mode at $S = 10$. Note that this method is quite important as it can be extended to **PSOR**(detail described in section 4) which is a good approach to American put options.

3.6 Crank-Nicolson Method

In the above explicit method and implicit methods, only one type of difference approximation is used. However, **Crank-Nicolson Method** uses the average of forward and backward approximation. Different from **Explicit Method**, **Implicit Method**, and **Crank-Nicolson Method** do not have any restriction on α. Also, **Crank-Nicolson Method** provides a rather smaller order of error $O((\delta\tau)^2)$ compared to $O(\delta\tau)$ of simple explicit and implicit methods(Proved by [1]). By taking the average of equation (1) and (2), we have

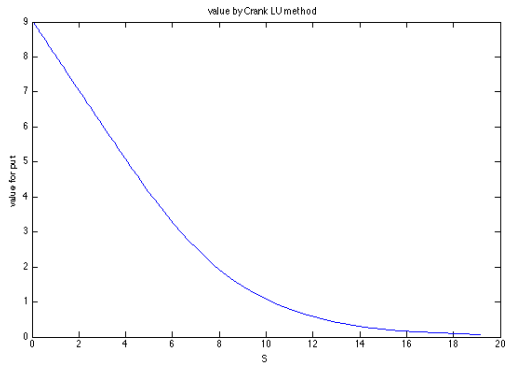
$$\frac{u_n^{m+1}-u_n^m}{\delta\tau}+O(\delta\tau)=\frac{1}{2}(\frac{u_{n+1}^m-2u_n^m+u_{n-1}^m}{(\delta x)^2}+\frac{u_{n+1}^{m+1}-2u_n^{m+1}+u_{n-1}^{m+1}}{(\delta x)^2})+O((\delta x)^2)$$

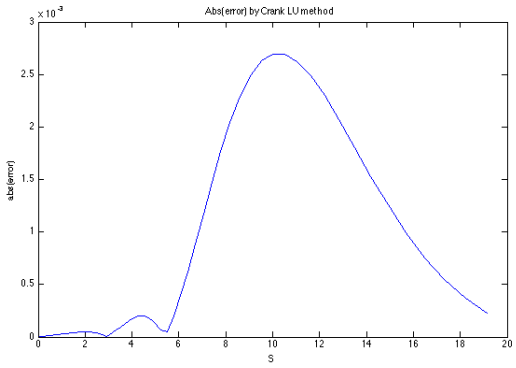
Ignoring the error term,

$$u_n^{m+1}-\frac{1}{2}\alpha(u_{n-1}^{m+1}-2u_n^{m+1}+u_{n+1}^{m+1})=u_n^m+\frac{1}{2}\alpha(u_{n-1}^m-2u_n^m+u_{n+1}^m). \quad (3)$$

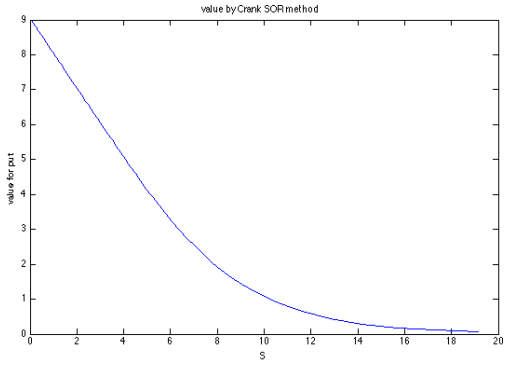
Similarly, we can solve (3) by both modified **LU** and **SOR** methods(detail omitted here). The relevant Matlab functions are listed in the Appendix(g,h). The result is plotted here.

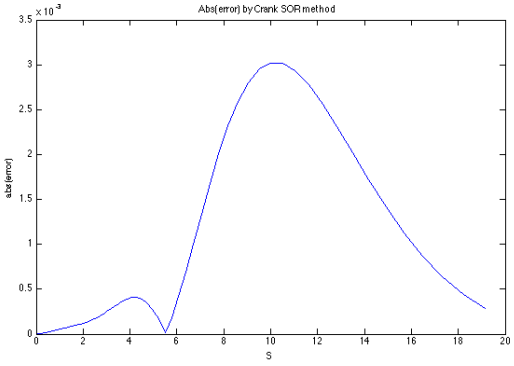
3.7 Crank-Nicolson LU Method





3.8 Crank-Nicolson SOR Method



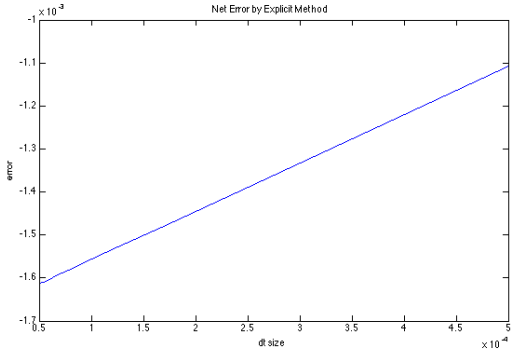


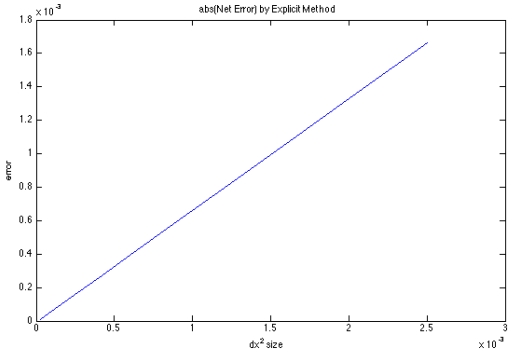
Note in either method, the magnitude of error is about 3e-3. Both errors are smaller than direct **LU** and **SOR** methods. This result corresponds to above analysis that **Crank's** estimation has a smaller order of error term.

3.9 Error Analysis Based on δx and $\delta\tau$

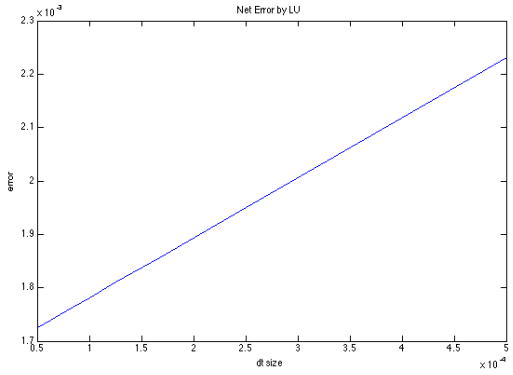
1. For **Explicit Method** and **Implicit Method**, we have proven the error is of order $O(\delta\tau) + O((\delta x)^2)$. To better check this important behavior, we can plot error versus $(\delta\tau)$ and $(\delta x)^2$ respectively.

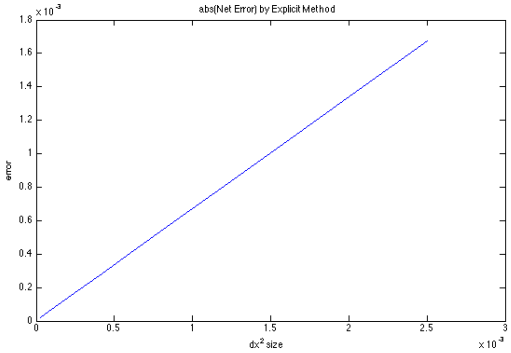
2. **Explicit Method**:





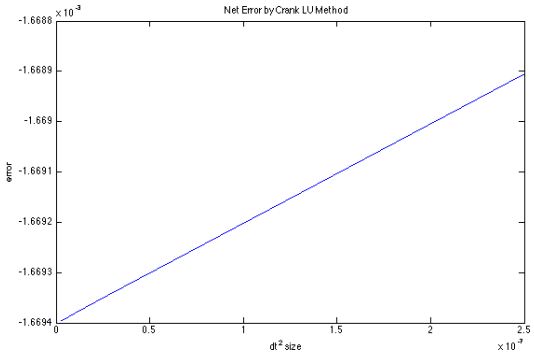
3. **Implicit Method**:

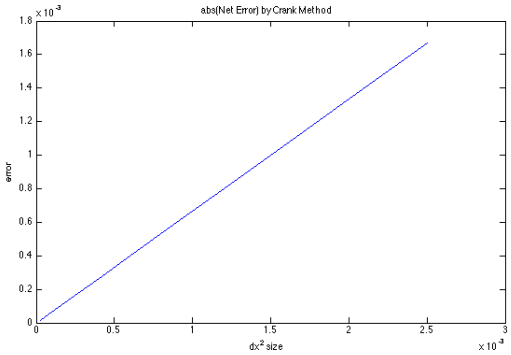




4. For **Crank-Nicolson Method**, we have proven the error is of order $O((\delta\tau)^2) + O((\delta x)^2)$. We can plot error versus $(\delta\tau)^2$ and $(\delta x)^2$ respectively.

5. **Crank-Nicolson Method**:





In all above plots, we can identify a very clear linear relationship between error and the variable examined. Therefore, the result corresponds to our assumption of the order of error of **Explicit Method**, **Implicit Method**, and **Crank-Nicolson Method**.

4 Numerical Approach to American Put Option

4.1 Linear Complementary Problem

What makes American put options different from European put options is only the time of exercise. As American put options can exercise early than European put option, it has more

"power" than European's. Therefore, by no arbitrage opportunity, American put option should have at least same price as European's. Also, for $P(S,t)$, there exist a free boundary $S_f(t)$ such that we exercise early than maturity if $S < S_f(t)$ and hold on if $S > S_f(t)$. By variable transformation to $u(x,\tau)$, there exist a new free boundary $x_f(\tau)$ such that

$$\frac{\partial u}{\partial \tau} = \frac{\partial^2 u}{\partial x^2} \; for \; x > x_f(\tau), \; else \; \frac{\partial u}{\partial \tau} > \frac{\partial^2 u}{\partial x^2} \; (4)$$

$$u(x,\tau) = g(x,\tau) \; for \; x \leq x_f(\tau), \; else \; u(x,\tau) > g(x,\tau) \; (5)$$

where $g(x,\tau)$ is the payoff function

$$g(x,\tau) = e^{\frac{1}{2}(k+1)^2\tau} max(e^{\frac{1}{2}(k-1)x} - e^{\frac{1}{2}(k+1)x}, 0).$$

Note that (4) and (5) can be also written into the form as

$$(\frac{\partial u}{\partial \tau} - \frac{\partial^2 u}{\partial x^2}) * (u(x,\tau) - g(x,\tau)) = 0 \; (6)$$

$$\frac{\partial u}{\partial \tau} - \frac{\partial^2 u}{\partial x^2} \geq 0, u(x,\tau) - g(x,\tau) \geq 0 \; (7)$$

(6) and (7) together are called a linear complementary problem as this combination is the same as explicitly (4) and (5) but efficiently eliminates the free boundary.

4.2 Numerical Solution and PSOR

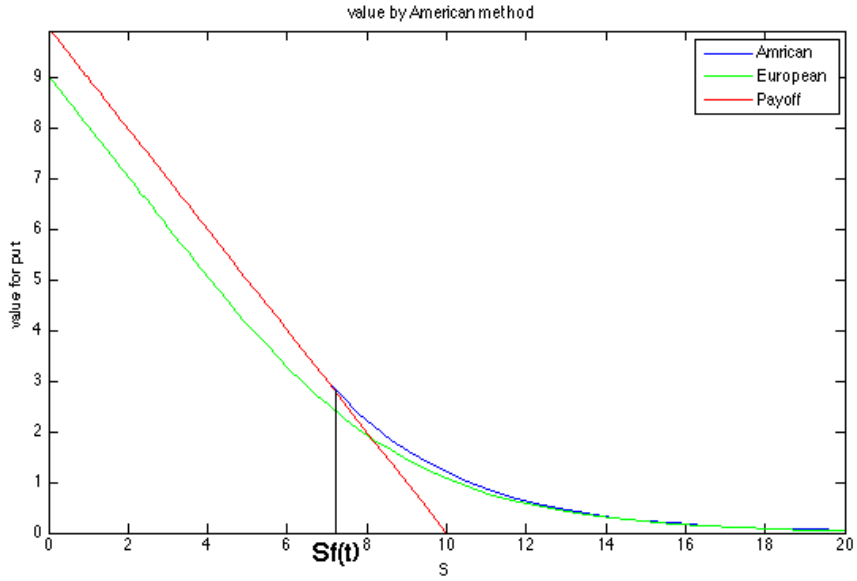
The numerical approach to this problem can be done by **PSOR**. Basically, we need to solve a modified version of a system of equations. where

$$C = \begin{bmatrix} 1+\alpha & -\frac{1}{2}\alpha & 0 & \dots & 0 \\ -\frac{1}{2}\alpha & 1+\alpha & -\frac{1}{2}\alpha & \dots & \vdots \\ \vdots & \vdots & \vdots & \ddots & -\frac{1}{2}\alpha \\ 0 & 0 & \dots & -\frac{1}{2}\alpha & 1+\alpha \end{bmatrix}$$

$$Cu^{m+1} \geq b^m, \; u^{m+1} \geq g^{m+1}$$

$$(u^{m+1} - g^{m+1})(Cu^{m+1} - b^m) = 0.$$

To ensure $u^{m+1} \geq g^{m+1}$, we only need to replace $u_n^{m,k+1} = u_n^{m,k} + \omega(y_n^{m,k+1} - u_n^{m,k})$ by $u_n^{m,k+1} = max(u_n^{m,k} + \omega(y_n^{m,k+1} - u_n^{m,k}), g_n^{m+1})$. The rest of the steps are just the same as ordinary **Crank SOR**. The code is listed in the Appendix(i), we only plot the value estimated here:



The $S_f(t)$ in the plot is at around 7.408. Before the free boundary, American put options will be exercised early. After the free boundary, it will be held for future decision.

4.24 Error Examination

For American put options, unfortunately, we do not have an explicit formula for its price like the *Black-Scholes* formula. Therefore, it is hard to compare our numerical solution to the exact value. However, we can still check some significant behaviour of the error.

Note that if we change the size of δx and $\delta\tau$ proportionally, the error should also change proportionally. As **PSOR** is modified from **Crank-Nicolson Method**, we have the order of

error is approximately $O((\delta\tau)^2) + O((\delta x)^2)$. Pick δx_1 and $\delta\tau_1$. Let

$$\delta x_2 = \frac{1}{2}\delta x_1,\ \delta\tau_2 = \frac{1}{2}\delta\tau_1$$
$$\delta x_3 = \frac{1}{4}\delta x_1,\ \delta\tau_3 = \frac{1}{4}\delta\tau_1$$

Then

$$P_1 \approx P + a(\delta\tau_1)^2 + b(\delta x_1)^2$$
$$P_2 \approx P + a(\delta\tau_2)^2 + b(\delta x_2)^2 = P + \frac{1}{4}(a(\delta\tau_1)^2 + b(\delta x_1)^2)$$
$$P_3 \approx P + a(\delta\tau_3)^2 + b(\delta x_3)^2 = P + \frac{1}{16}(a(\delta\tau_1)^2 + b(\delta x_1)^2)$$

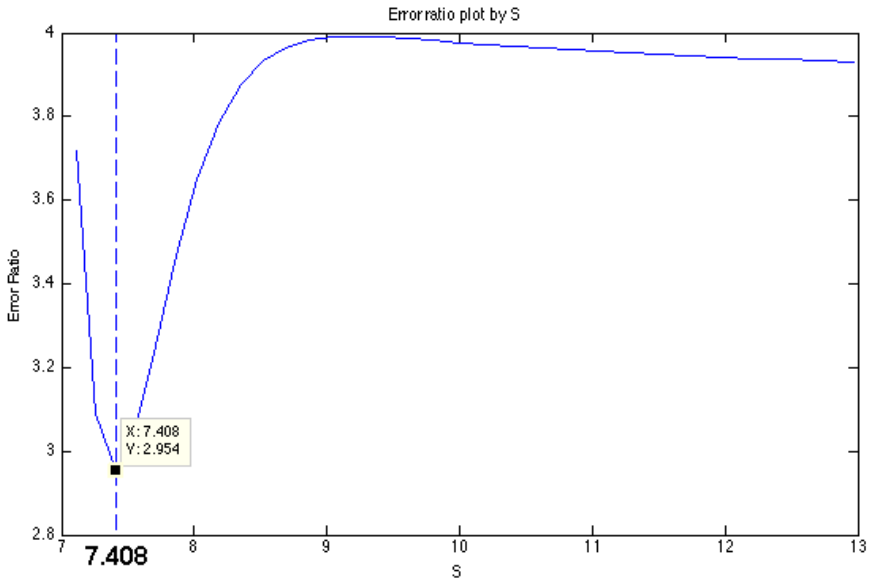
Then by subtraction,

$$P_1 - P_2 = \frac{3}{4}(a(\delta\tau_1)^2 + b(\delta x_1)^2)$$
$$P_2 - P_3 = \frac{3}{16}(a(\delta\tau_1)^2 + b(\delta x_1)^2).$$

Therefore we should have the relationship that:

$$P_1 - P_2 \approx 4(P_2 - P_3).$$

The ratio of $\frac{P_1-P_2}{P_2-P_3}$ is plotted here(taking $\delta\tau_1 = 2e-5$, $\delta x_1 = 0.02$):



Note that only the ratio near $S = 10$ part is of our interest. When S is smaller than the free boundary $S_f(t) = 7.408$, the error ratio randomly pick 0 or NaN. This is because that before free boundary, the estimated values are the same as payoff function. Therefore, in most time the denominator is 0 and the ratio is undefined. As S increases, the estimated value will be convergent to 0 very fast that the approximation algorithm is not deeply used. The ratio will again be uninterpretable.

5 Conclusion

Overall, this project exam different methods for pricing European put options with **PSOR Method** for American put options. For the European put option part, the **Crank-Nicolson Method** provides a more precise estimation. When talking about efficiency, the **LU Method** is much faster than the **SOR Method** because inside **SOR-solver** function need to iterate until convergence at each time step. Back to American options, in practice, the **PSOR Method** computes very slow as it also requires to iterate until convergence. By adjusting over-relaxation parameter, the runtime of computing convergence is optimized.

The underlying assumption for the pricing problem is quite simple. We ignore the effect of transaction fee, dividends, etc. However, these advanced assumptions can be included by further modified based on these algorithms. Moreover, we can also price another type of options by changing the payoff function. These topics are beyond our discussion here.

The major difficulty for this project is to understand the idea how to transform the **Black-Scholes equation** to a numerically solvable problem. I believe the value of this idea is beyond only pricing options. This project may inspire me in the future whenever I need to apply numerical approaches to the encountered problem.

6 Reference

[1] Wilmott, P., Howison, S., Dewynne, J. (1995). The mathematics of financial derivatives: A student introduction. Oxford: Cambridge University Press.

7 Appendix

6(a) Explicit Method

```
function [ values ] = explicit_fd(dx, dt, M, Nminus,Nplus, k )

    nplus = Nplus - Nminus;
    nminus = 0;
    newu = zeros(nplus - nminus + 1, 1);
    b = zeros(nplus - nminus - 1, 1);
    a = dt / (dx*dx);
    fprintf('Result by Explicit. Alpha = %d\n', a);
    x = (Nminus*dx : dx: Nplus*dx)';
    oldu = pay_off(x, k);
    for m = 1:M
        tau = m*dt;
        newu(nminus+1) = u_m_inf(Nminus*dx, tau,k);
        newu(nplus+1) = u_m_inf(Nplus*dx, tau,k);
        n = nminus+2:nplus-1;
        newu(n) = oldu(n) + a*(oldu(n-1)-2*oldu(n)+oldu(n+1));
        oldu = newu;
    end
    values = oldu;
end
```

6(b) LU find y

```
function [ y ] = lu_find_y( a, Nminus, Nplus )

    nplus = Nplus - Nminus;
    nminus = 0;
    asq = a*a;
    y = zeros(nplus - nminus - 1,1);
    y(nminus + 1) = 1 + 2 * a;

    for n = nminus + 2: nplus - 1
        y(n) = 1 + 2*a - asq/y(n-1);
        if y(n) == 0
            error('SINGULA\n');
        end
    end
    fprintf('OK\n')
end
```

6(c) LU solver

```
function [ u ] = lu_solver( b, y, a, Nminus, Nplus )

    nplus = Nplus - Nminus;
    nminus = 0;
    q = zeros(nplus - nminus - 1,1);
    u = zeros(nplus - nminus - 1,1);
    q(nminus + 1) = b(nminus +1);

    for n = nminus + 2:nplus - 1
        q(n) = b(n) + a*q(n-1) / y(n-1);
    end
    u(nplus-1) = q(nplus-1)/y(nplus-1);

    for n = nplus-2:-1:nminus + 1
        u(n) = (q(n) + a*u(n+1)) / y(n);
    end
end
```

6(d) Implicit LU Method

```
function [ values ] = implicit_fd1(dx, dt, M, Nminus,Nplus, k)

    nplus = Nplus - Nminus;
    nminus = 0;
    a = dt / (dx*dx);
    fprintf('Result by LU. Alpha = %d\n', a);
    x =  (Nminus*dx : dx: Nplus*dx)';
```

```
8       values = pay_off(x, k);
9       y = lu_find_y(a, Nminus, Nplus);
10      for m = 1:M
11          tau = m*dt;
12          b = values(nminus+2:nplus);
13          values(nminus + 1) = u_m_inf(Nminus * dx, tau, k);
14          values(nplus + 1) = u_p_inf(Nplus*dx, tau, k);
15          b(nminus +1 ) =b(nminus +1 ) + a*values(nminus+1);
16          b(nplus -1) =b(nplus -1) + a*values(nplus +1);
17          values(nminus +2 : nplus) = lu_solver(b,y,a,Nminus,Nplus);
18      end
19  end
```

6(e) SOR solver

```
1   function [ loops,u ] = SOR_solver( u, b, Nminus, Nplus, a, omega, eps)
2
3       nplus = Nplus - Nminus;
4       nminus = 0;
5       loops = 0;
6       error = eps + 1;
7       while(error > eps)
8           error = 0;
9           for n = nminus + 1:nplus-1
10              y = zeros(nplus - nminus - 1,1);
11              y = (b(n) + a*(u(n-1 + 1)+u(n+1 + 1)))/(1+2*a);
12              y = u(n + 1) + omega*(y - u(n + 1));
13              error =error+ (u(n + 1) - y)^2;
14              u(n + 1) = y;
15              loops = loops +1 ;
16          end
17      end
18
19
20  end
```

6(f) Implicit SOR Method

```
1   function [ values ] = implicit_fd2( dx, dt, M, Nminus,Nplus, k)
2
3       nplus = Nplus - Nminus;
4       nminus = 0;
5       a = dt / (dx*dx);
6       fprintf('Result by SOR. Alpha = %d\n', a);
7       eps = 1.0e-08;
8       omega = 1.0;
9       domega = 0.05;
10      oldloops = 1000;
11      x = (Nminus*dx : dx: Nplus*dx)';
12      values = pay_off(x, k);
13      for m = 1:M
14          tau = m*dt;
15          b = values(nminus+2:nplus);
16          values(nminus + 1) = u_m_inf(Nminus * dx, tau, k);
17          values(nplus + 1) = u_p_inf(Nplus*dx, tau, k);
18          [loops,values] = SOR_solver(values,b,Nminus,Nplus,a,omega,eps);
19          if loops > oldloops
20              domega = domega * (-1);
21              omega = omega+ domega;
22              oldloops = loops;
23          end
24      end
25  end
```

6(g) Crank LU Method

```
1   function [ values ] = implicit_fd2( dx, dt, M, Nminus,Nplus, k)
2
3       nplus = Nplus - Nminus;
4       nminus = 0;
5       a = dt / (dx*dx);
6       fprintf('Result by SOR. Alpha = %d\n', a);
7       eps = 1.0e-08;
8       omega = 1.0;
9       domega = 0.05;
10      oldloops = 1000;
11      x = (Nminus*dx : dx: Nplus*dx)';
```

```
    values = pay_off(x, k);
    for m = 1:M
        tau = m*dt;
        b = values(nminus+2:nplus);
        values(nminus + 1) = u_m_inf(Nminus * dx, tau, k);
        values(nplus + 1) = u_p_inf(Nplus*dx, tau, k);
        [loops,values] = SOR_solver(values,b,Nminus,Nplus,a,omega,eps);
        if loops > oldloops
            domega = domega * (-1);
            omega = omega+ domega;
            oldloops = loops;
        end
    end
end
```

6(h) Crank SOR Method

```
function [ values ] = implicit_fd2( dx, dt, M, Nminus,Nplus, k)

    nplus = Nplus - Nminus;
    nminus = 0;
    a = dt / (dx*dx);
    fprintf('Result by SOR. Alpha = %d\n', a);
    eps = 1.0e-08;
    omega = 1.0;
    domega = 0.05;
    oldloops = 1000;
    x = (Nminus*dx : dx: Nplus*dx)';
    values = pay_off(x, k);
    for m = 1:M
        tau = m*dt;
        b = values(nminus+2:nplus);
        values(nminus + 1) = u_m_inf(Nminus * dx, tau, k);
        values(nplus + 1) = u_p_inf(Nplus*dx, tau, k);
        [loops,values] = SOR_solver(values,b,Nminus,Nplus,a,omega,eps);
        if loops > oldloops
            domega = domega * (-1);
            omega = omega+ domega;
            oldloops = loops;
        end
    end
end
```

6(i) American Method

```
function [ values ] = American( dx, dt, M, Nminus,Nplus, k)

    nplus = Nplus - Nminus;
    nminus = 0;
    a = dt / (dx*dx);
    fprintf('Result by American. Alpha = %d\n', a);
    a2 = a / 2.0;
    eps = 1.0e-32;
    omega = 1.0;
    domega = 0.05;
    oldloops = 1000;
    x =  (Nminus*dx : dx: Nplus*dx)';
    values = g(x,0,k);
    for m = 1:M
        tau = m*dt;
        G = g(x,tau, k);
        b= values(nminus+2:nplus) +a2*(values(nminus+3:nplus+1) -2*values(nminus+2:nplus)+ values(nminus+1:nplus-1));
        values(nminus + 1) = G(nminus+1);
        values(nplus + 1) = G(nplus+1);
        [loops,values] = PSOR_solver(values,b,G,Nminus,Nplus,a,omega,eps);
        if loops > oldloops
            domega = -domega;
             omega = omega+ domega;
         end
         oldloops = loops;
    end
end
```