Numerical Methods For American Option Pricing

Peng Liu

June 2008

Abstract

An analytic solution does not exist for evaluating the American put option. Usually, the value is obtained by applying numerical methods. For instance, the PSOR algorithm is a widely used one in financial industry. In the past few years, many other methods to solve American option problems have been introduced, two examples are Linear Programming and Penalty method. The aims of this dissertation are: first, to provide an introduction to four algorithms - Explicit, PSOR, Penalty and Linear Programming on pricing American put options; and second, to make comparisons through numerical tests.

Acknowledgments
Sincere gratitude to my supervisor, Dr. Christoph Reisinger, for his professional guidance and encouragement throughout this project.

1 Introduction

Options are financial instruments that give the holder the right, but not obligation, to buy or sell an asset at a specified price(strike price) at some future time. A European option can only be exercised at maturity time. An American option has the feature that the holder can exercise at any time before maturity. By now, there is no analytic representation for the value of American put options. Their values are usually obtained by numerical methods.
We start with Black-Schole model with constant volatility for option pricing.

$$dS = \mu S dt + \sigma S dW$$

$$\frac{\partial u}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 u}{\partial S^2} + rS\frac{\partial u}{\partial S} - ru = 0$$

In order to apply numerical analysis method to price the option backward, We write $\tau = T - t$. Then the Black-Schole formula becomes,

$$\frac{\partial u}{\partial \tau} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 u}{\partial S^2} + rS\frac{\partial u}{\partial S} - ru$$

Consider an American put option, which can be exercised at any time up to expiry T. It is not possible for the option price below the payoff value, hence we must have

$$u(S,t) \geq g(S)$$

where $g(S) = K - S$, S is stock price, K is the strike price, and $g(S)$ is the payoff function.
In fact, the value of American option is always greater than or equal to European option and payoff value, as illustrated in Figure 1.1.

At every point of time, the American option holder is confronted with the decision that either exercising or holding the option, so

$$\mathcal{L}u = \frac{\partial u}{\partial \tau} - \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 u}{\partial S^2} - rS\frac{\partial u}{\partial S} + ru = 0 \quad \vee \quad u(S,t) = g(S) \tag{1.1}$$

Because if it is optimal to exercise, the return on hedging portfolio can be at most risk-free interest rate, this leads the following inequality,

$$\mathcal{L}u = \frac{\partial u}{\partial \tau} - \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 u}{\partial S^2} - rS\frac{\partial u}{\partial S} + ru \geq 0 \tag{1.2}$$

Combining these conditions, we get the *linear complementarity problem*(LCP) for American put option.

$$\begin{aligned} \mathcal{L}u &\geq 0 \\ u &\geq g \\ \mathcal{L}u = 0 \vee (u - g) &= 0 \end{aligned} \tag{1.3}$$

where the notation $\mathcal{L}u = 0 \vee (u - g) = 0$ denotes that either $\mathcal{L}u = 0$ or $(u - g) = 0$ everywhere in the solution domain.
The boundary condition associated with American put option is

$$u(S,\tau) = 0, \ as \ S \to \infty \tag{1.4}$$

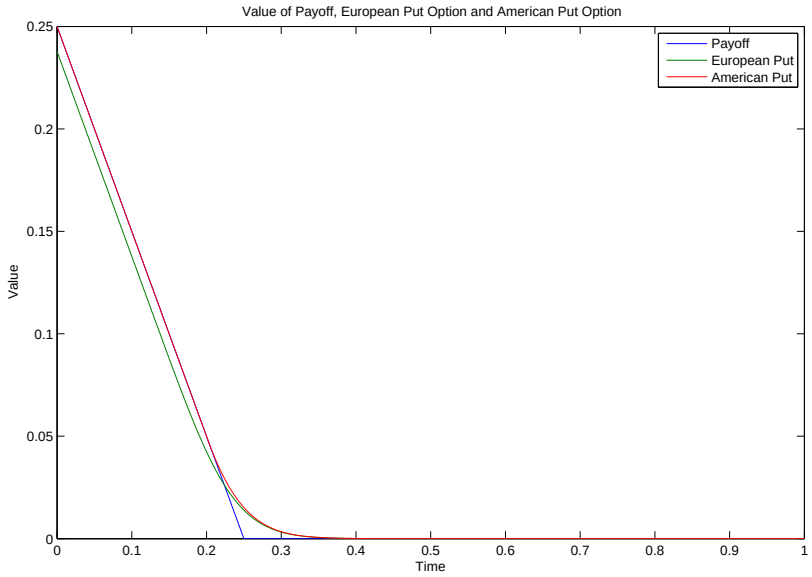


Figure 1.1: Payoff, European put and American put with strike=0.25, $\sigma = 0.2$, $r = 0.05$

This dissertation is structured as follows: In Section One we already introduced Black-Schole equation with free boundary conditions, that leads to linear complementarity problem. Section Two provides an introduction on finite different method. Section Three to Six present the algorithm of Explicit, PSOR, Penalty and Linear Programming respectively. The numerical results regarding accuracy and time costs are shown in section 7.

2 Finite Difference Discretisation

Applying finite difference discretisation to the backward Black-Schole equation (1.2), we get

$$\frac{u_i^m - u_i^{m-1}}{\Delta\tau} = \theta(\frac{1}{2}\sigma^2 S_i^2 \frac{u_{i+1}^m - 2u_i^m + u_{i-1}^m}{(\Delta S)^2} + rS_i \frac{u_{i+1}^m - u_{i-1}^m}{2\Delta S} - ru_i^m)$$
$$+(1-\theta)(\frac{1}{2}\sigma^2 S_i^2 \frac{u_{i+1}^{m-1} - 2u_i^{m-1} + u_{i-1}^{m-1}}{(\Delta S)^2} + rS_i \frac{u_{i+1}^{m-1} - u_{i-1}^{m-1}}{2\Delta S} - ru_i^{m-1}) \quad (2.1)$$

Where θ is the weight parameter, it gives explicit scheme with $\theta = 0$, implicit scheme with $\theta = 1$, and Crank-Nicolson method with $\theta = 0.5$.
Explicit and implicit schemes have first-order accurate in $\Delta\tau$ and second-order accurate in ΔS, while Crank-Nicolson gives both second-order accurate in $\Delta\tau$ and ΔS.
Rearranging equation (2.1), it follows that

$$a_i^m u_{i-1}^m + b_i^m u_i^m + c_i^m u_{i+1}^m = A_i^m u_{i-1}^{m-1} + B_i^m u_i^{m-1} + C_i^m u_{i+1}^{m-1} \quad (2.2)$$

where,

$$\begin{aligned} a_i^m &= -\tfrac{1}{2}\theta\Delta\tau(\sigma^2 S_i^2 \tfrac{1}{(\Delta S)^2} - rS_i \tfrac{1}{\Delta S}) \\ b_i^m &= 1 + \theta\Delta\tau(\sigma^2 S_i^2 \tfrac{1}{(\Delta S)^2} + r) \\ c_i^m &= -\tfrac{1}{2}\theta\Delta\tau(\sigma^2 S_i^2 \tfrac{1}{(\Delta S)^2} + rS_i \tfrac{1}{\Delta S}) \end{aligned}$$

and

$$\begin{aligned} A_i^m &= \tfrac{1}{2}(1-\theta)\Delta\tau(\sigma^2 S_i^2 \tfrac{1}{(\Delta S)^2} - rS_i \tfrac{1}{\Delta S}) \\ B_i^m &= 1 - (1-\theta)\Delta\tau(\sigma^2 S_i^2 \tfrac{1}{(\Delta S)^2} + r) \\ C_i^m &= \tfrac{1}{2}(1-\theta)\Delta\tau(\sigma^2 S_i^2 \tfrac{1}{(\Delta S)^2} + rS_i \tfrac{1}{\Delta S}) \end{aligned}$$

The boundary condition (1.4) gives

$$u_N^m = u_N^{m-1} = 0 \quad (2.3)$$

It is often attempting to write (2.2) in matrix form

$$\begin{bmatrix} b_0 & c_0 & 0 & \cdots & 0 \\ a_1 & b_1 & c_1 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & a_{N-1} & b_{N-1} & c_{N-1} \\ 0 & \cdots & 0 & a_N & b_N \end{bmatrix} \begin{bmatrix} u_0^m \\ u_1^m \\ \vdots \\ u_{N-1}^m \\ u_N^m \end{bmatrix} = \begin{bmatrix} B_0 & C_0 & 0 & \cdots & 0 \\ A_1 & B_1 & C_1 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & A_{N-1} & B_{N-1} & C_{N-1} \\ 0 & \cdots & 0 & A_N & B_N \end{bmatrix} \begin{bmatrix} u_0^{m-1} \\ u_1^{m-1} \\ \vdots \\ u_{N-1}^{m-1} \\ u_N^{m-1} \end{bmatrix}$$

According to boundary conditions, the value of boundary parameters can be set as following.

$$b_N = 1,\ a_N = 0 \tag{2.4}$$
$$B_N = 1,\ A_N = 0 \tag{2.5}$$

We can represent the whole matrix equation in short form.

$$M_1 u^m = M_2 u^{m-1} \tag{2.6}$$

The corresponding discrete linear complementarity problem becomes

$$\begin{aligned} M_1 u^m - M_2 u^{m-1} &\geq 0 \\ u^m &\geq g \\ (M_1 u^m - M_2 u^{m-1}) \cdot (u^m - g) &= 0 \end{aligned} \tag{2.7}$$

Where g is value of payoff, the inequalities $\geq$ and multiplication $\cdot$ are understood to be elementwise.

3 Explicit Method

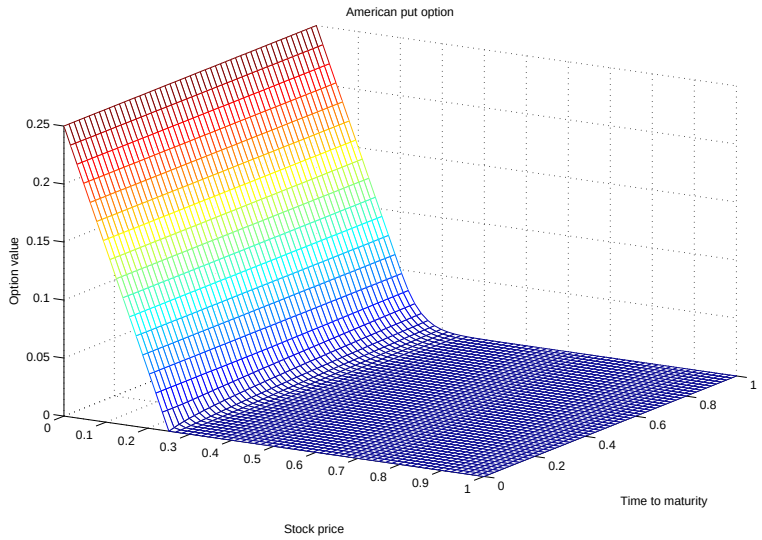


Figure 3.1: American put option solution surface with strike=0.25, $\sigma = 0.2$, $r = 0.05$

Figure (3.1) represents the solution surface of an American put option. The S-axis and τ-axis have been discretized. The option values are computed stepwise from $\tau = 0$ to 1.
The algorithm of *explicit method* is as its name suggested, the LCP constraints are handled *explicitly* in each time step calculation.

At time step m, we first compute the option value $u^{(m)} = (u_0^{(m)}, u_1^{(m)}, \ldots, u_N^{(m)})$ from BSPDE

$$\mathcal{L}u = 0$$

Notice that if $u_i^{(m)} < g_i$, where g_i is the payoff value at price node i, then LCP constraints (1.3) are violated at this point. In this case, we force $u_i^{(m)} = g_i$ to make it satisfy the constraints. In summary, the algorithm of Explicit Method can be presented as follows.

Explicit Method for Computing American Put Option

Initialization:

financial variables (K, T, σ, r) as needed.
Grid variable :S_{max}, S_{min}, e.g, $S_{min} = 0$,
M as maximum of time steps, N as maximum of nodes,
Δt, ΔS.
algorithm variables : $\theta \in [0, 1]$, e.g. $\theta = \frac{1}{2}$
Set up matrix M_1, M_2, as in (2.6) together with corresponding boundary conditions (2.4), (2.5).

Core Calculations

Initial iteration vector
$u^{(0)} := (g(S_0), g(S_1), \ldots, g(S_N))^T$,
where $g(\cdot)$ is the payoff function at maturity.
Time loop: for $m = 1, 2, \ldots, M$
    calculate $u^{(m)} = M_1^{-1} M_2 u^{m-1}$
    $u^{(m)} = max\{u^{(m)}, g\}$
end for
final result: $u^{(M)}$

Remark:

- $u^{(m)} = max\{u^{(m)}, g\}$ is computed elementwise.
- In MATLAB, we can use the operation of $M_1\backslash$ instead of computing inverse of M_1, as this would be much faster.

4 Projected SOR Method

4.1 The Jacobi and Gauss-Seidel Methods

Now we are going to seek the solution of the following equation for unknown vector v.

$$Av = f \tag{4.1}$$

where $A \in \mathbb{R}^{n\times n}$, $v \in \mathbb{R}^n$, and $f \in \mathbb{R}^n$.
There are, in fact, many ways to solve the above linear equations. In our context, we consider iteration methods, which requires less PC memory and computational cost compared to ordinary elimination schemes.
We choose a suitable regular matrix $M \in \mathbb{R}^{n\times n}$, such that

$$\begin{aligned} Mv &= Mv - Av + f \\ &= (M - A)v + f \\ \Longrightarrow v &= (I - M^{-1}A)v + M^{-1}f \end{aligned}$$

Let $G := (I - M^{-1}A)$, $d := M^{-1}f$, the above equation can be written in iteration form

$$v^{(k+1)} = Gv^{(k)} + d$$

The solution v is fixed,

$$v = Gv + d$$

Now we can analyze the error term

$$e^{(k+1)} = v - v^{(k+1)} = G(v - v^{(k)}) = Ge^{(k)}$$

The iteration is convergent, if

$$\lim_{k\to\infty} e^{(k)} = 0$$

for all $e^{(k)}$, that is

$$|\rho(G)| < 1$$

where $\rho(G)$ is the spectral radius of G.

Classical relaxation methods use an additive splitting as in

$$A = L + D + U$$

where L and U are strictly lower and upper part of A respectively, D is diagonal.

$$L = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ a_{21} & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \\ a_{n-1,1} & \cdots & a_{n-1,n-2} & 0 & 0 \\ a_{n,1} & \cdots & a_{n,n-2} & a_{n,n-1} & 0 \end{bmatrix}, U = \begin{bmatrix} 0 & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & 0 & a_{23} & \cdots & a_{2n} \\ 0 & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & a_{n-1,n} \\ 0 & \cdots & 0 & 0 & 0 \end{bmatrix},$$

$$D = \begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & a_{n-1,n-1} & 0 \\ 0 & \cdots & 0 & 0 & a_{nn} \end{bmatrix}$$

We can solve (4.1) by the following iterative algorithm.
Jacobi method

$$\begin{aligned} Dv^{(k+1)} &= -(U+L)v^{(k)} + f \\ or \quad v^{(k+1)} &= -D^{-1}(U+L)v^{(k)} + D^{-1}f \end{aligned}$$

Gauss-Seidel method

$$\begin{aligned} (D+L)v^{(k+1)} &= -Uv^{(k)} + f \\ or \quad v^{(k+1)} &= -(D+L)^{-1}Uv^{(k)} + (D+L)^{-1}f \end{aligned} \tag{4.2}$$

Suitable stopping criteria are

$$\| v^{(k+1)} - v^{(k)} \| \leq \epsilon$$

The characteristic equation for Jacobi method is

$$\begin{aligned} &det[\lambda I - D^{-1}(U+L)] = 0 \\ \Longrightarrow &det[\lambda D - L - U] = 0 \end{aligned} \tag{4.3}$$

And for Gauss-Seidel

$$\begin{aligned} &det[\lambda I - (D-L)^{-1}U] = 0 \\ \Longrightarrow &det[\lambda(D-L) - U] = 0 \end{aligned}$$

Gauss-Seidel algorithm (4.2) can be written as component form

$$a_{ii}v_i^{(k+1)} + \sum_{j=1}^{i-1} a_{ij}v_j^{(k+1)} = -\sum_{j=i+1}^{n} a_{ij}v_j^{(k)} + f_i$$

Rearrange it, gives

$$v_i^{(k+1)} = -\frac{1}{a_{ii}}\sum_{j=1}^{i-1} a_{ij}v_j^{(k+1)} - \frac{1}{a_{ii}}\sum_{j=i+1}^{n} a_{ij}v_j^{(k)} + \frac{1}{a_{ii}}f_i \tag{4.4}$$

In practical, when we calculate $v_i^{(k+1)}$ on the LHS, $v_j^{(k+1)}, j = 1, \ldots, i-1$ on RHS is already known. This represents the "up-to-date" algorithm, where the latest information of v_i can be used in each step to calculate v_{i+1}.
(4.4) can also be represented in matrix form

$$v^{(k+1)} = D^{-1}Lv^{(k+1)} + D^{-1}Uv^{(k)} + D^{-1}f \tag{4.5}$$

Theorem 4.1 [1] *The Gauss-Seidel method converges to the solution of $Av = f$ if*

$$r = \max_i \sum_{j=1, j\neq i}^{n} \left|\frac{a_{ij}}{a_{ii}}\right| < 1$$

where a_{ij}'s are entries of matrix A. Equivalently, A is strictly diagonally dominant.

4.2 Successive Over Relaxation(SOR) Method

Subtract $v^{(k)}$ on both side of equation (4.5), get

$$v^{(k+1)} - v^{(k)} = D^{-1}Lv^{(k+1)} + D^{-1}Uv^{(k)} + D^{-1}f - v^{(k)} \tag{4.6}$$

Hence Gaussian-Seidel method can be written as

$$v^{(k+1)} = v^{(k)} + (v^{(k+1)} - v^{(k)})$$

[1] The proof can be found in [7]

where $v^{(k+1)} - v^{(k)}$ is given by (4.6).

We can regard Gaussian-Seidel algorithm as adding an increment $v^{(k+1)} - v^{(k)}$ to $v^{(k)}$, until the difference between $v^{(k)}$ and $v^{(k+1)}$ is small enough. If we adjust the increment by multiplying a parameter ω, a better rate of convergent may be obtained. This gives Successive Over Relaxation(SOR) algorithm.

$$\begin{aligned} v^{(k+1)} &= v^{(k)} + \omega(v^{(k+1)} - v^{(k)}) \\ &= v^{(k)} + \omega(D^{-1}Lv^{(k+1)} + D^{-1}Uv^{(k)} + D^{-1}f - v^{(k)}) \\ &= (1-\omega)v^{(k)} + \omega(D^{-1}Lv^{(k+1)} + D^{-1}Uv^{(k)} + D^{-1}f) \end{aligned} \quad (4.7)$$

The component form of SOR method is useful in practice, it can be derived directly as follows

$$v_i^{(k+1)} = (1-\omega)v_i^{(k)} + \omega\left(-\sum_{j=1}^{i-1}\frac{a_{ij}}{a_{ii}}v_j^{(k+1)} - \sum_{j=i+1}^{n}\frac{a_{ij}}{a_{ii}}v_j^{(k)} + \frac{1}{a_{ii}}f_i\right) \quad (4.8)$$

Working through $i = 1, 2, \ldots, n$, the value of v can be found without any computation of matrix inverse.
Moving all $v^{(k+1)}$ terms of (4.7) to the LHS, gives

$$(I - \omega D^{-1}L)v^{(k+1)} = ((1-\omega)I + \omega D^{-1}U)v^{(k)} + \omega D^{-1}f$$

So that the iteration matrix of SOR is

$$G_{SOR\omega} = (I - \omega D^{-1}L)^{-1}((1-\omega)I + \omega D^{-1}U)$$

The SOR algorithm converges if

$$|\rho(G_{SOR\omega})| < 1, \quad for\ \omega \in (0, 2)$$

The characteristic equation is as follows

$$\begin{aligned} det[\lambda I - (I - \omega D^{-1}L)^{-1}((1-\omega)I + \omega D^{-1}U)] = 0 \\ \Longrightarrow det[(\lambda - 1 + \omega)D - \lambda\omega L - \omega U] = 0 \end{aligned} \quad (4.9)$$

4.3 Optimum ω

In SOR algorithm, the value of ω influences the speed of convergent. To optimize ω, here we restrict ourselves to the case that A is a block-tridiagonal matrix. A tridiagonal matrix is a matrix who has nonzero elements only in the main diagonal, the first diagonal below the main diagonal, and the first diagonal above it. That is, A takes the following form.

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ 0 & \cdots & 0 & a_{n,n-1} & a_{nn} \end{bmatrix}$$

It turns out that the matrix we concern, which is M_1 in (2.6), has precisely the same form.

The first step to optimize ω is to scale the rows and columns of characteristic function for SOR, so that the off-diagonal elements equal to those for Jacobi case.
Note that SOR characteristic polynomial is given by (4.9). Defining $B = diag[b_i]$, and $C = diag[c_i]$, we multiply matrix B and C to SOR characteristic polynomial on LHS and RHS respectively.

$$\begin{aligned} & det(B[(\lambda_S - 1 + \omega)D - \lambda_S \omega L - \omega U]C) \\ = & det[(\lambda_S - 1 + \omega)BDC - \lambda_S \omega BLC - \omega BUC] \end{aligned}$$

The corresponding Jacobi characteristic polynomial is given by (4.3).

$$det[\lambda_J D - L - U]$$

To make them equal, we must choose B and C, such that

$$\begin{aligned} \lambda_S \omega BLC &= L \\ \omega BUC &= U \\ (\lambda_S - 1 + \omega)BDC &= \lambda_J D \end{aligned}$$

Represent them in component form

$$\begin{aligned}
\lambda_S \omega b_i a_{i,i-1} c_{i-1} &= a_{i,i-1}, & i &= 2, 3, \ldots, n \\
\omega b_i a_{i,i+1} c_{i+1} &= a_{i,i+1}, & i &= 1, 2, \ldots, n-1 \\
(\lambda_S - 1 + \omega) b_i a_{ii} c_i &= \lambda_J a_{ii}, & i &= 1, 2, \ldots, n
\end{aligned}$$

These equations can be simplified to

$$\begin{aligned}
b_i c_{i-1} &= \frac{1}{\lambda_S \omega} \\
b_i c_{i+1} &= \frac{1}{\omega} \\
b_i c_i &= \frac{\lambda_J}{\lambda_S - 1 + \omega}
\end{aligned}$$

Dividing the first and second equations by the last one, get

$$\begin{aligned}
\frac{c_{i-1}}{c_i} &= \frac{\lambda_S - 1 + \omega}{\lambda_S \omega \lambda_J} \\
\frac{c_i}{c_{i+1}} &= \frac{\omega \lambda_J}{\lambda_S - 1 + \omega}
\end{aligned}$$

Since the above equations hold for all $i = 2, 3, \ldots, n-1$, so we must have

$$\frac{\lambda_S \omega \lambda_J}{\lambda_S - 1 + \omega} = \frac{\lambda_S - 1 + \omega}{\omega \lambda_J}$$

Rearrange it, gives

$$\lambda_S \omega^2 \lambda_J^2 = (\lambda_S - 1 + \omega)^2 \tag{4.10}$$

This is known as Young's formula. In particular, setting $\omega = 1$, we obtain $\lambda_S = \lambda_J^2$. So that

$$\rho(G_{GS}) = \rho(G_{Jac})^2$$

i.e. if $\rho(G_{Jac}) < 1$, the Gauss-Seidel method converges twice as fast as the Jacobi method. Note that (4.10) can be written in terms of $\sqrt{\lambda_S}$

$$(\sqrt{\lambda_S})^2 - \omega \lambda_J \sqrt{\lambda_S} + \omega - 1 = 0 \tag{4.11}$$

This can be considered as a quadratic equation of $\sqrt{\lambda_S}$. The product of the two roots equals to $\omega - 1$. In order to have convergence of SOR, we require that $\sqrt{\lambda_S} < 1$, hence we must have $|\omega - 1| < 1$ or $0 < \omega < 2$. Solving (4.11), we can obtain

$$\sqrt{\lambda_S} = \frac{\lambda_J}{2}\left(\omega \pm \sqrt{(\omega - d_1)(\omega - d_2)}\right)$$

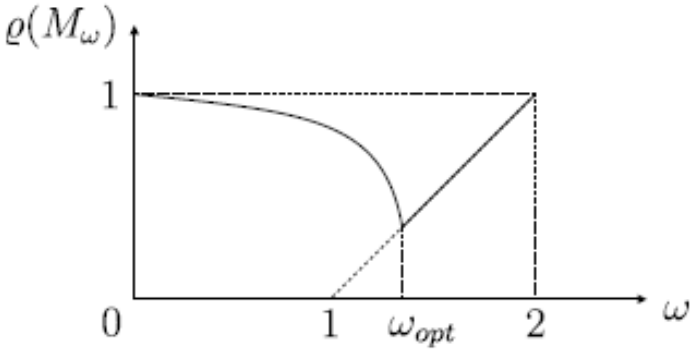
where

$$d_1 = \frac{2}{1 + \sqrt{1 - \lambda_J^2}},\ d_2 = \frac{2}{1 - \sqrt{1 - \lambda_J^2}}$$

λ_S is real if $0 \leq \omega \leq d_1$, and it can be shown that

$$\frac{d|\sqrt{\lambda_S}|}{d\omega} = \frac{|\lambda_J|}{2}\left(1 - \sqrt{1 + \frac{(d_2 - d_1)^2}{4(\omega - d_1)(\omega - d_2)}}\right) < 0$$

so that $|\sqrt{\lambda_S}|$ is a decreasing function of ω, as shown in the figure[2] below.



Obviously, $|\lambda|$ is minimal when $\omega = d_1$. Therefore,

$$\omega_{opt} = \frac{2}{1 + \sqrt{1 - \rho(G_{Jac})^2}}$$

In practice, the computation of ω_{opt} is costly. However, in many cases, $\omega \approx 1.3$ gives a good initial choice.

[2]comes from [8]

4.4 The Projected SOR Method

Recall that the initial problem we were dealing with is discrete LCP (2.7) for American put option.

$$\begin{aligned} M_1 u^m - M_2 u^{m-1} &\geq 0 \\ u^m &\geq g \\ (M_1 u^m - M_2 u^{m-1}) \cdot (u^m - g) &= 0 \end{aligned}$$

The above problem equivalent to

$$\min\{M_1 u^m - M_2 u^{m-1}, u^m - g\} = 0$$

Defining $A := M_1$, $f := M_2 u^{m-1}$, and $v := u^{(m)}$ it follows that

$$\begin{aligned} & \min_v\{v - A^{-1} f, v - g\} = 0 \\ \iff \quad v &= \max\{A^{-1} f, g\} \end{aligned} \tag{4.12}$$

This problem can be solved by *Projected SOR*(PSOR) which is suggested by Cryer(1971). Its idea is to apply maximization constraint (4.12) to SOR algorithm (4.8).
For $i = 1, 2, \ldots, n$:

$$t_i^{(k+1)} = \frac{1}{a_{ii}} \left(-\sum_{j=1}^{i-1} a_{ij} v_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} v_j^{(k)} + a_{ii} f_i \right) \tag{4.13}$$

$$v_i^{(k+1)} = \max\{v_i^{(k)} + \omega(t_i^{(k+1)} - v_i^{(k)}), g\} \tag{4.14}$$

Having given all necessary details about PSOR method, now we can present the algorithm for computing American put option.

PSOR Algorithm to Compute American Put Option

Initialization:

financial variables (K, T, σ, r) as needed.
Grid variable :S_{max}, S_{min}, e.g, $S_{min} = 0$,
M as maximum of time steps, N as maximum of nodes,
Δt, ΔS.
algorithm variables : $\theta \in [0, 1]$,$\omega \in [1, 2)$, e.g. $\theta = \frac{1}{2}$, $\omega = 1.3$
PSOR Convergence error tolerance, e.g. $\epsilon = 10^{-10}$
Set up matrix M_1, M_2, as in (2.6) together with corresponding boundary conditions.

Core Calculations

Initial iteration vector
$w^{(0)} := (g(S_0), g(S_1), \ldots, g(S_N))^T$,
where $g(\cdot)$ is the payoff function at maturity.
Time loop: for $m = 1, 2, \ldots, M$
    calculate $b^{(m)} = M_2 w^{(m-1)}$ as in (2.6)
    set $v^{(0)} := max\{M_1^{-1} b^{(m)}, g(S)\}$
    PSOR loop: while $\| v^{(k+1)} - v^{(k)} \| / \sqrt{N} > \epsilon$ do
        (4.13), (4.14) componentwise
    end while
    $w^{(m)} = v^{(k+1)}$
end for
final result: $V := w^{(M)}$

5 Penalty Method

The Penalty Method for American option pricing discussed below is proposed by P.A.Forsyth and K.R.Vetzal(2002)[2].
The idea of penalty method is to "penalize" the violation by setting a "penalize" parameter ρ in the BSPDE.

$$\frac{\partial u}{\partial \tau} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 u}{\partial S^2} + rS\frac{\partial u}{\partial S} - ru + \rho\ max(g-u) \tag{5.1}$$

or

$$\mathcal{L}u = \rho\ max(g-u)$$

If $\rho \to \infty$, then the LCP constraints in (1.3) can be satisfied.
To discretize the above equation, it is straightforward that we add a penalty term to (2.1).

$$\begin{aligned} u_i^m - u_i^{m-1} = \theta\Delta\tau(\frac{1}{2}\sigma^2 S_i^2 \frac{u_{i+1}^m - 2u_i^m + u_{i-1}^m}{(\Delta S)^2} + rS_i\frac{u_{i+1}^m - u_{i-1}^m}{2\Delta S} - ru_i^m) \\ +(1-\theta)\Delta\tau(\frac{1}{2}\sigma^2 S_i^2 \frac{u_{i+1}^{m-1} - 2u_i^{m-1} + u_{i-1}^{m-1}}{(\Delta S)^2} + rS_i\frac{u_{i+1}^{m-1} - u_{i-1}^{m-1}}{2\Delta S} - ru_i^{m-1}) \\ +P_i^m(g_i - u_i^m) \end{aligned} \tag{5.2}$$

where

$$P_i^m = \begin{cases} \rho, & if\ u_i^m < g_i \\ 0, & otherwise \end{cases}$$

Again, we need to write the penalty term in matrix form. Let $\bar{P}$ be a diagonal matrix, such that

$$\bar{P}(u^m)_{ij} = \begin{cases} \rho & if\ u_i^m < g_i\ and\ i = j \\ 0 & otherwise \end{cases} \tag{5.3}$$

The the matrix form of equation (5.2) is

$$M_1 u^m = M_2 u^{m-1} + [\bar{P}(u^m)](g - u^m)$$

Where M_1 and M_2 are the same as that in (2.6). Rearrange it, gives

$$[M_1 + \bar{P}(u^m)]u^m = M_2 u^{m-1} + [\bar{P}(u^m)]g \tag{5.4}$$

After the discretization work, we will now seek the solution of LCP equation. The non-linear discrete equation (5.4) can be solved by generalized Newton iteration. Note that equation (5.4) can be written as

$$F(u^m) = 0 \tag{5.5}$$

where

$$F(u^m) := [M_1 + \bar{P}^m]u^m - M_2 u^{m-1} - [\bar{P}^m]g$$

and

$$\bar{P}^m := \bar{P}(u^m)$$

We define the derivative of the penalty term

$$\frac{\partial \bar{P}_i^m (g_i - u_i^m)}{\partial u_i^m} = \begin{cases} -\rho & if\ u_i^m < g_i \\ 0 & otherwise \end{cases}$$

Thus

$$F'(u^m) = M_1 + \bar{P}^m$$

Apply generalized Newton iteration method to (5.5)

$$\begin{aligned} [u^m]^{k+1} &= [u^m]^k - [F'(u^m)]^{-1}[F(u^m)] \\ &= [u^m]^k - [M_1 + \bar{P}^m]^{-1}[(M_1 + \bar{P}^m)[u^m]^k \\ &\quad -M_2 u^{m-1} - \bar{P}^m g] \\ &= [u^m]^k - [u^m]^k + [M_1 + \bar{P}^m]^{-1}[M_2 u^{m-1} + \bar{P}^m g] \\ &= [M_1 + \bar{P}^m]^{-1}[M_2 u^{m-1} + \bar{P}^m g] \end{aligned}$$

Hence we can summarize the algorithm as follows.

Penalty algorithm to compute American put option

Initialization:

The same as Explicit method.

Core Calculations

Initialize $u^0 := (g(S_0), g(S_1), \ldots, g(S_N))^T$

for m=1 to M % M is the max time steps

  $U = M_2 * u^{m-1}$

  $\bar{P}_0 = \bar{P}(u^{m-1})$ % $\bar{P}(\cdot)$ is defined at (5.3)

  for k=0, 1, ... until convergence

    $u^m_{k+1} = (M_1 + \bar{P}_k)^{-1} * (U + \bar{P}_k * g)$

    $\bar{P}_{k+1} = \bar{P}(u^m_{k+1})$

    if $[\max_i \frac{|(u^m_{k+1})_i - (u^m_k)_i|}{\max(|1, (u^m_{k+1})_i|)} < tol]$ or $\bar{P}_{k+1} = \bar{P}_k$

      quit % tol is the error tolerance

  end for

end for

final result: $u^{(M)}$

6 Linear Programming Method

A.Borici and H.J.Lüthi presented this algorithm for American option pricing in [6].

6.1 Linear Programming(LP)

Recall that we are dealing with the discrete linear complementary problem (2.7) with appropriate boundary conditions.

$$\begin{aligned} M_1 u^m - M_2 u^{m-1} &\geq 0 \\ u^m &\geq g \\ (M_1 u^m - M_2 u^{m-1}) \cdot (u^m - g) &= 0 \end{aligned}$$

where $m = 1, 2, \ldots, M$ are time steps in finite difference discretisation, $M_1, M_2 \in \mathbb{R}^{(N+1)\times(N+1)}$, $N+1$ is the maximum number of nodes.
Let $v^m := u^m - g$ be the excess value vector, and $s^m := M_1 u^m - M_2 u^{m-1}$ be the slack vector.
Then the LCP can be written as follows

$$\begin{aligned} M_1 v^m - s^m &= b^m \\ v^m \geq 0, \quad s^m \geq 0, \quad (s^m)^T v^m &\geq 0 \end{aligned} \tag{6.1}$$

where

$$\begin{aligned} b^m &= M_1 v^m - s^m \\ &= M_1(u^m - g) - (M_1 u^m - M_2 u^{m-1}) \\ &= -M_1 g + M_2 u^{m-1} \\ &= (M_2 - M_1) g + M_2 v^{m-1} \end{aligned}$$

We can also write $b^m = b^0 + M_2 v^{m-1}$ with $b^0 = (M_2 - M_1)g$.
It is easy to verify the boundary condition of v is

$$v^0 = 0$$

Definition 1 *A square matrix is called Z-matrix if all its off-diagonal elements are less or equal to zero.*

Obviously, matrix M_1 is Z-matrix if and only if $a_i^m \leq 0$ and $c_i^m \leq 0$, which is the case

$$\left| r - \frac{\sigma^2}{2} \right| \leq \frac{\sigma^2}{\Delta S}$$

This condition holds if we take ΔS small enough. In realistic, the condition is easy to be satisfied since the critical value for ΔS is very large.

Suppose M_1 is a Z-matrix, the following statements are equivalent(see [5][Fiedler and Ptak, 1962]):

- M_1 is a P-matrix.
- $M_1^{-1} \geq 0$.
- There is $x \geq 0$, s.t. $M_1 x > 0$ has a solution.

We will use these properties to design the algorithm for solving LCP.
According to [Dempster and Hutton, 1999], if M_1 is a Z-matrix and v^{m-1} is know in step m, problem 6.1 can be solved by the sequence of *linear programming*(LP) as follows.

$$\begin{aligned} For \quad & m = 1, 2, \ldots, M : \\ & min \ c^T v^m, \quad s.t. \\ & M_1 v^m - s^m = b^m \ \ and \\ & v^m \geq 0, \ s^m \geq 0 \end{aligned} \tag{6.2}$$

where $c \in \mathbb{R}^{N+1}$ is an arbitrary positive vector.

6.2 Algorithm to Solve LCP

Theorem 6.1 *The solution of linear complementary problem (*6.1*) is unique and the complementary feasible bases have the following structure.[6]*

$$\bar{v}_{nb}^m = (s_1^m, \ldots, s_{nb-1}^m, v_{nb}^m, \ldots, v_{N+1}^m)$$

where $nb \in \{1, 2, \ldots, N+1\}$.

(The proof can be found in [6])
The algorithm can be expressed in terms of following partitions of LCP (6.1), with fixed time step m and node nb.

$$\begin{bmatrix} A_{11} & a_{nb-2,nb-1} & 0 \\ a_{nb-1,nb-2} & a_{nb-1,nb-1} & a_{nb-1,nb} \\ 0 & a_{nb,nb-1} & A_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ v_3 \end{bmatrix} - \begin{bmatrix} s_1 \\ s_2 \\ 0 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{6.3}$$

- Time index m has been omitted for simplicity.
- A_{11} and A_{33} and all a's belong to square matrix M_1, where $M_1 \in \mathbb{R}^{(N+1)\times(N+1)}$.
- A_{33} is a block with dimension $(N+2-nb)\times(N+2-nb)$. A_{11} is a block with dimension $(nb-2)\times(nb-2)$.
- a's are scalar elements in M_1. Zero spaces of M_1 are omitted.
- s_1 and b_1 are vectors with dimension $(nb-2)\times 1$. v_3 and $b3$ are also vectors which has dimension $(N+2-nb)\times 1$. s_2 and b_2 are scalars.

Lemma[6] For each time step there is a partition of the above form such that:

$$b_1 < 0,\ b_2 < 0,\ A_{33}^{-1} b_3 \geq 0 \tag{6.4}$$

for small enough ΔS. (The proof can be found in [6].)

In a fixed time step m, suppose we find a partition (6.3) with certain index nb, and property (6.4) was satisfied. It follows that

$$\begin{aligned} s_1 &= -b_1 > 0 \\ s_2 &= a_{nb-1,nb} e_1^T v_3 - b_2 \qquad (6.5) \\ v_3 &= A_{33}^{-1} b_3 \geq 0 \qquad (6.6) \end{aligned}$$

Note that if $s_2 \geq 0$ then LCP (6.1) is already solved in this time step m. Otherwise we decrease nodes index nb by one unit, and we have

$$b_1^{new} < 0,\ b_2^{new} = b_1^{old} < 0,\ b_3^{new} = \begin{pmatrix} b_2^{old} \\ b_3^{old} \end{pmatrix}$$

The new complementary basic solution becomes

$$\begin{aligned} s_1^{new} &= -b_1^{new} \\ s_2^{new} &= a_{nb-1,nb} z - b_2^{new} \qquad (6.7) \\ v_3^{new} &= \begin{pmatrix} z \\ v_3^{old} + z a_{nb,nb-1} A_{33}^{-1} e_1 \end{pmatrix} \qquad (6.8) \end{aligned}$$

where

$$z = \frac{-s_2^{old}}{a_{nb-1,nb-1} - a_{nb-1,nb}a_{nb,nb-1}e_1^T A_{33}^{-1} e_1} \tag{6.9}$$

Now we are going to show $v_3^{new} \geq 0$. Recall that if matrix A is a Z-matrix, then we have A is a P-matrix and $A^{-1} \geq 0$. M_1 and it partition A_{33} are all Z-matrix, hence we have $A_{33}^{-1}e_1 \geq 0$ and M_1 is positive define. It follows that

$$\begin{aligned} a_{nb-1,nb-1} - a_{nb-1,nb}a_{nb,nb-1}e_1^T A_{33}^{-1} e_1 &= det \begin{pmatrix} a_{nb-1,nb-1} & a_{nb-1,nb}e_1^T \\ a_{nb,nb-1}e_1 & A_{33} \end{pmatrix} det(A_{33}^{-1}) \\ &> 0 \end{aligned}$$

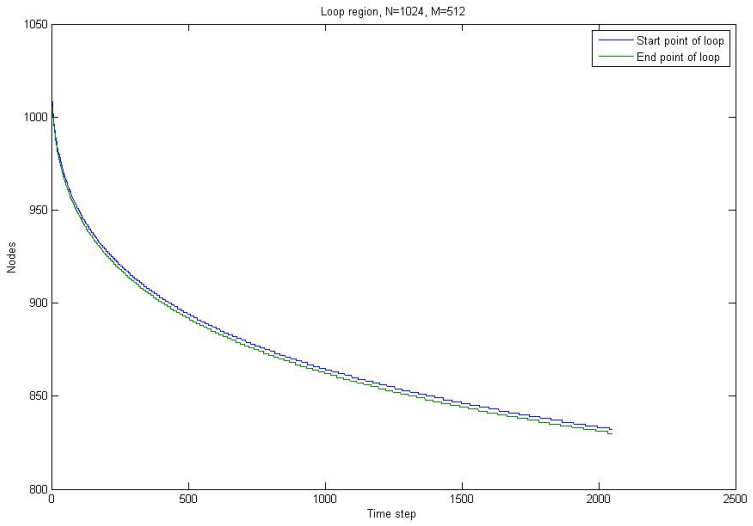
And we know that $s_2^{old} < 0$, so $z > 0$ therefore $v_3^{new} \geq 0$.

This algorithm can be stated as follows

Algorithm 6.1

Time loop: for $m = 1, 2, \ldots, M$
  $b = b^0 + M_2 v$
  for $nb = N+1, N, \ldots, 3, 2$
    get b_1, b_2, b_3 and A_{33} as in (6.3)
    if (6.4) holds
      compute s_2, v_3 as in (6.5) and (6.6)
      break;
    end if
  end for
  while $s_2 < 0$, do
    compute z according to (6.9)
    compute v_3 according to (6.8)
    nb:=nb-1
    if $nb = 0$
      break
    end if
    compute s_2 as in (6.7)
  end while
  v:=$[0_{1\times\ (nb-1)}\ v_3^T]^T$
end for

Figure 6.1: The iterations for Linear Programming in each time step, with time steps M=2048, nodes N=4096, the American option has strike $K = 0.25$, maturity time $T = 1$, risk free interest rate $r = 0.05$, and volatility $\sigma = 0.2$.



6.3 Improving the Algorithm

A.Borici and H.J. Luthi[6] proved that b^0 takes the following sign structure.

$$b^0 = \begin{pmatrix} \ominus \\ \oplus \end{pmatrix}$$

A.Borici and H.J. Luthi[6] also conclude that, at each time step, the starting complementary partition has property (6.4) may be given by the partition obtain in the last time step. That is, to find the partition with (6.4), we can seek from the position in last time step instead of seeking from the bottom. Figure 6.1 shows the begin points and end points of iteration in each time step.

Also we notice that at the end of nodes loop in time step m, all of the computation on v_3 is equivalent to $A_{33}^{-1}b_3$. Therefore, we do not need to compute v_3 at each node loop, just determine the nodes index nb at the last loop

and then compute it as follows.

$$v_3 = A_{33}^{-1} b_3$$

Computing matrix inverse is very costly in practice, now we will seek some algorithm to reduce the computational cost. The square matrix A_{33} can be expressed in the following form.

$$A_{33} = \begin{bmatrix} a_{11} & a_{12} e_1^T \\ a_{21} e_1 & B \end{bmatrix}$$

where a's are scalars and B is a square block in A_{33}. Suppose $B^{-1} e_1$ is given, then

$$A_{33}^{-1} e_1 = \begin{bmatrix} y \\ -a_{21} z B^{-1} e_1 \end{bmatrix} \tag{6.10}$$

where

$$y = \frac{1}{a_{11} - a_{21} a_{12} e_1^T B^{-1} e_1} \tag{6.11}$$

We also notice that if denoting $e_1^T A_{33}^{-1} e_1$ as y, (6.9) can be rewritten as below,

$$z = \frac{-s_2^{old}}{a_{nb-1,nb-1} - a_{nb-1,nb} a_{nb,nb-1} y^{old}}$$

while y can be computed recursively as

$$y^{new} = \frac{1}{a_{nb-1,nb-1} - a_{nb,nb-1} a_{nb-1,nb} y^{old}}$$

In the next problem we aim to solve for v_3 from the equation

$$A_{33} v_3 = b_3$$

with

$$A_{33} = \begin{bmatrix} a_{11} & a_{12} e_1^T \\ a_{21} e_1 & B \end{bmatrix}, \; b_3 = \begin{bmatrix} b^{(1)} \\ b_3^{old} \end{bmatrix}$$

Suppose the value of A_{33}, $B^{-1} e_1$ and b_3^{old} is already known, moreover, v_3^{old} have already been solved from $B v_3^{old} = b_3^{old}$, then v_3 can be obtained by

$$v_3 = \begin{bmatrix} f \\ v_3^{old} - a_{21} f B^{-1} e_1 \end{bmatrix} \tag{6.12}$$

where

$$f = \frac{b^{(1)} - a_{12} e_1^T v_3^{old}}{a_{11} - a_{12} a_{21} e_1^T B^{-1} e_1} \tag{6.13}$$

With all of these tools to reduce the computational cost in Algorithm (6.1), now we can present the improved algorithm as follows.

Linear Programming Method to Compute American Put Option

Initialization:

The same as *Initialization part in Explicit Method.*

$g := (g(S_0), g(S_1), \ldots, g(S_N))^T$,

where $g(\cdot)$ is the payoff function at maturity.

$b^0 = (M_2 - M_1)g$

$v := 0_{(N+1)\times 1}$, $s_2 := 0$

Core Calculations

find index nb such that $b_1^0 < 0$, $b_2^0 < 0$ and $b_3^0 > 0$

$pre\_nb := nb$

Time loop: for $m = 1, 2, \ldots, M$

$b := b^0 + M_2 v$

for $nb = pre\_nb, pre\_nb - 1, \ldots, 3, 2$

get b_1, b_2, b_3 and A_{33} as in (6.3)

if $nb == pre\_nb$

compute $A_{33}^{-1}b_3$ and $A_{33}^{-1}e_1$ by linear solver

else

compute $A_{33}^{-1}b_3$ as in (6.12), (6.13)

compute $A_{33}^{-1}e_1$ as in (6.10), (6.11)

end if

if (6.4) holds

$pre\_nb := nb$

$z := e_1^T A_{33}^{-1} b_3$, $y := -a_{nb,nb-1} e_1^T A_{33}^{-1} e_1$

$s_2 := -b_2 + a_{nb-1,nb} z$

break

end if

end for

while $s_2 < 0$, do

$z := -s_2/(a_{nb-1,nb-1} + a_{nb-1,nb} y)$

$y := -a_{nb,nb-1}/(a_{nb-1,nb-1} + a_{nb-1,nb} y)$

if $nb == 1$, break, end if

$nb := nb - 1$

$s_2 := -b_2 + a_{nb-1,nb} z$

end while

get b_3 and A_{33} as in (6.3)

$v_3 := A_{33}^{-1} b_3$, $v := [0_{1\times\ (nb-1)}\ v_3^T]^T$

end for

final result: $u := v + g$

7 Numerical Results

In this section, we will demonstrate some numerical results from empirical tests on the four algorithms: Explicit, PSOR, Penalty and Linear Programming. These four algorithms will be applied to price a specific American put option, which has strike $K = 0.25$, maturity time $T = 1$, risk free interest rate $r = 0.05$, and volatility $\sigma = 0.2$.
Crank-Nicolson scheme $\theta = 0.5$ is used in finite different discretisation. In PSOR algorithm, the relaxation parameter is set to be $\omega = 1.3$, and the convergence error tolerance $\lambda = 10^{-10}$.
Sometime we need to measure the error from the comparison to the "exact solution", which is approximated by penalty algorithm with 10000 time steps and 10000 price nodes.

Tables $7.1 - 7.4$ and Figures $7.1 - 7.3$ represent numerical results on error convergence and time costs. In practical, people care about the option pricing error when it is around the spot at S = K, thus we also present some numerical analysis on option value at $S = K$. Note that some decimals may not displayed due to the limitation of the width of the tables. The values of "rate" under "Value at (S=K)" item are calculated by $rate_i = change_i/change_{i-1}$. Analogously, the "rate" on "Average error" is given by $rate_i = value_i/value_{i-1}$.

We can see in Figure 7.1 and 7.2 the error of Penalty and Linear Programming totally overlap each other, while PSOR has the same convergence rate until the average absolute error goes to 10^{-8}, then it becomes slower than Linear Programming and Penalty. Obviously, Explicit method gives the least accurate level, because this method just "forces" the value to equal to one lower boundary when the value is less than it, which violates the complementary conditions.
Penalty, PSOR and Linear Programming converge with second order in both $\Delta\tau$ and ΔS, while Explicit converges with first order in $\Delta\tau$ and second order in ΔS. In Table $2 - 4$, the rates about average absolute error are approximately 1/4, reflecting the second order of convergence. These rates for Explicit are obviously greater than 1/4 and they keep growing with nodes

Table 7.1: Explicit Method

Nodes	Time Steps	Iterations	Time used	Value at (S=K) value	change	rate	Average error value	rate
128	64	64	0.0341	0.01514			7.91E-06	
256	128	128	0.0489	0.0152	6.24E-05		2.63E-06	0.332
512	256	256	0.1092	0.01522	1.792E-05	0.2873	9.82E-07	0.3739
1024	512	512	0.3392	0.01522	5.865E-06	0.3272	4.10E-07	0.4176
2048	1024	1024	1.2779	0.01522	2.177E-06	0.3711	1.82E-07	0.4442
4096	2048	2048	4.8725	0.01523	8.995E-07	0.4132	8.54E-08	0.4689
8192	4096	4096	20.488	0.01523	4.034E-07	0.4485	4.08E-08	0.4775

Table 7.2: PSOR Method

Nodes	Time Steps	Iteration	Time used	Value at (S=K) value	change	rate	Average error value	rate
128	64	527	0.0207	0.01515			5.87E-06	
256	128	915	0.0619	0.01521	5.341E-05		1.54E-06	0.263
512	256	1611	0.2149	0.01522	1.323E-05	0.2477	4.35E-07	0.2818
1024	512	2770	0.7661	0.01522	3.438E-06	0.2598	1.08E-07	0.248
2048	1024	4984	2.9004	0.01523	8.735E-07	0.2541	2.90E-08	0.2688
4096	2048	11783	13.167	0.01523	2.309E-07	0.2643	7.86E-09	0.2712
8192	4096	33663	63.623	0.01523	5.562E-08	0.2409	2.80E-09	0.3567

Table 7.3: Penalty Method

Nodes	Time Steps	Iterations	Time used	Value at (S=K) value	change	rate	Average error value	rate
128	64	133	0.0564	0.01515			5.87E-06	
256	128	267	0.1095	0.01521	5.341E-05		1.54E-06	0.263
512	256	534	0.3095	0.01522	1.323E-05	0.2477	4.35E-07	0.2819
1024	512	1069	1.0754	0.01522	3.437E-06	0.2598	1.08E-07	0.2481
2048	1024	2137	4.2304	0.01523	8.74E-07	0.2543	2.90E-08	0.2684
4096	2048	4261	17.078	0.01523	2.341E-07	0.2678	7.49E-09	0.2585
8192	4096	8443	66.85	0.01523	6.57E-08	0.2807	1.33E-09	0.1778

Table 7.4: Linear Programming Method

Nodes	Time Steps	Iterations	Time used	Value at (S=K) value	change	rate	Average error value	rate
128	64	84	0.0426	0.01515			5.87E-06	
256	128	182	0.0845	0.01521	5.341E-05		1.54E-06	0.263
512	256	412	0.2274	0.01522	1.323E-05	0.2477	4.35E-07	0.2819
1024	512	1043	0.8891	0.01522	3.437E-06	0.2598	1.08E-07	0.2481
2048	1024	2622	3.4306	0.01523	8.74E-07	0.2543	2.90E-08	0.2684
4096	2048	6854	13.506	0.01523	2.341E-07	0.2678	7.49E-09	0.2585
8192	4096	18224	65.399	0.01523	6.57E-08	0.2807	1.33E-09	0.1778

and time steps, this is because its slower error convergence rate compared with other three methods.
Figure (7.3) depicts the time costs of the four algorithms. Explicit method is the fastest one, as the computation within each time step is very simple. PSOR, Penalty and Linear Programming are quite close to each other, while PSOR is little bit faster that the other two.

Let A be a square matrix, we solve f from $Af = b$ by compute $f = A^{-1}b$, it can be done by $f = A \setminus b$ in Matlab. This computation is usually called "call a linear system solver". Consider the case with nodes $N = 4096$, $M = 2048$, Table 7.5 shows the number of times calling a linear solver in each time step. There is only one time for each of Explicit and PSOR, while three times for

Figure 7.1: Convergence of absolute average error

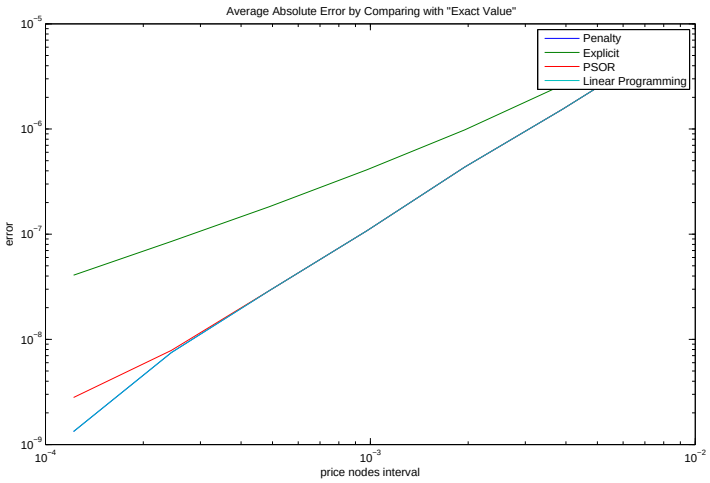


Figure 7.2: Convergence of error at (S=K)

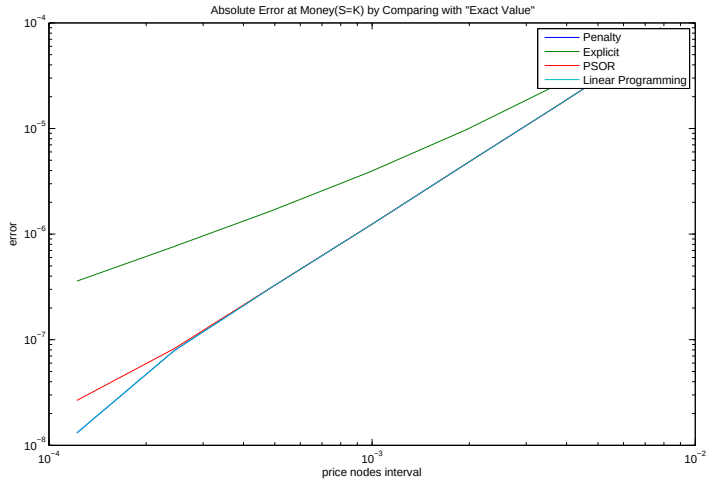


Figure 7.3: Time cost for the four methods

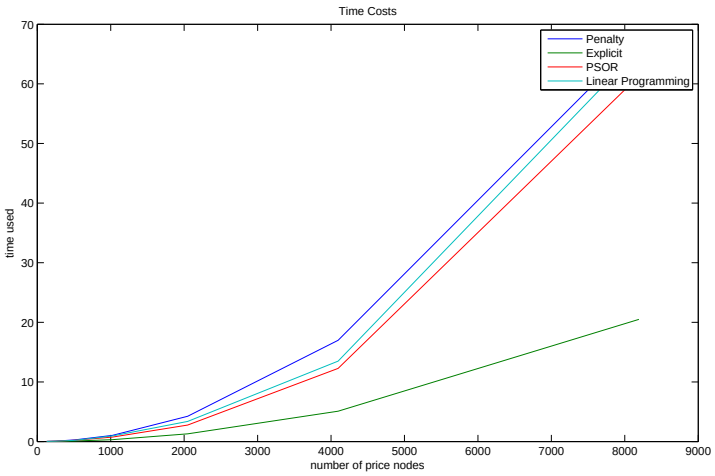


Table 7.5: Number of times call for a linear system solver, $N = 4096$, $M = 2048$

Explicit	PSOR	Penalty	Linear Programming
1	1	2.08	3

Linear Programming. For Penalty method, it calls linear system solver at every iteration for convergence; fortunately, the average number of iterations is only 2.08 to converge at each time step.

From figure 7.4, we can see that the time spent on linear system solver accounts for the majority of the total time cost. For Linear Programming, when the linear system solvers are called, the involving matrix is less than $N \times N$, in fact, it only computes part of the $N \times N$ matrix. So the time spent on linear system solvers is not as much as three times of that in Explicit or PSOR.

PSOR calls linear system solver at initializing part, $v^{(0)} := max\{M_1^{-1}b^{(m)}, g(S)\}$, in fact, this could be replaced by $v^{(0)} := max\{w^{m-1}, g(S)\}$, which can make

Figure 7.4: Time steps M=2048, Nodes N=4096. Ex:=Explicit, PS:=PSOR, Pe:=Penalty, LP:=Linear Programming

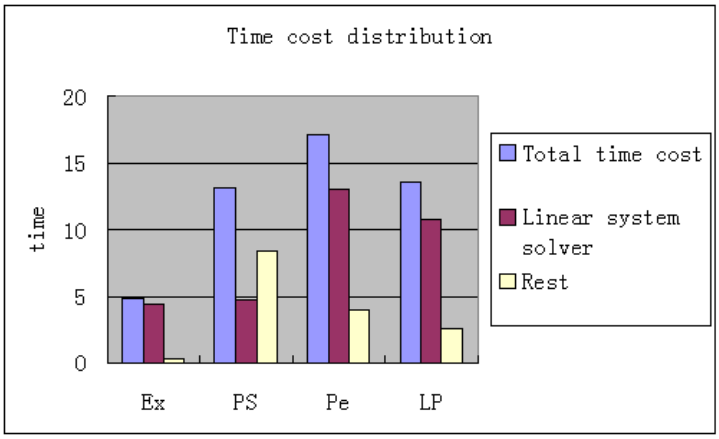
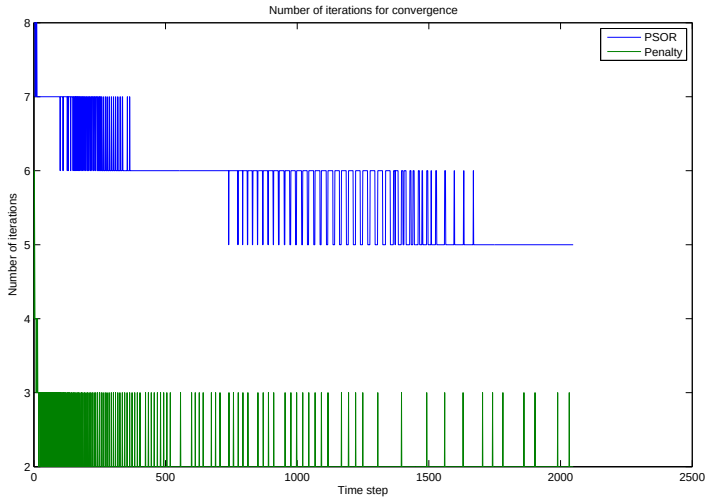


Figure 7.5: PSOR and Penalty: number of iterations needed for convergence in each time step, M=2048, N=4096

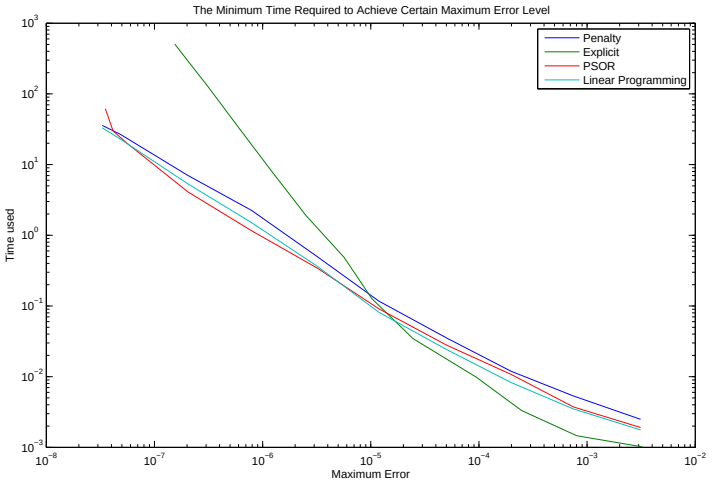


the program work without calling an linear system solver. However, according to empirical test, it takes more iterations for convergence, resulting in a

sharp increase on total time cost with no help on accuracy.

In Figure 7.4, the time cost under "rest" means total time consumed not by calling linear system solvers. PSOR and Penalty both need iterations to ensure convergence in each time step, the number of iterations is shown in Figure 7.5. Obviously, PSOR needs more iterations to converge, which leads to more time spend on "rest" part than other methods. Linear Programming also has iterations in each time step, but the computational cost is small. The details of the loop is shown in Figure 6.1.

Figure 7.6: The minimum time required to achieve certain maximum error level



In real activities, people care about to achieve a certain level of accuracy, which method can provide minimum time cost. From Figure 7.6 we can see that the explicit method is the fastest one when error level greater than 10^{-3}. But if lower level of error is required, the time cost of Explicit blows up. PSOR takes the least time cost at the error level between 10^{-6} and 10^{-7}. When the error level close to 10^{-8}, its performance is not as good as Penalty and Linear Programming, due to the slower error convergence which can be seen in figure 7.1.

In conclusion, Explicit is simple and fast, but it doesn't provide good solution of LCP. PSOR, Penalty and Linear Programming are all reliable with high accuracy level. PSOR is derived from Successive Over-Relaxation(SOR) Algorithm, which is an iterative method to solve (4.1). Cryer(1971) developed Projected SOR so that the LCP constraints can be handled at the same time. Linear Programming algorithm considers the LCP as an equivalent Linear Programs problem, and solves the problem by finding the optimal values. The idea of Penalty is to penalize the violation against the LCP constraints, and it can handle options with more complicated payoffs, for instance, step options.

References

[1] Christoph Reisinger, *Numerical Methods I: Finite Difference Methods*, Lecture notes, 2008.

[2] P.A.Forsyth and K.R.Vetzal. *Quadratic Convergence of a Penalty Method for Valuing American Options*. SIAM Journal on Scientific Computation, 23:2096–2123, 2002.

[3] Josef Stoer, Roland Bulirsch: *Numerische Mathematik 2*, Springer, 2000

[4] C.W.Cryer. *The Solution of Quadratic Programming Problem Using Systematic Overrelaxation*. SIAM Journal on Control and Optimization, 1997.

[5] M.Fiedler and V.Ptak, *On Matrices with Non-positive Off-diagonal Elements and Positive Principal Minors*, Czechoslovak Math. J. 12(1962) 382-400.

[6] A.Borici and H.J.Lüthi, *Pricing American Put Options By Fast Solutions Of The Linear Complementarity Problem*, Computational Methods In Decision-making, Economics And finance, 2002

[7] John Gilbert, online notes, http://www.maths.lancs.ac.uk/%7Egilbert/m306b/node14.html

[8] Oliver Pauly, *Numerical Simulation of American Options*, 2004.

[9] Y.d'halluin, P.A.Forsyth, and G.Labahn, *A Penalty Method for American Options with Jump Diffusion Process*, Numerische Mathematik, Volume 97, 2003

[10] M.A.H.Dempster, J.P.Hutton, *Pricing American Stock Options by Linear Programming*, Mathematical Finance, 1999.