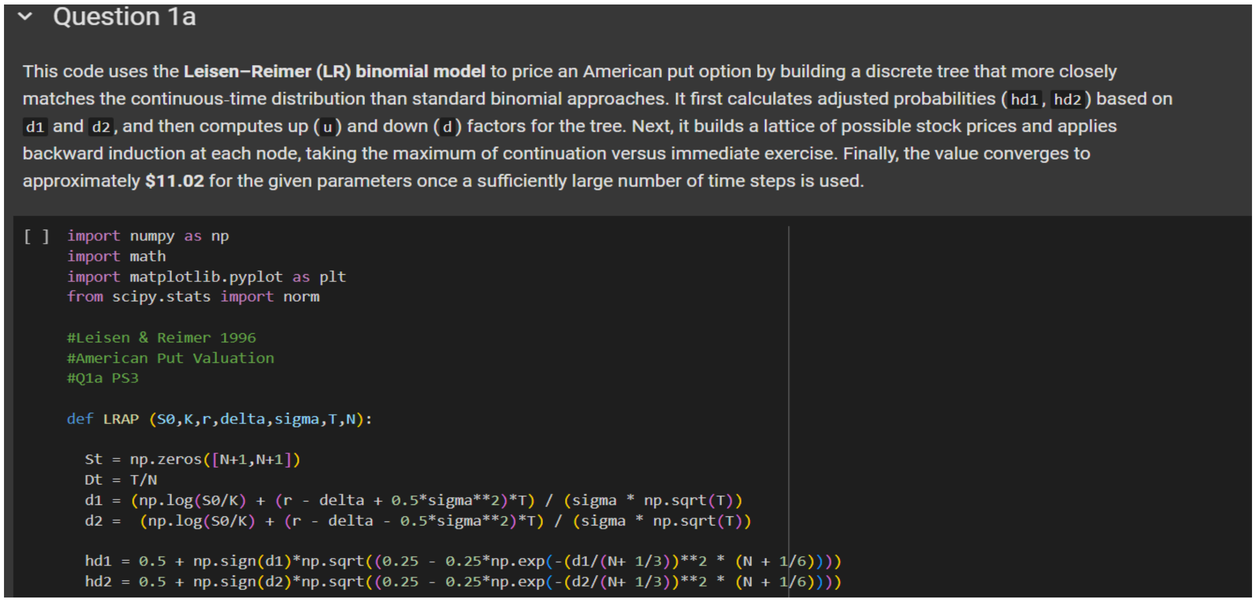
Q1)

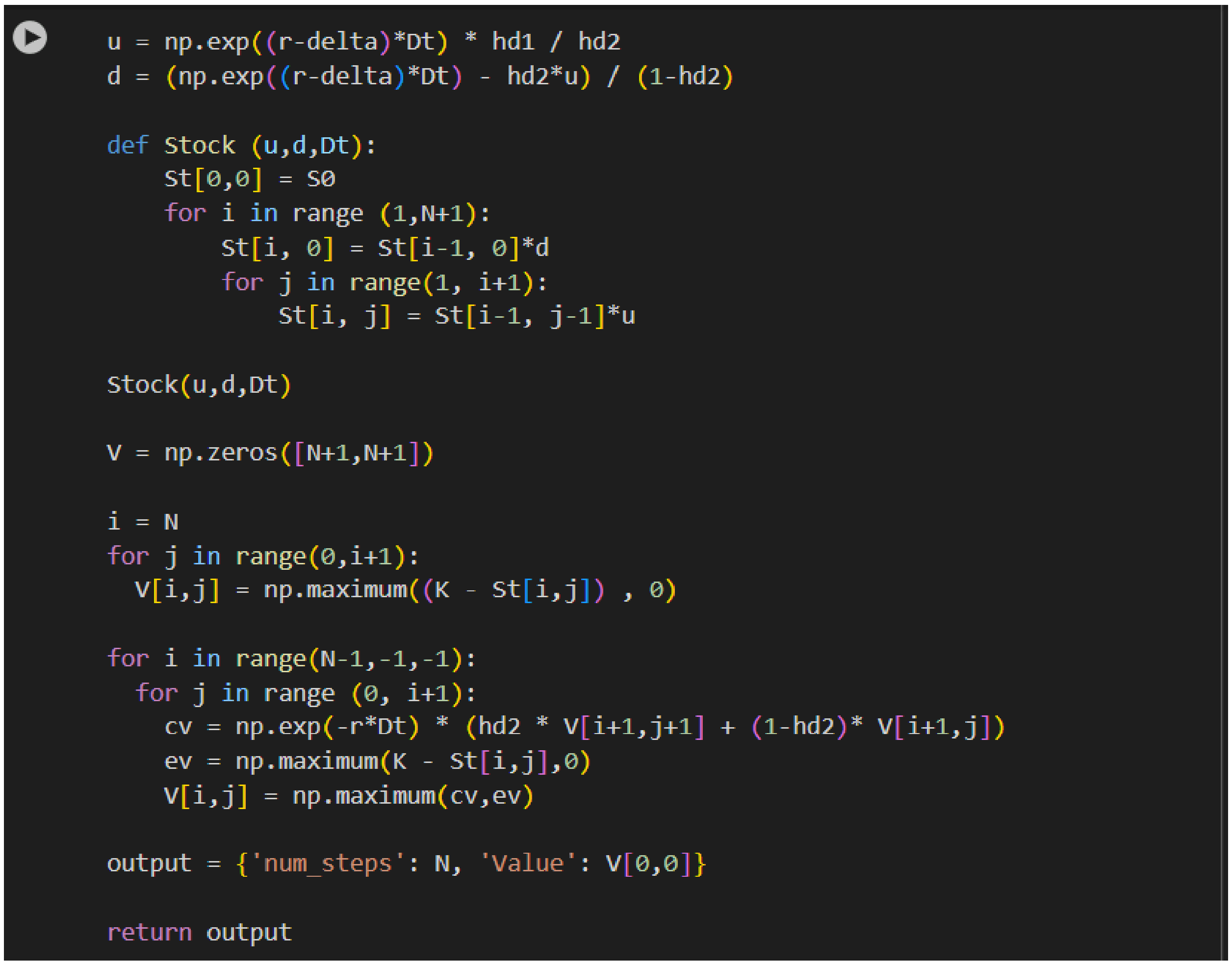
A1a) The value of put option via LR method with N=10001 is 11.02035348



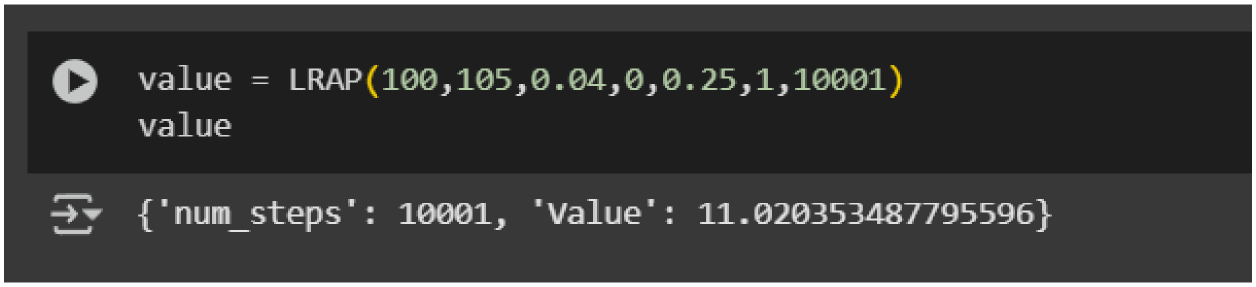
Question 1a

This code uses the **Leisen–Reimer (LR) binomial model** to price an American put option by building a discrete tree that more closely matches the continuous-time distribution than standard binomial approaches. It first calculates adjusted probabilities (`hd1`, `hd2`) based on `d1` and `d2`, and then computes up (`u`) and down (`d`) factors for the tree. Next, it builds a lattice of possible stock prices and applies backward induction at each node, taking the maximum of continuation versus immediate exercise. Finally, the value converges to approximately **$11.02** for the given parameters once a sufficiently large number of time steps is used.

```
[ ] import numpy as np
    import math
    import matplotlib.pyplot as plt
    from scipy.stats import norm

    #Leisen & Reimer 1996
    #American Put Valuation
    #Q1a PS3

    def LRAP (S0,K,r,delta,sigma,T,N):

      St = np.zeros([N+1,N+1])
      Dt = T/N
      d1 = (np.log(S0/K) + (r - delta + 0.5*sigma**2)*T) / (sigma * np.sqrt(T))
      d2 =  (np.log(S0/K) + (r - delta - 0.5*sigma**2)*T) / (sigma * np.sqrt(T))

      hd1 = 0.5 + np.sign(d1)*np.sqrt((0.25 - 0.25*np.exp(-(d1/(N+ 1/3))**2 * (N + 1/6))))
      hd2 = 0.5 + np.sign(d2)*np.sqrt((0.25 - 0.25*np.exp(-(d2/(N+ 1/3))**2 * (N + 1/6))))
```



```
  u = np.exp((r-delta)*Dt) * hd1 / hd2
  d = (np.exp((r-delta)*Dt) - hd2*u) / (1-hd2)

  def Stock (u,d,Dt):
      St[0,0] = S0
      for i in range (1,N+1):
          St[i, 0] = St[i-1, 0]*d
          for j in range(1, i+1):
              St[i, j] = St[i-1, j-1]*u

  Stock(u,d,Dt)

  V = np.zeros([N+1,N+1])

  i = N
  for j in range(0,i+1):
    V[i,j] = np.maximum((K - St[i,j]) , 0)

  for i in range(N-1,-1,-1):
    for j in range (0, i+1):
      cv = np.exp(-r*Dt) * (hd2 * V[i+1,j+1] + (1-hd2)* V[i+1,j])
      ev = np.maximum(K - St[i,j],0)
      V[i,j] = np.maximum(cv,ev)

  output = {'num_steps': N, 'Value': V[0,0]}

  return output
```

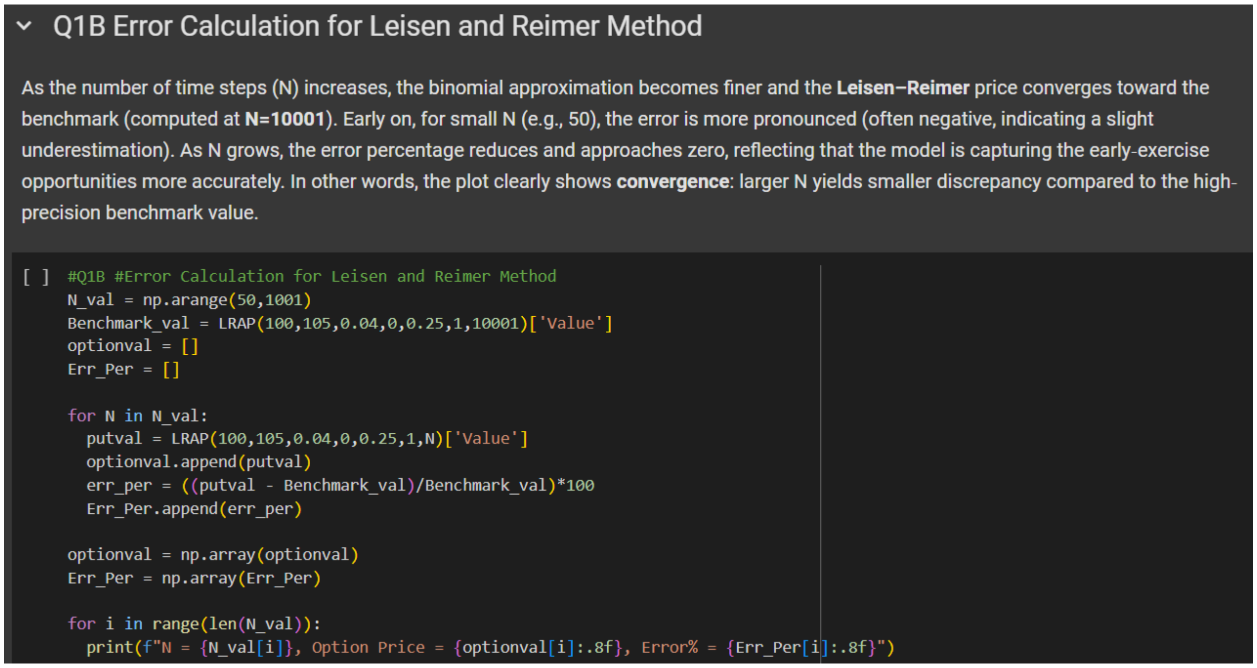


```
value = LRAP(100,105,0.04,0,0.25,1,10001)
value
```

```
{'num_steps': 10001, 'Value': 11.020353487795596}
```

Q1b)

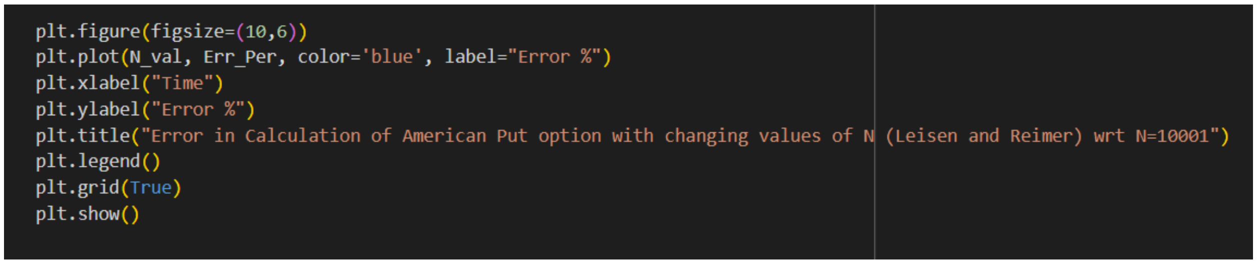
For LR model we see as we increase N the oscillation between errors of odd and even N decreases and it converges, as in LR method odd N has low error compared to even N



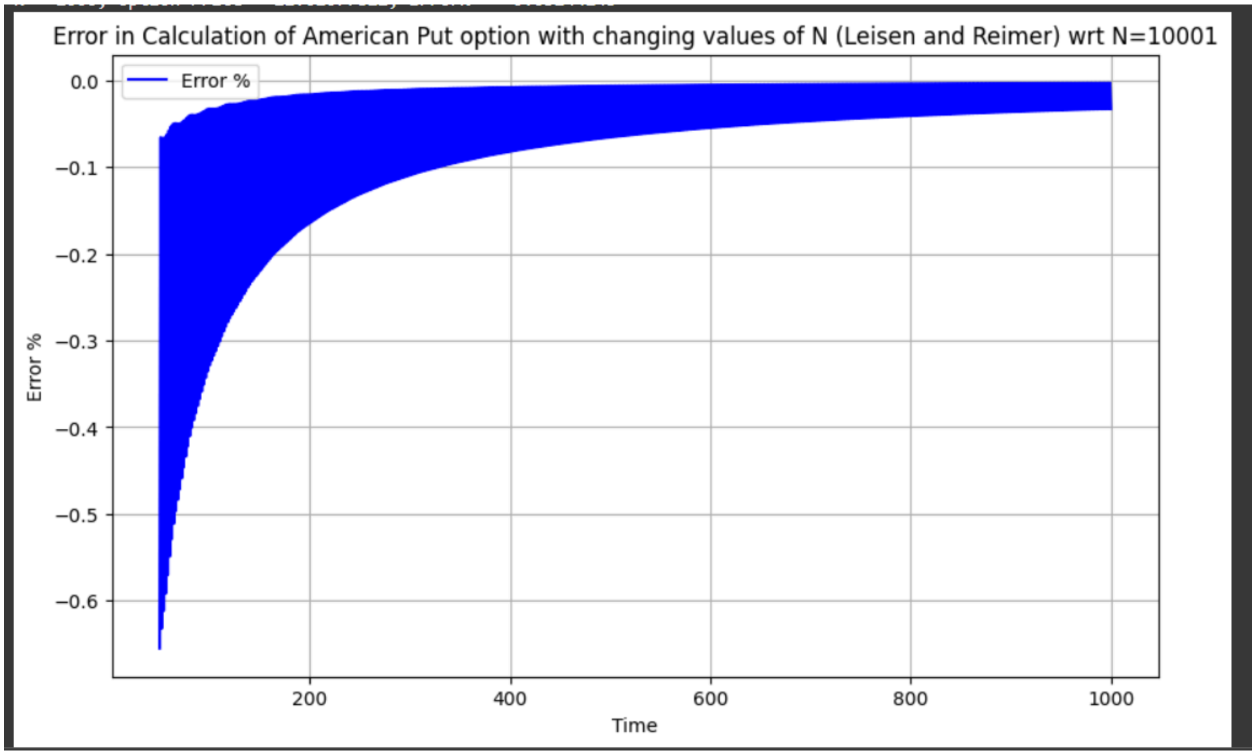
Q1B Error Calculation for Leisen and Reimer Method

As the number of time steps (N) increases, the binomial approximation becomes finer and the **Leisen–Reimer** price converges toward the benchmark (computed at **N=10001**). Early on, for small N (e.g., 50), the error is more pronounced (often negative, indicating a slight underestimation). As N grows, the error percentage reduces and approaches zero, reflecting that the model is capturing the early-exercise opportunities more accurately. In other words, the plot clearly shows **convergence**: larger N yields smaller discrepancy compared to the high-precision benchmark value.

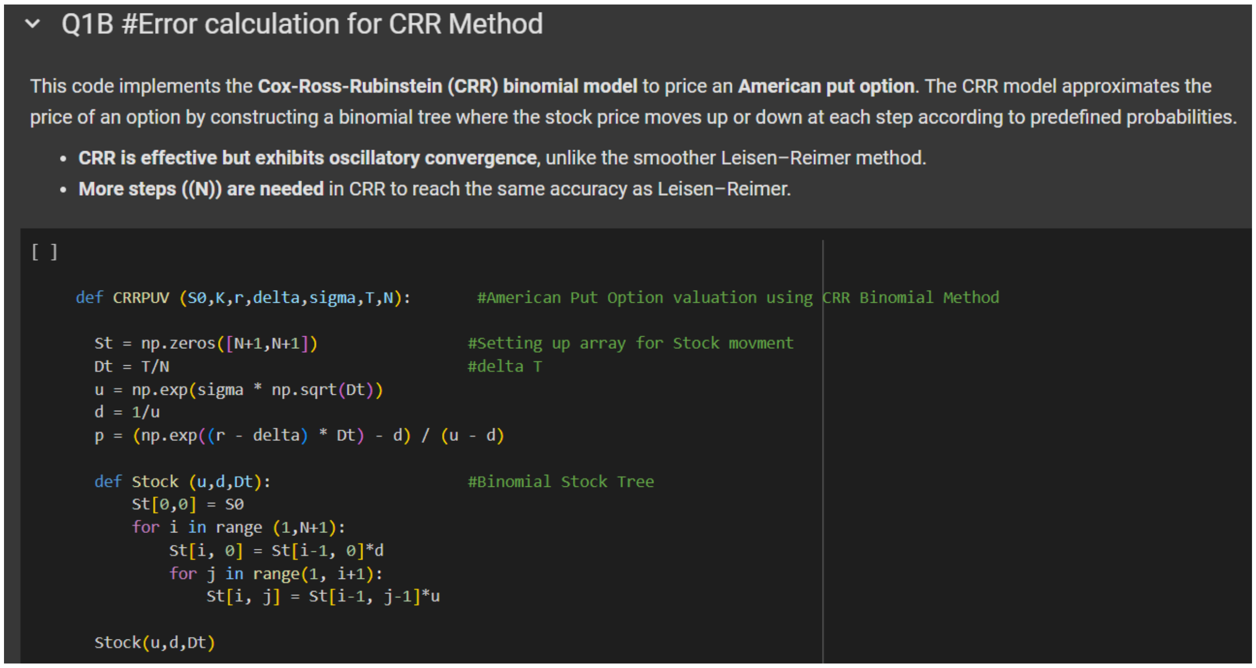
```
#Q1B #Error Calculation for Leisen and Reimer Method
N_val = np.arange(50,1001)
Benchmark_val = LRAP(100,105,0.04,0,0.25,1,10001)['Value']
optionval = []
Err_Per = []

for N in N_val:
  putval = LRAP(100,105,0.04,0,0.25,1,N)['Value']
  optionval.append(putval)
  err_per = ((putval - Benchmark_val)/Benchmark_val)*100
  Err_Per.append(err_per)

optionval = np.array(optionval)
Err_Per = np.array(Err_Per)

for i in range(len(N_val)):
  print(f"N = {N_val[i]}, Option Price = {optionval[i]:.8f}, Error% = {Err_Per[i]:.8f}")
```



```
plt.figure(figsize=(10,6))
plt.plot(N_val, Err_Per, color='blue', label="Error %")
plt.xlabel("Time")
plt.ylabel("Error %")
plt.title("Error in Calculation of American Put option with changing values of N (Leisen and Reimer) wrt N=10001")
plt.legend()
plt.grid(True)
plt.show()
```



1.b For CRR method we see as we increases N the error continuously converges but requires more steps than LR model to converge

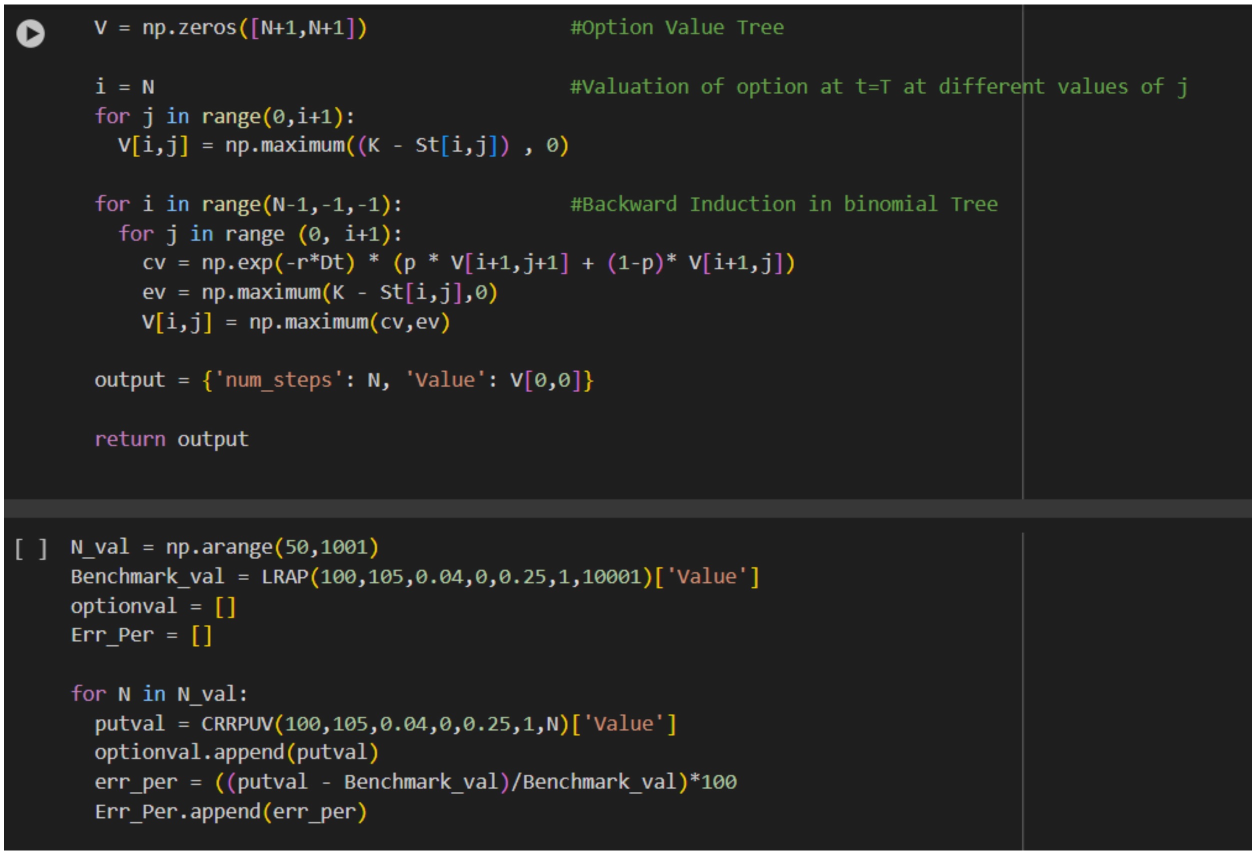


Q1B #Error calculation for CRR Method

This code implements the **Cox-Ross-Rubinstein (CRR) binomial model** to price an **American put option**. The CRR model approximates the price of an option by constructing a binomial tree where the stock price moves up or down at each step according to predefined probabilities.

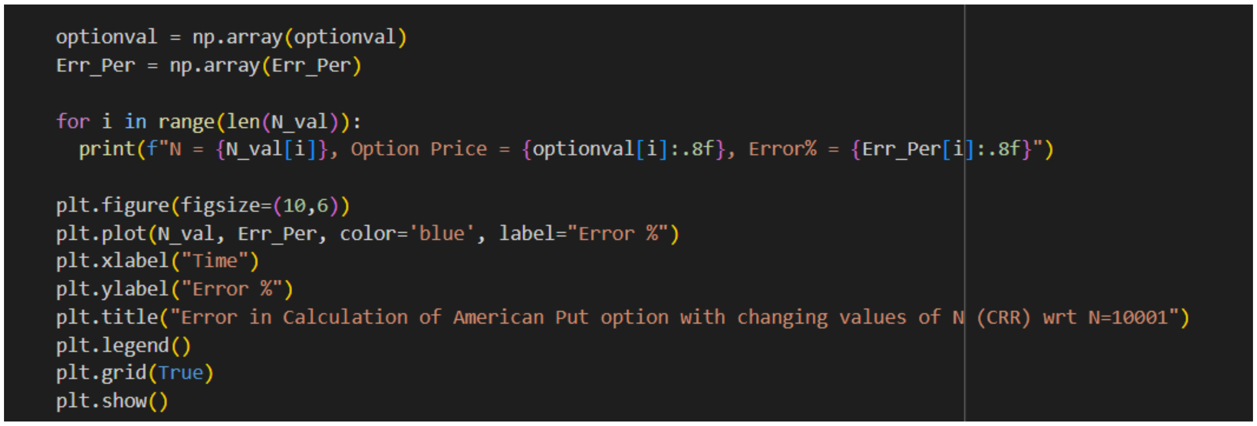
- **CRR is effective but exhibits oscillatory convergence,** unlike the smoother Leisen–Reimer method.
- **More steps ((N)) are needed** in CRR to reach the same accuracy as Leisen–Reimer.

```
[ ]

def CRRPUV (S0,K,r,delta,sigma,T,N):       #American Put Option valuation using CRR Binomial Method

  St = np.zeros([N+1,N+1])                #Setting up array for Stock movment
  Dt = T/N                                #delta T
  u = np.exp(sigma * np.sqrt(Dt))
  d = 1/u
  p = (np.exp((r - delta) * Dt) - d) / (u - d)

  def Stock (u,d,Dt):                     #Binomial Stock Tree
      St[0,0] = S0
      for i in range (1,N+1):
          St[i, 0] = St[i-1, 0]*d
          for j in range(1, i+1):
              St[i, j] = St[i-1, j-1]*u

  Stock(u,d,Dt)
```

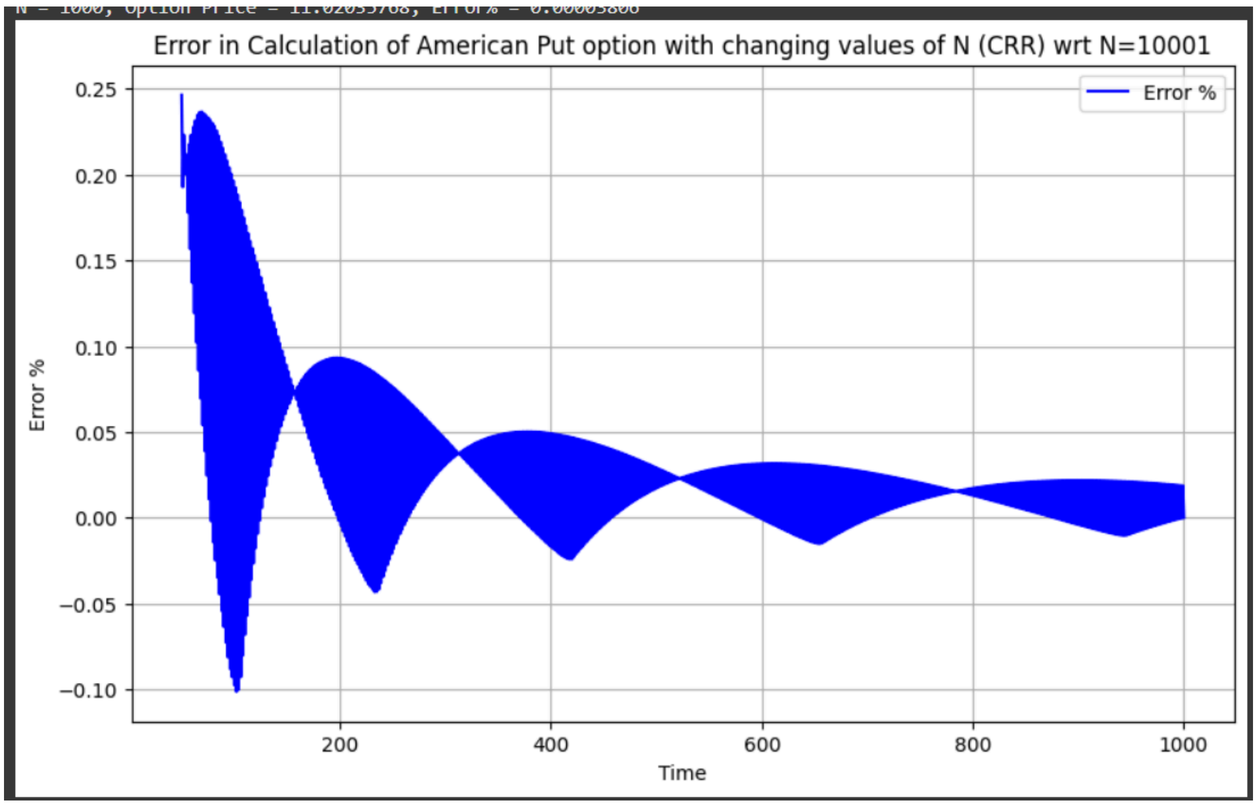


```python
  V = np.zeros([N+1,N+1])                  #Option Value Tree

  i = N                                    #Valuation of option at t=T at different values of j
  for j in range(0,i+1):
    V[i,j] = np.maximum((K - St[i,j]) , 0)

  for i in range(N-1,-1,-1):               #Backward Induction in binomial Tree
    for j in range (0, i+1):
      cv = np.exp(-r*Dt) * (p * V[i+1,j+1] + (1-p)* V[i+1,j])
      ev = np.maximum(K - St[i,j],0)
      V[i,j] = np.maximum(cv,ev)

  output = {'num_steps': N, 'Value': V[0,0]}

  return output
```

```python
[ ] N_val = np.arange(50,1001)
    Benchmark_val = LRAP(100,105,0.04,0,0.25,1,10001)['Value']
    optionval = []
    Err_Per = []

    for N in N_val:
      putval = CRRPUV(100,105,0.04,0,0.25,1,N)['Value']
      optionval.append(putval)
      err_per = ((putval - Benchmark_val)/Benchmark_val)*100
      Err_Per.append(err_per)
```

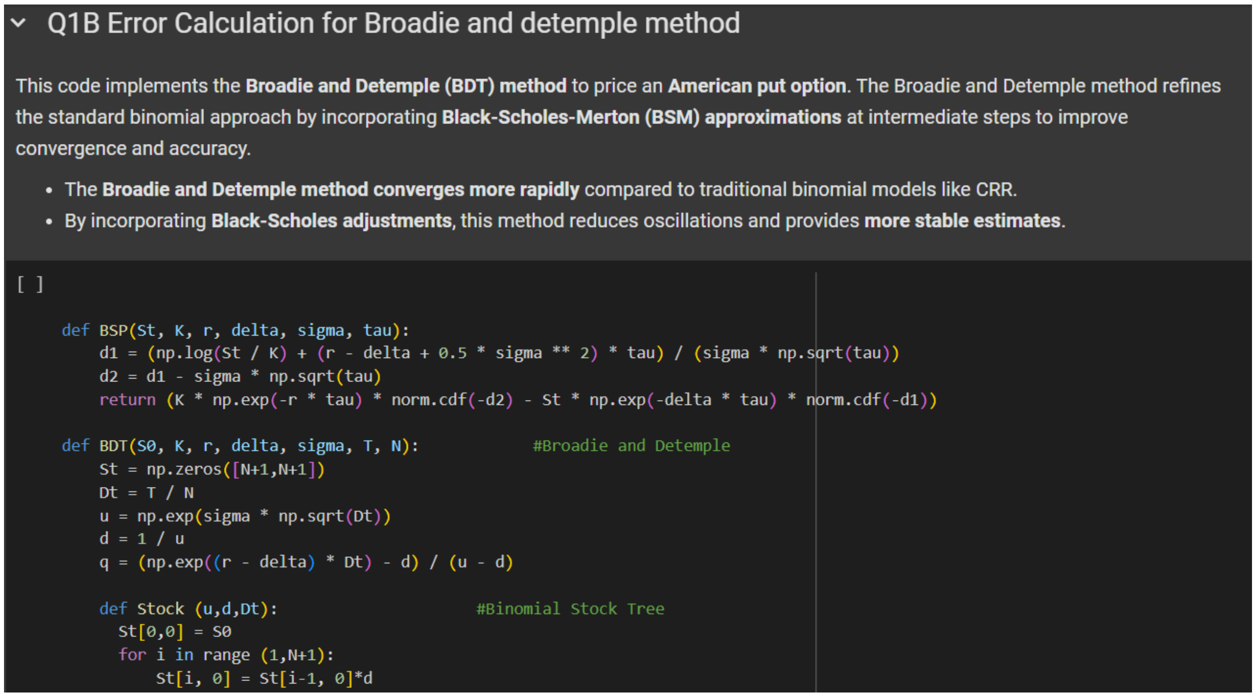


```python
optionval = np.array(optionval)
Err_Per = np.array(Err_Per)

for i in range(len(N_val)):
  print(f"N = {N_val[i]}, Option Price = {optionval[i]:.8f}, Error% = {Err_Per[i]:.8f}")

plt.figure(figsize=(10,6))
plt.plot(N_val, Err_Per, color='blue', label="Error %")
plt.xlabel("Time")
plt.ylabel("Error %")
plt.title("Error in Calculation of American Put option with changing values of N (CRR) wrt N=10001")
plt.legend()
plt.grid(True)
plt.show()
```



```
N = 1000, Option Price = 11.02033768, Error% = 0.00003808
```

In Broadie and Detemple we use a combination of Black Scholes and CRR to reduce error at expiration and we observer least error and fastest convergence through this method.

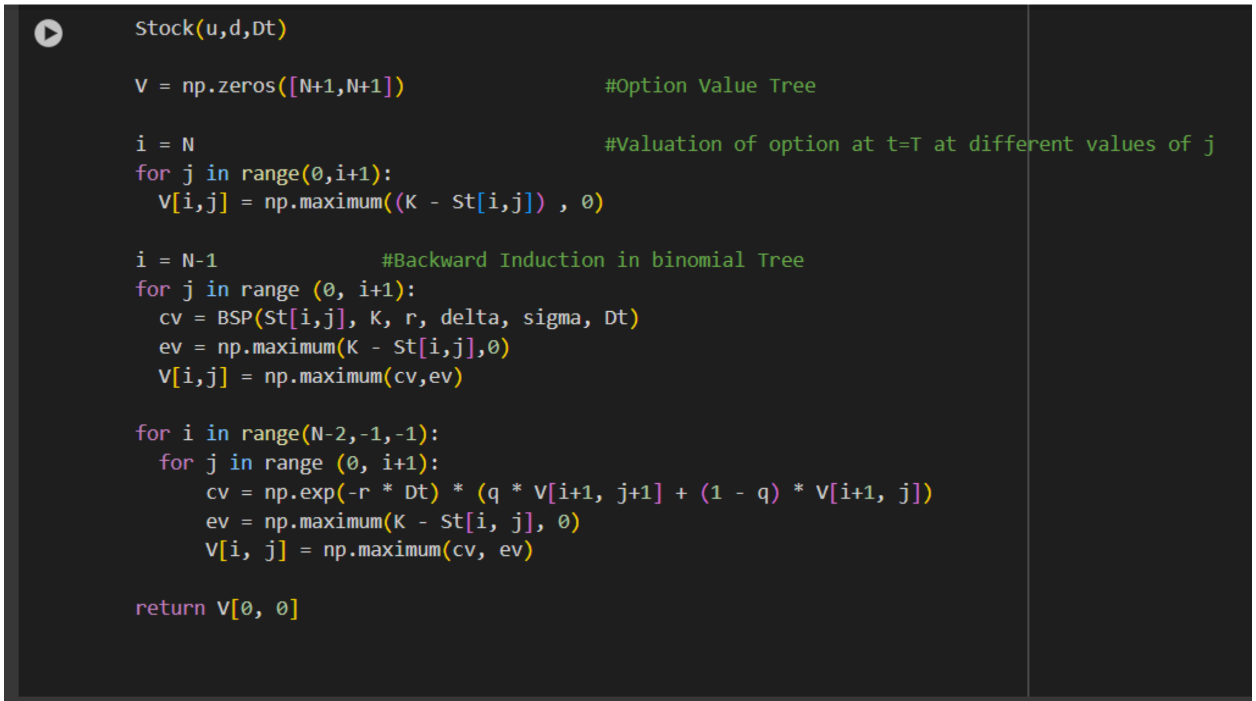


Q1B Error Calculation for Broadie and detemple method

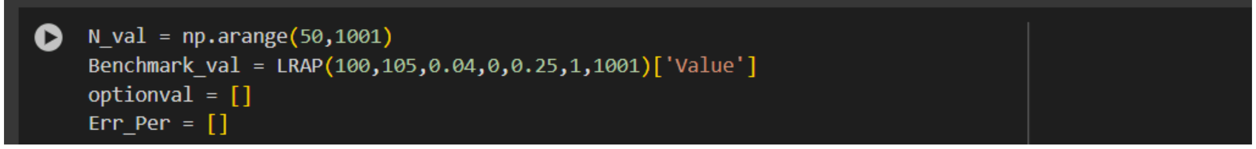
This code implements the **Broadie and Detemple (BDT) method** to price an **American put option**. The Broadie and Detemple method refines the standard binomial approach by incorporating **Black-Scholes-Merton (BSM) approximations** at intermediate steps to improve convergence and accuracy.

- The **Broadie and Detemple method converges more rapidly** compared to traditional binomial models like CRR.
- By incorporating **Black-Scholes adjustments**, this method reduces oscillations and provides **more stable estimates**.

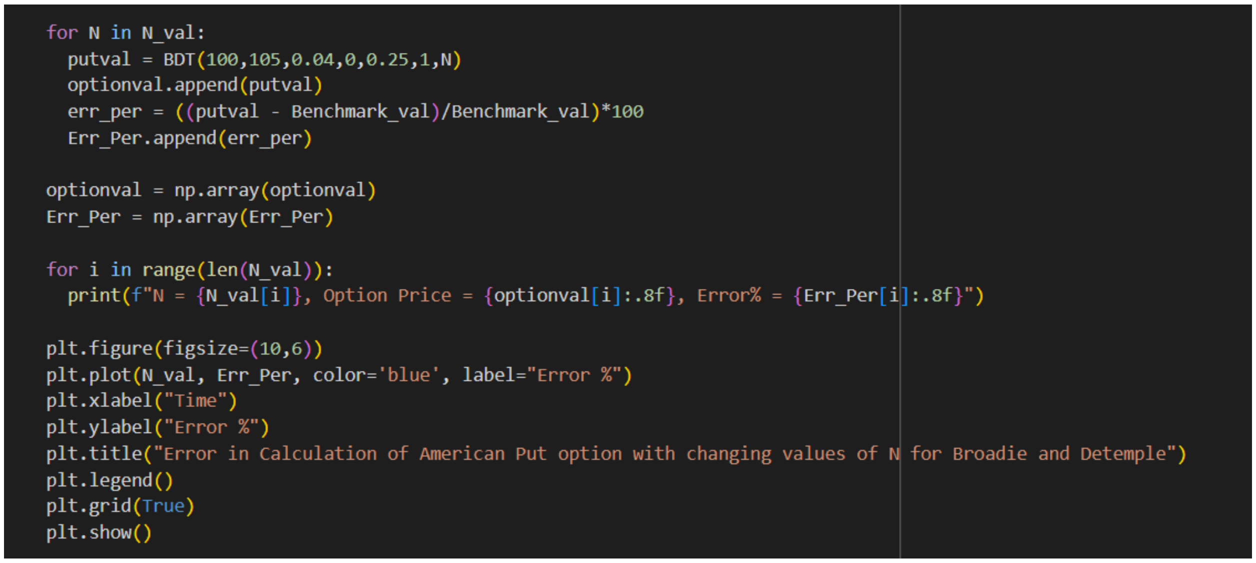
```
[ ]

    def BSP(St, K, r, delta, sigma, tau):
        d1 = (np.log(St / K) + (r - delta + 0.5 * sigma ** 2) * tau) / (sigma * np.sqrt(tau))
        d2 = d1 - sigma * np.sqrt(tau)
        return (K * np.exp(-r * tau) * norm.cdf(-d2) - St * np.exp(-delta * tau) * norm.cdf(-d1))

    def BDT(S0, K, r, delta, sigma, T, N):            #Broadie and Detemple
        St = np.zeros([N+1,N+1])
        Dt = T / N
        u = np.exp(sigma * np.sqrt(Dt))
        d = 1 / u
        q = (np.exp((r - delta) * Dt) - d) / (u - d)

        def Stock (u,d,Dt):                          #Binomial Stock Tree
          St[0,0] = S0
          for i in range (1,N+1):
              St[i, 0] = St[i-1, 0]*d
```



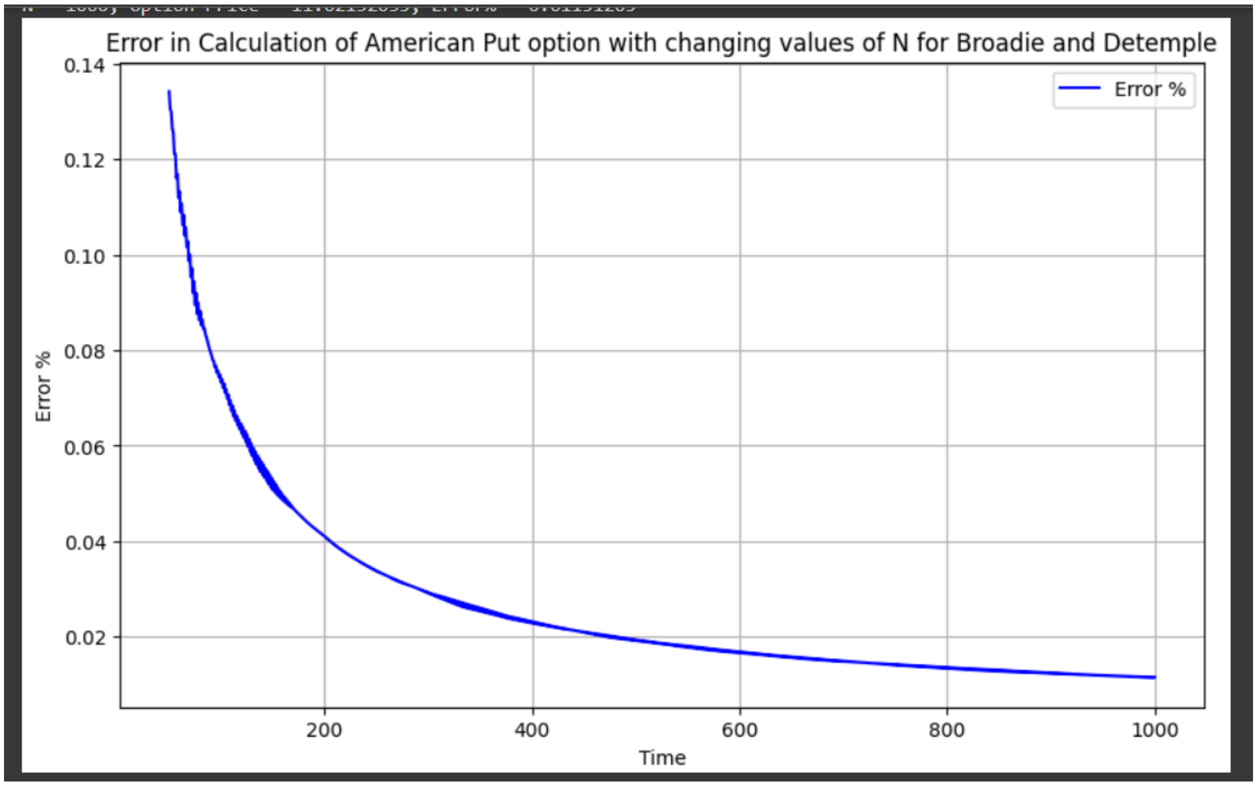
```
      Stock(u,d,Dt)

      V = np.zeros([N+1,N+1])                   #Option Value Tree

      i = N                                     #Valuation of option at t=T at different values of j
      for j in range(0,i+1):
        V[i,j] = np.maximum((K - St[i,j]) , 0)

      i = N-1                  #Backward Induction in binomial Tree
      for j in range (0, i+1):
        cv = BSP(St[i,j], K, r, delta, sigma, Dt)
        ev = np.maximum(K - St[i,j],0)
        V[i,j] = np.maximum(cv,ev)

      for i in range(N-2,-1,-1):
        for j in range (0, i+1):
            cv = np.exp(-r * Dt) * (q * V[i+1, j+1] + (1 - q) * V[i+1, j])
            ev = np.maximum(K - St[i, j], 0)
            V[i, j] = np.maximum(cv, ev)

      return V[0, 0]
```



```
N_val = np.arange(50,1001)
Benchmark_val = LRAP(100,105,0.04,0,0.25,1,1001)['Value']
optionval = []
Err_Per = []
```

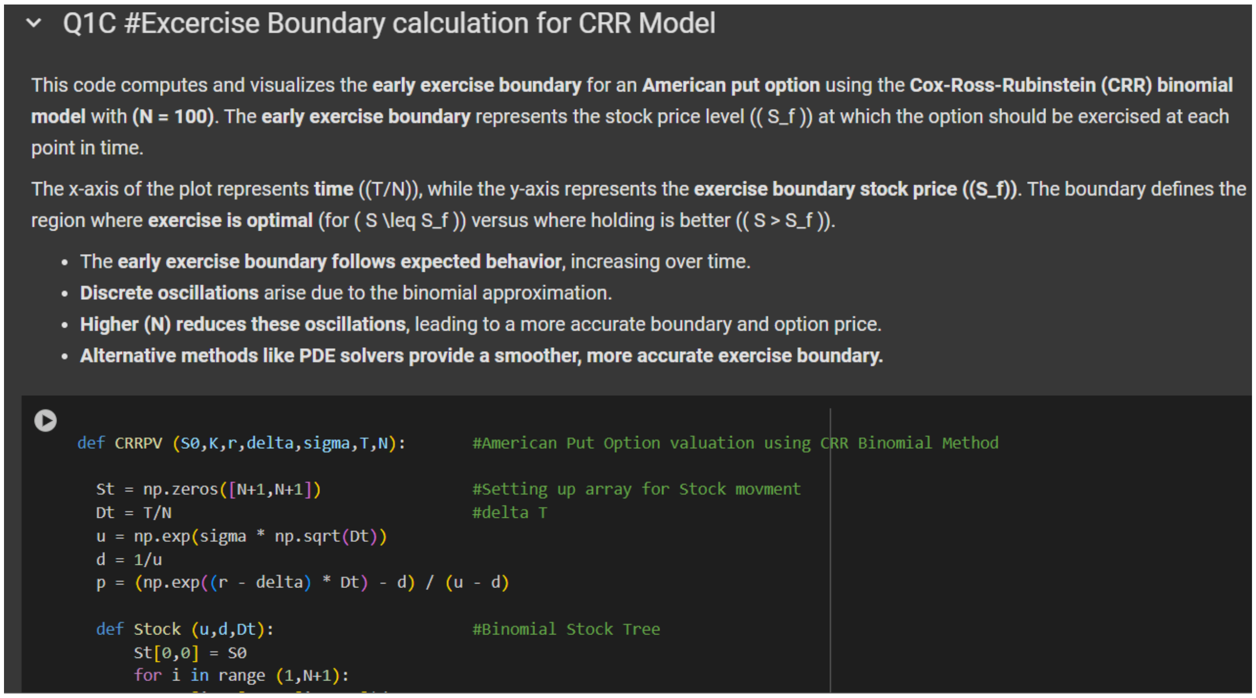


```
for N in N_val:
  putval = BDT(100,105,0.04,0,0.25,1,N)
  optionval.append(putval)
  err_per = ((putval - Benchmark_val)/Benchmark_val)*100
  Err_Per.append(err_per)

optionval = np.array(optionval)
Err_Per = np.array(Err_Per)

for i in range(len(N_val)):
  print(f"N = {N_val[i]}, Option Price = {optionval[i]:.8f}, Error% = {Err_Per[i]:.8f}")

plt.figure(figsize=(10,6))
plt.plot(N_val, Err_Per, color='blue', label="Error %")
plt.xlabel("Time")
plt.ylabel("Error %")
plt.title("Error in Calculation of American Put option with changing values of N for Broadie and Detemple")
plt.legend()
plt.grid(True)
plt.show()
```



Q1.c)

This will reduce the accuracy of option value



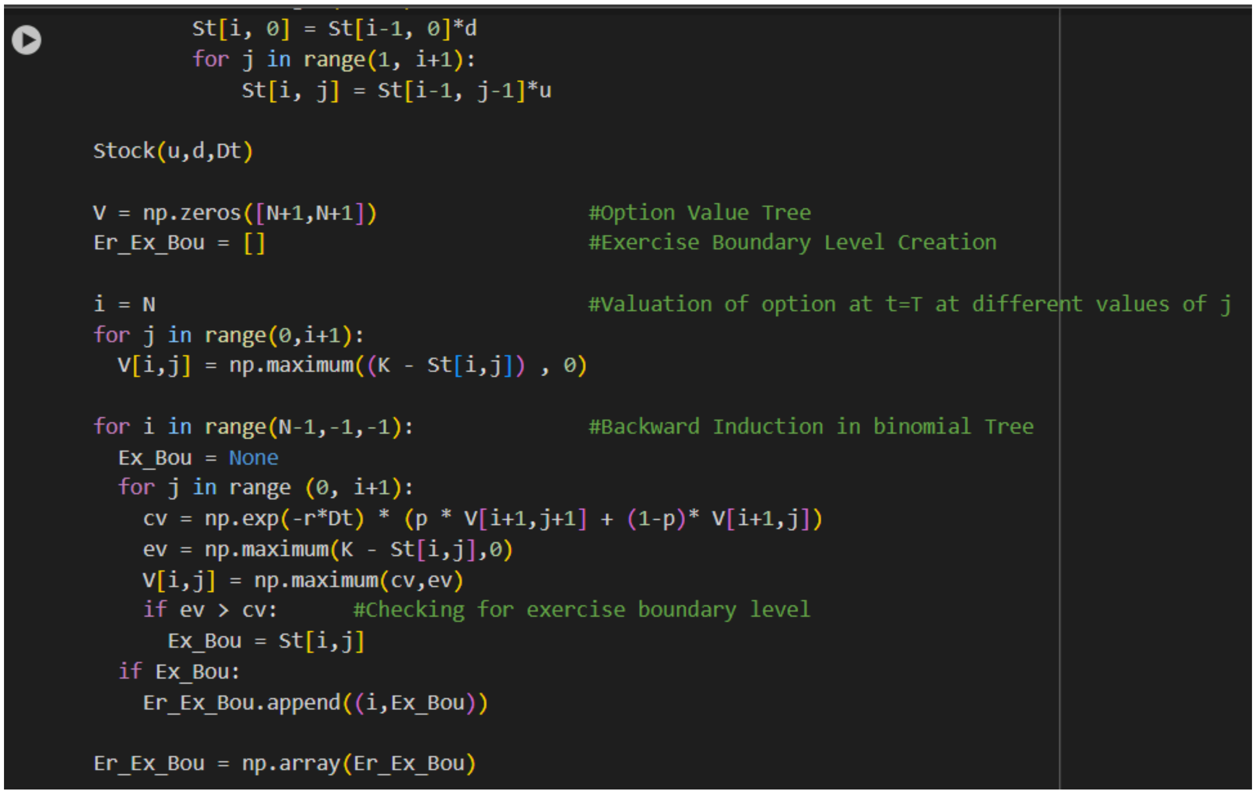
Q1C #Excercise Boundary calculation for CRR Model

This code computes and visualizes the **early exercise boundary** for an **American put option** using the **Cox-Ross-Rubinstein (CRR) binomial model** with **(N = 100)**. The **early exercise boundary** represents the stock price level ((S_f)) at which the option should be exercised at each point in time.

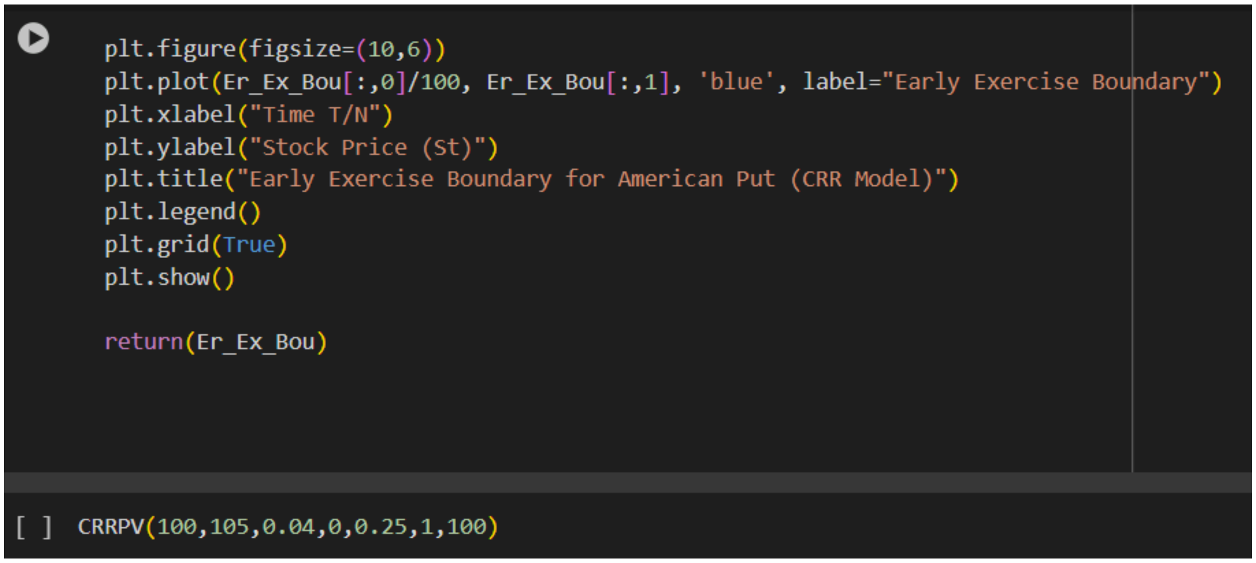
The x-axis of the plot represents **time** ((T/N)), while the y-axis represents the **exercise boundary stock price ((S_f))**. The boundary defines the region where **exercise is optimal** (for (S \leq S_f)) versus where holding is better ((S > S_f)).

- The **early exercise boundary follows expected behavior**, increasing over time.
- **Discrete oscillations** arise due to the binomial approximation.
- **Higher (N) reduces these oscillations**, leading to a more accurate boundary and option price.
- **Alternative methods like PDE solvers provide a smoother, more accurate exercise boundary.**

```python
def CRRPV (S0,K,r,delta,sigma,T,N):       #American Put Option valuation using CRR Binomial Method

  St = np.zeros([N+1,N+1])                #Setting up array for Stock movment
  Dt = T/N                                #delta T
  u = np.exp(sigma * np.sqrt(Dt))
  d = 1/u
  p = (np.exp((r - delta) * Dt) - d) / (u - d)

  def Stock (u,d,Dt):                     #Binomial Stock Tree
      St[0,0] = S0
      for i in range (1,N+1):
```

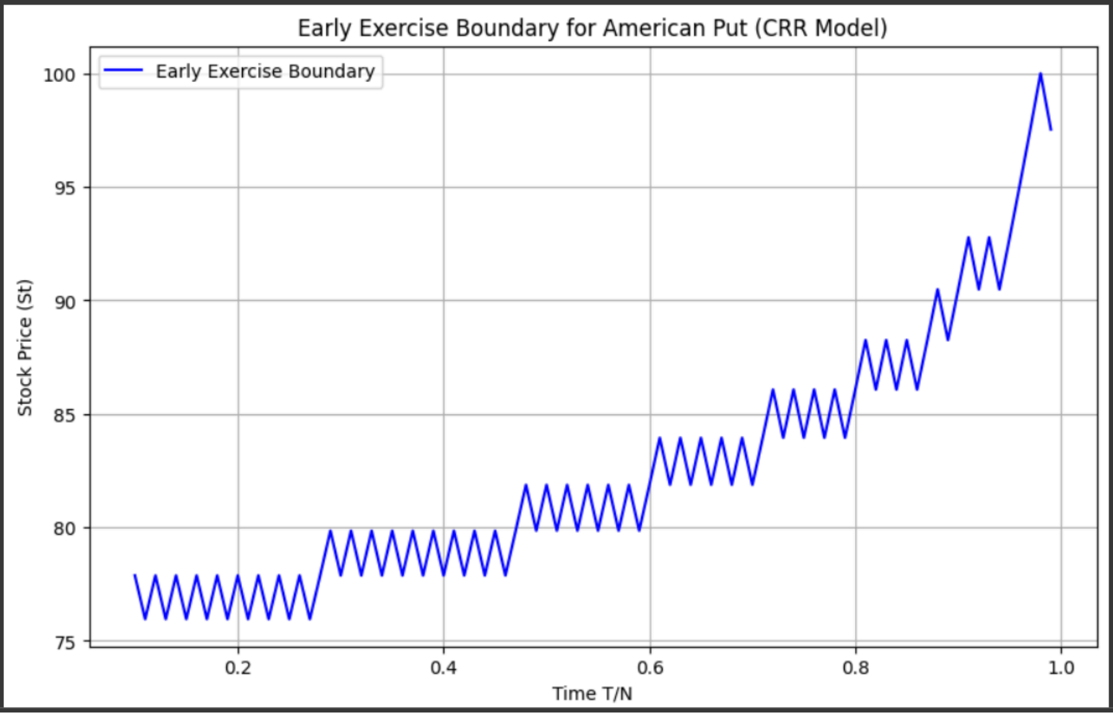


```python
          St[i, 0] = St[i-1, 0]*d
          for j in range(1, i+1):
              St[i, j] = St[i-1, j-1]*u

  Stock(u,d,Dt)

  V = np.zeros([N+1,N+1])                 #Option Value Tree
  Er_Ex_Bou = []                          #Exercise Boundary Level Creation

  i = N                                   #Valuation of option at t=T at different values of j
  for j in range(0,i+1):
    V[i,j] = np.maximum((K - St[i,j]) , 0)

  for i in range(N-1,-1,-1):              #Backward Induction in binomial Tree
    Ex_Bou = None
    for j in range (0, i+1):
      cv = np.exp(-r*Dt) * (p * V[i+1,j+1] + (1-p)* V[i+1,j])
      ev = np.maximum(K - St[i,j],0)
      V[i,j] = np.maximum(cv,ev)
      if ev > cv:      #Checking for exercise boundary level
        Ex_Bou = St[i,j]
    if Ex_Bou:
      Er_Ex_Bou.append((i,Ex_Bou))

  Er_Ex_Bou = np.array(Er_Ex_Bou)
```



```
    plt.figure(figsize=(10,6))
    plt.plot(Er_Ex_Bou[:,0]/100, Er_Ex_Bou[:,1], 'blue', label="Early Exercise Boundary")
    plt.xlabel("Time T/N")
    plt.ylabel("Stock Price (St)")
    plt.title("Early Exercise Boundary for American Put (CRR Model)")
    plt.legend()
    plt.grid(True)
    plt.show()

    return(Er_Ex_Bou)
```

```
[ ] CRRPV(100,105,0.04,0,0.25,1,100)
```

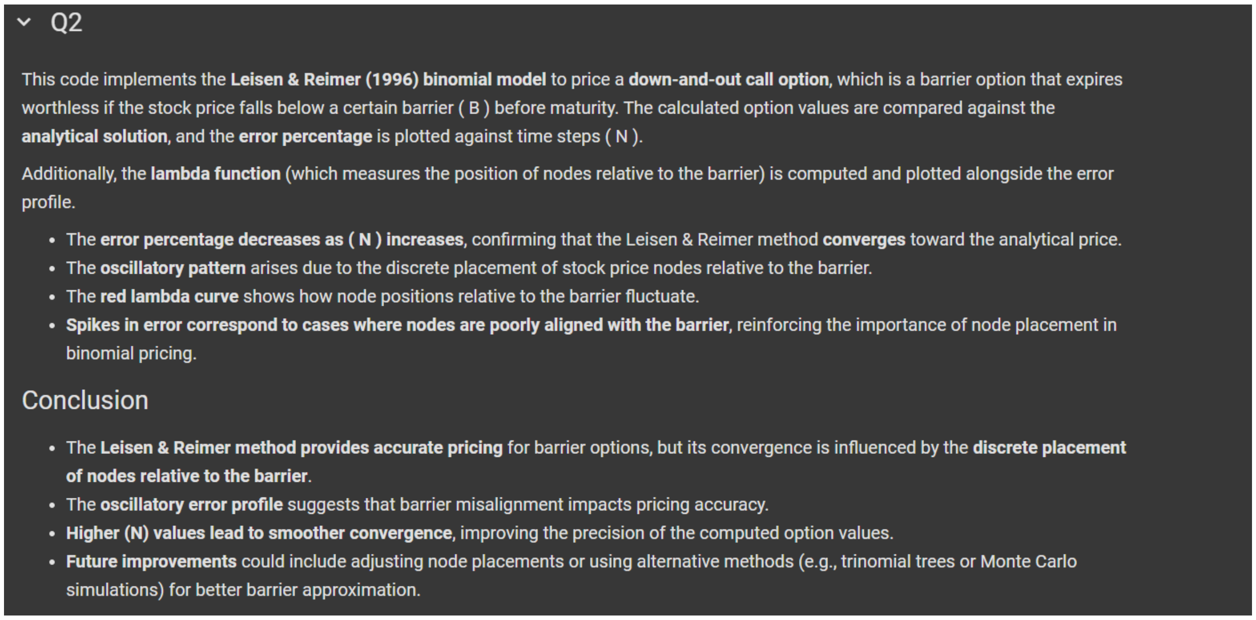


Q2)

a) b) and C) For this as we are using CRR and as per discussion we only used odd values of N to arrive at results, though most the error in valuation arises due to barrie property of the call product

b) Here we see the error oscillate and is lowest when lambda is the highest and also when lambda is near the middle (accounting for N=even part), that we can tell that we get highest error when B is just below Sk leading to large gap between B and Sk-1 leading to overvaluation of the option price and thus an error arises from that, while its lowest when B is at Sk-1 leading to least error

c) To improve the convergence we need to select N such that Sk-1 = B while calculating lambda thus increasing the value of lambda and decreasing error



Q2

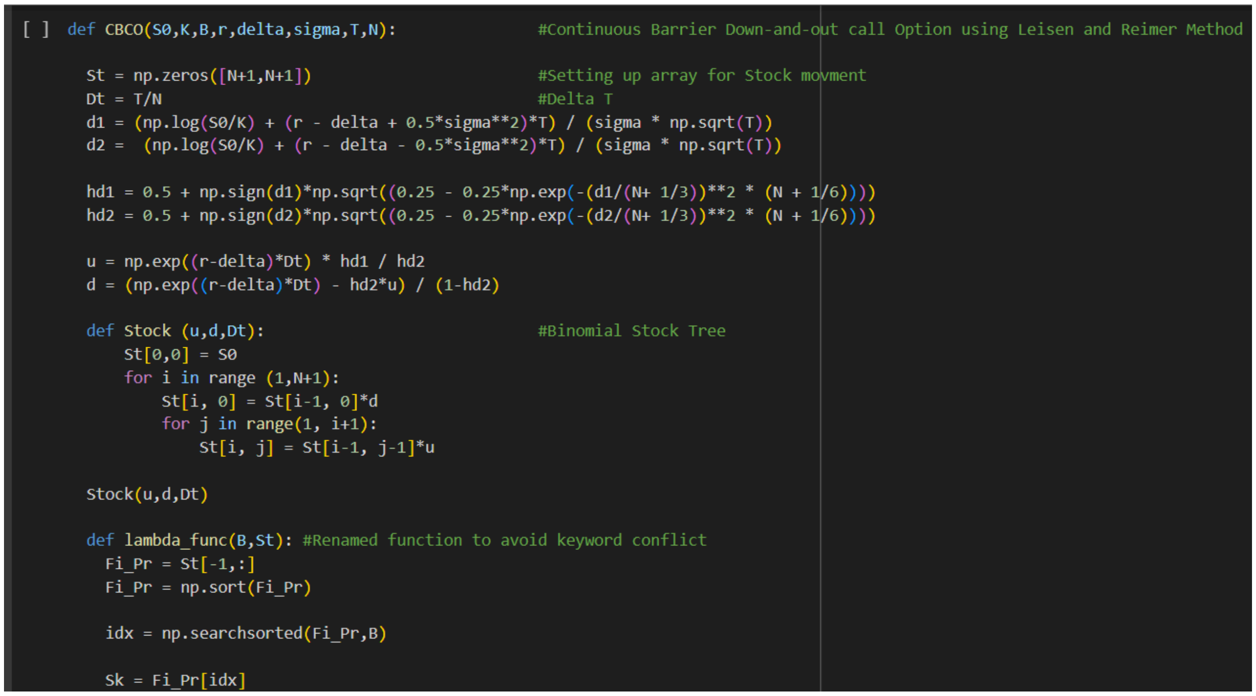
This code implements the **Leisen & Reimer (1996) binomial model** to price a **down-and-out call option**, which is a barrier option that expires worthless if the stock price falls below a certain barrier (B) before maturity. The calculated option values are compared against the **analytical solution**, and the **error percentage** is plotted against time steps (N).

Additionally, the **lambda function** (which measures the position of nodes relative to the barrier) is computed and plotted alongside the error profile.

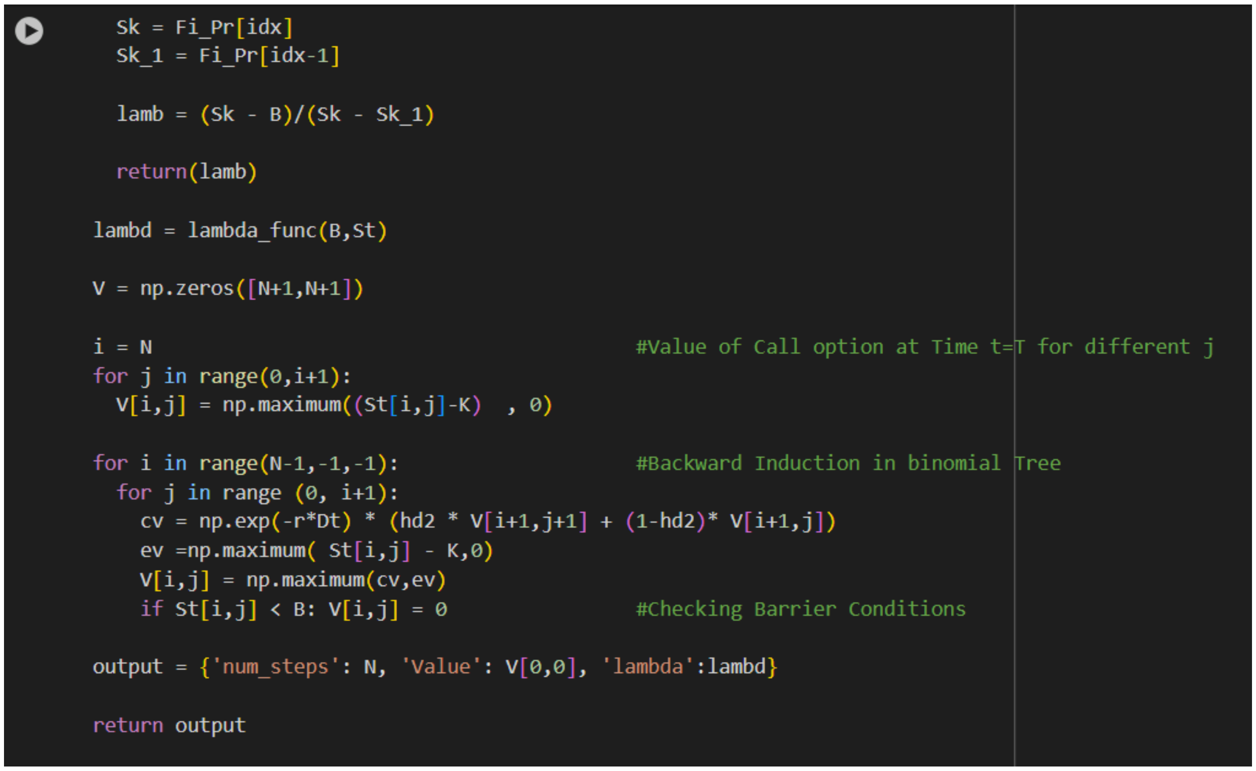
- The **error percentage decreases as (N) increases**, confirming that the Leisen & Reimer method **converges** toward the analytical price.
- The **oscillatory pattern** arises due to the discrete placement of stock price nodes relative to the barrier.
- The **red lambda curve** shows how node positions relative to the barrier fluctuate.
- **Spikes in error correspond to cases where nodes are poorly aligned with the barrier**, reinforcing the importance of node placement in binomial pricing.

Conclusion

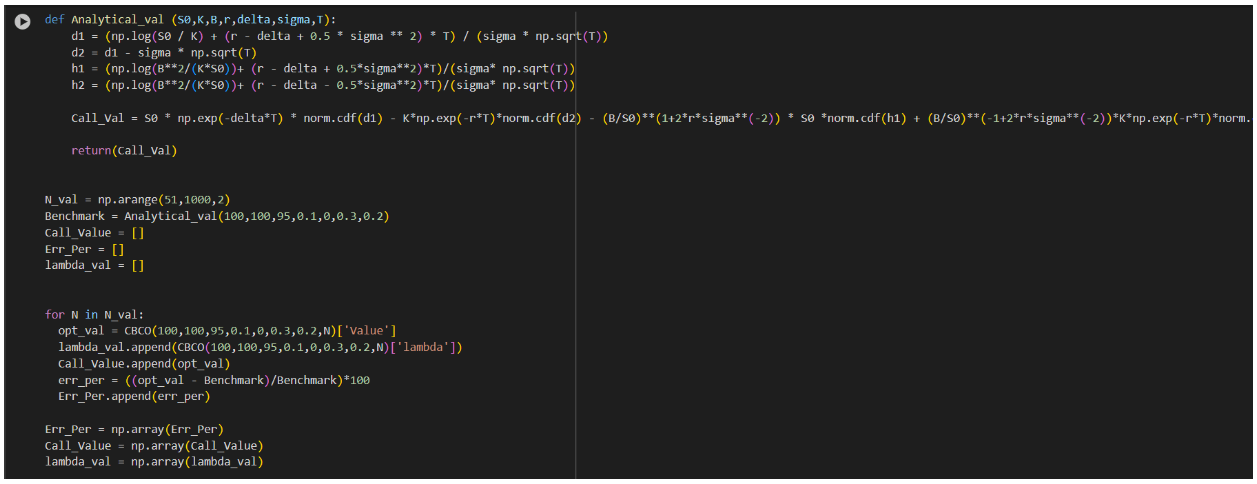
- The **Leisen & Reimer method provides accurate pricing** for barrier options, but its convergence is influenced by the **discrete placement of nodes relative to the barrier**.
- The **oscillatory error profile** suggests that barrier misalignment impacts pricing accuracy.
- **Higher (N) values lead to smoother convergence**, improving the precision of the computed option values.
- **Future improvements** could include adjusting node placements or using alternative methods (e.g., trinomial trees or Monte Carlo simulations) for better barrier approximation.



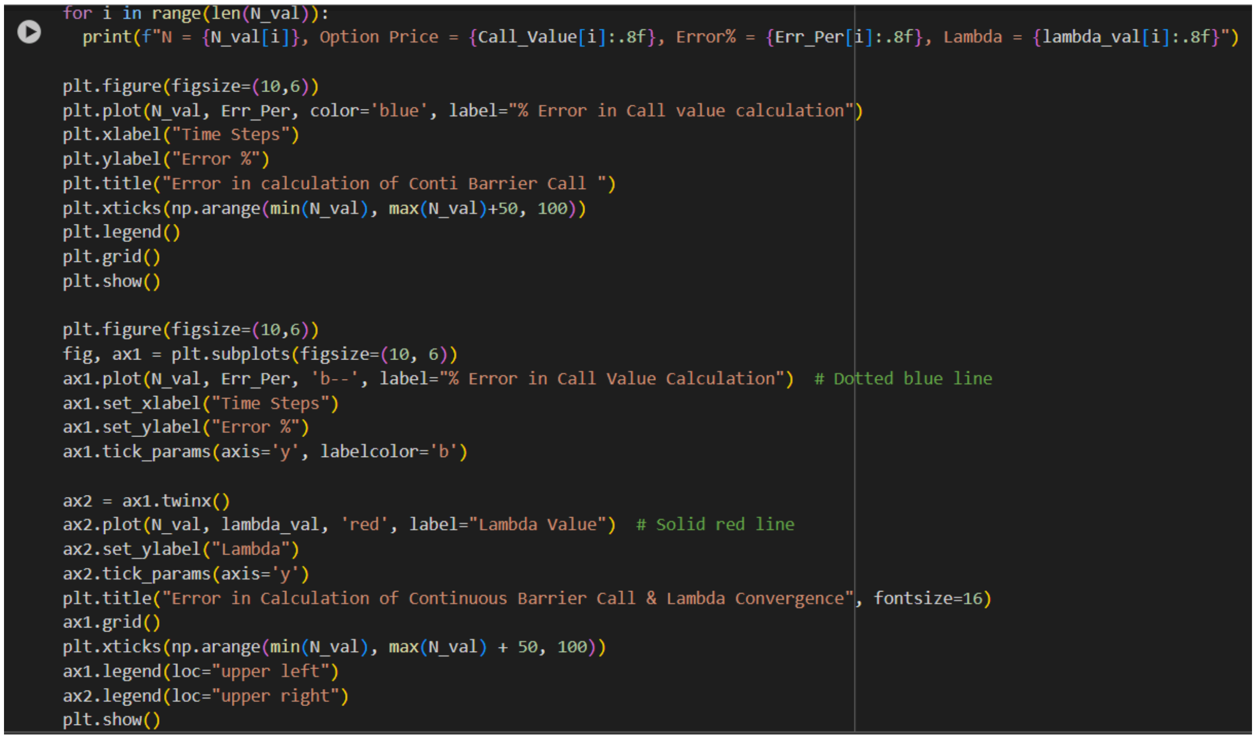
```
[ ] def CBCO(S0,K,B,r,delta,sigma,T,N):             #Continuous Barrier Down-and-out call Option using Leisen and Reimer Method

      St = np.zeros([N+1,N+1])                      #Setting up array for Stock movment
      Dt = T/N                                      #Delta T
      d1 = (np.log(S0/K) + (r - delta + 0.5*sigma**2)*T) / (sigma * np.sqrt(T))
      d2 =  (np.log(S0/K) + (r - delta - 0.5*sigma**2)*T) / (sigma * np.sqrt(T))

      hd1 = 0.5 + np.sign(d1)*np.sqrt((0.25 - 0.25*np.exp(-(d1/(N+ 1/3))**2 * (N + 1/6))))
      hd2 = 0.5 + np.sign(d2)*np.sqrt((0.25 - 0.25*np.exp(-(d2/(N+ 1/3))**2 * (N + 1/6))))

      u = np.exp((r-delta)*Dt) * hd1 / hd2
      d = (np.exp((r-delta)*Dt) - hd2*u) / (1-hd2)

      def Stock (u,d,Dt):                           #Binomial Stock Tree
          St[0,0] = S0
          for i in range (1,N+1):
              St[i, 0] = St[i-1, 0]*d
              for j in range(1, i+1):
                  St[i, j] = St[i-1, j-1]*u

      Stock(u,d,Dt)

      def lambda_func(B,St): #Renamed function to avoid keyword conflict
        Fi_Pr = St[-1,:]
        Fi_Pr = np.sort(Fi_Pr)

        idx = np.searchsorted(Fi_Pr,B)

        Sk = Fi_Pr[idx]
```



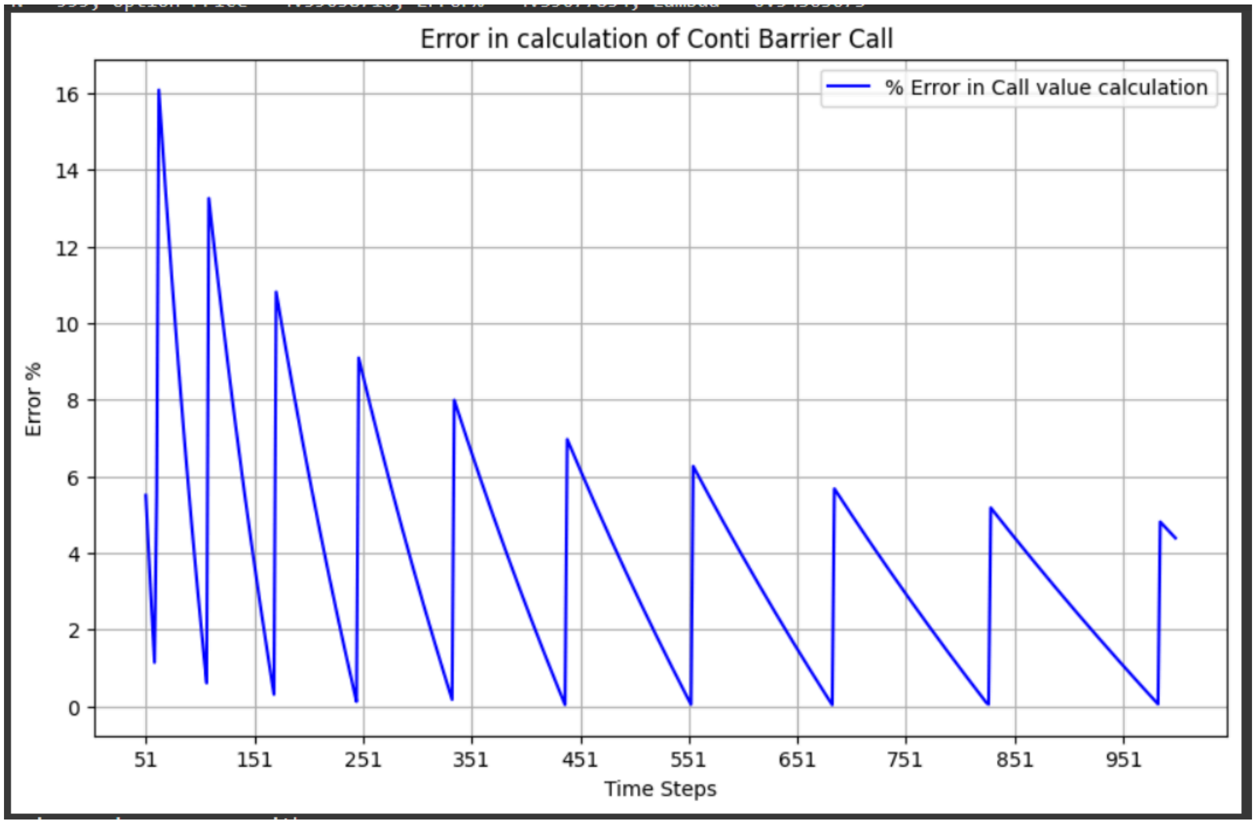
```
      Sk = Fi_Pr[idx]
      Sk_1 = Fi_Pr[idx-1]

      lamb = (Sk - B)/(Sk - Sk_1)

      return(lamb)

    lambd = lambda_func(B,St)

    V = np.zeros([N+1,N+1])

    i = N                                        #Value of Call option at Time t=T for different j
    for j in range(0,i+1):
      V[i,j] = np.maximum((St[i,j]-K)  , 0)

    for i in range(N-1,-1,-1):                   #Backward Induction in binomial Tree
      for j in range (0, i+1):
        cv = np.exp(-r*Dt) * (hd2 * V[i+1,j+1] + (1-hd2)* V[i+1,j])
        ev =np.maximum( St[i,j] - K,0)
        V[i,j] = np.maximum(cv,ev)
        if St[i,j] < B: V[i,j] = 0               #Checking Barrier Conditions

    output = {'num_steps': N, 'Value': V[0,0], 'lambda':lambd}

    return output
```

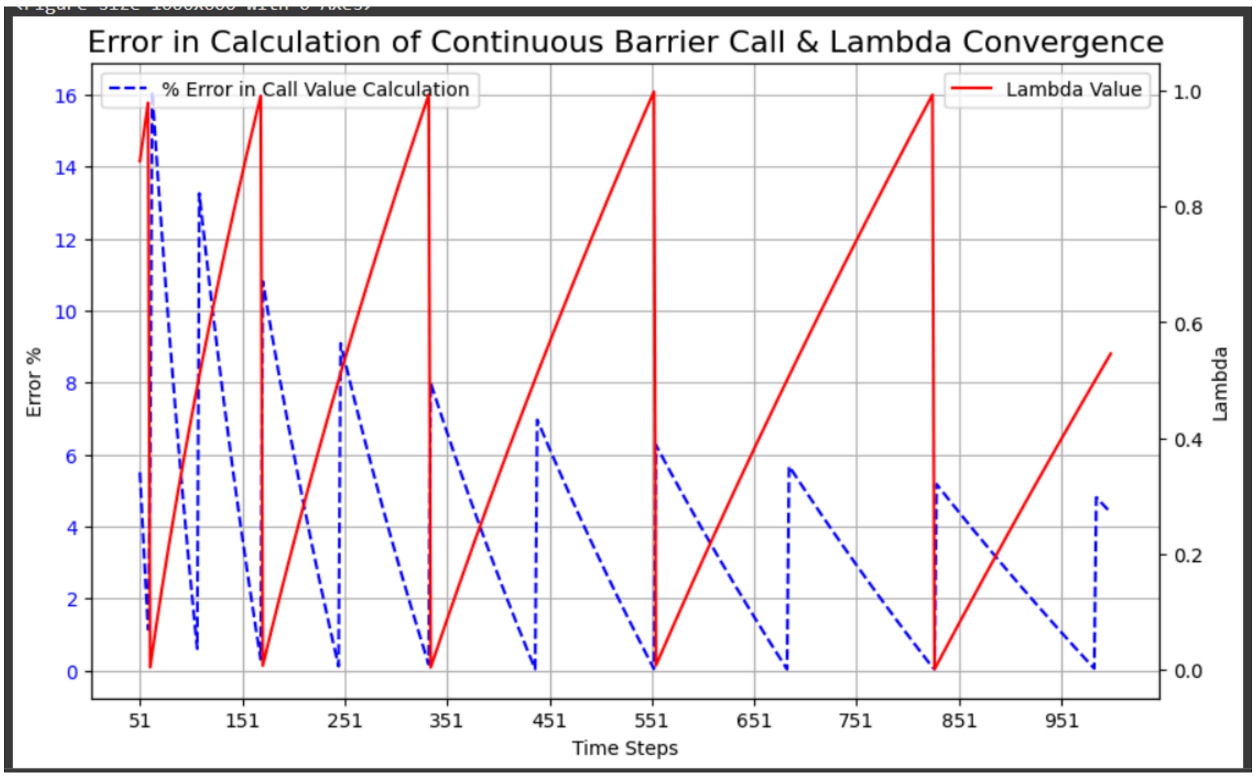


```
def Analytical_val (S0,K,B,r,delta,sigma,T):
    d1 = (np.log(S0 / K) + (r - delta + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    h1 = (np.log(B**2/(K*S0))+ (r - delta + 0.5*sigma**2)*T)/(sigma* np.sqrt(T))
    h2 = (np.log(B**2/(K*S0))+ (r - delta - 0.5*sigma**2)*T)/(sigma* np.sqrt(T))

    Call_Val = S0 * np.exp(-delta*T) * norm.cdf(d1) - K*np.exp(-r*T)*norm.cdf(d2) - (B/S0)**(1+2*r*sigma**(-2)) * S0 *norm.cdf(h1) + (B/S0)**(-1+2*r*sigma**(-2))*K*np.exp(-r*T)*norm.

    return(Call_Val)


N_val = np.arange(51,1000,2)
Benchmark = Analytical_val(100,100,95,0.1,0,0.3,0.2)
Call_Value = []
Err_Per = []
lambda_val = []


for N in N_val:
  opt_val = CBCO(100,100,95,0.1,0,0.3,0.2,N)['Value']
  lambda_val.append(CBCO(100,100,95,0.1,0,0.3,0.2,N)['lambda'])
  Call_Value.append(opt_val)
  err_per = ((opt_val - Benchmark)/Benchmark)*100
  Err_Per.append(err_per)

Err_Per = np.array(Err_Per)
Call_Value = np.array(Call_Value)
lambda_val = np.array(lambda_val)
```



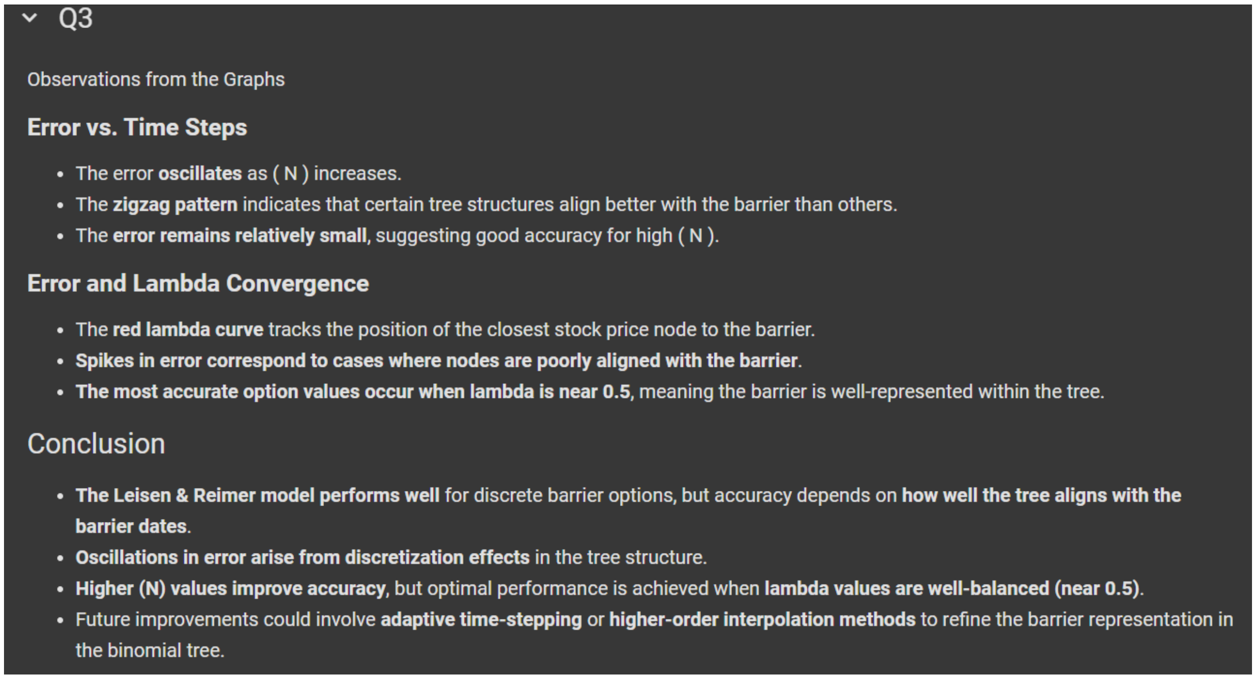
```python
for i in range(len(N_val)):
  print(f"N = {N_val[i]}, Option Price = {Call_Value[i]:.8f}, Error% = {Err_Per[i]:.8f}, Lambda = {lambda_val[i]:.8f}")

plt.figure(figsize=(10,6))
plt.plot(N_val, Err_Per, color='blue', label="% Error in Call value calculation")
plt.xlabel("Time Steps")
plt.ylabel("Error %")
plt.title("Error in calculation of Conti Barrier Call ")
plt.xticks(np.arange(min(N_val), max(N_val)+50, 100))
plt.legend()
plt.grid()
plt.show()

plt.figure(figsize=(10,6))
fig, ax1 = plt.subplots(figsize=(10, 6))
ax1.plot(N_val, Err_Per, 'b--', label="% Error in Call Value Calculation")  # Dotted blue line
ax1.set_xlabel("Time Steps")
ax1.set_ylabel("Error %")
ax1.tick_params(axis='y', labelcolor='b')

ax2 = ax1.twinx()
ax2.plot(N_val, lambda_val, 'red', label="Lambda Value")  # Solid red line
ax2.set_ylabel("Lambda")
ax2.tick_params(axis='y')
plt.title("Error in Calculation of Continuous Barrier Call & Lambda Convergence", fontsize=16)
ax1.grid()
plt.xticks(np.arange(min(N_val), max(N_val) + 50, 100))
ax1.legend(loc="upper left")
ax2.legend(loc="upper right")
plt.show()
```





Q3)

A3) Error is lowest when lamda is 0.5 for the given oscillation, though the error oscillation decreases as we increase N, most of the error in this is also arising because of barrier property of call



Q3

Observations from the Graphs

Error vs. Time Steps

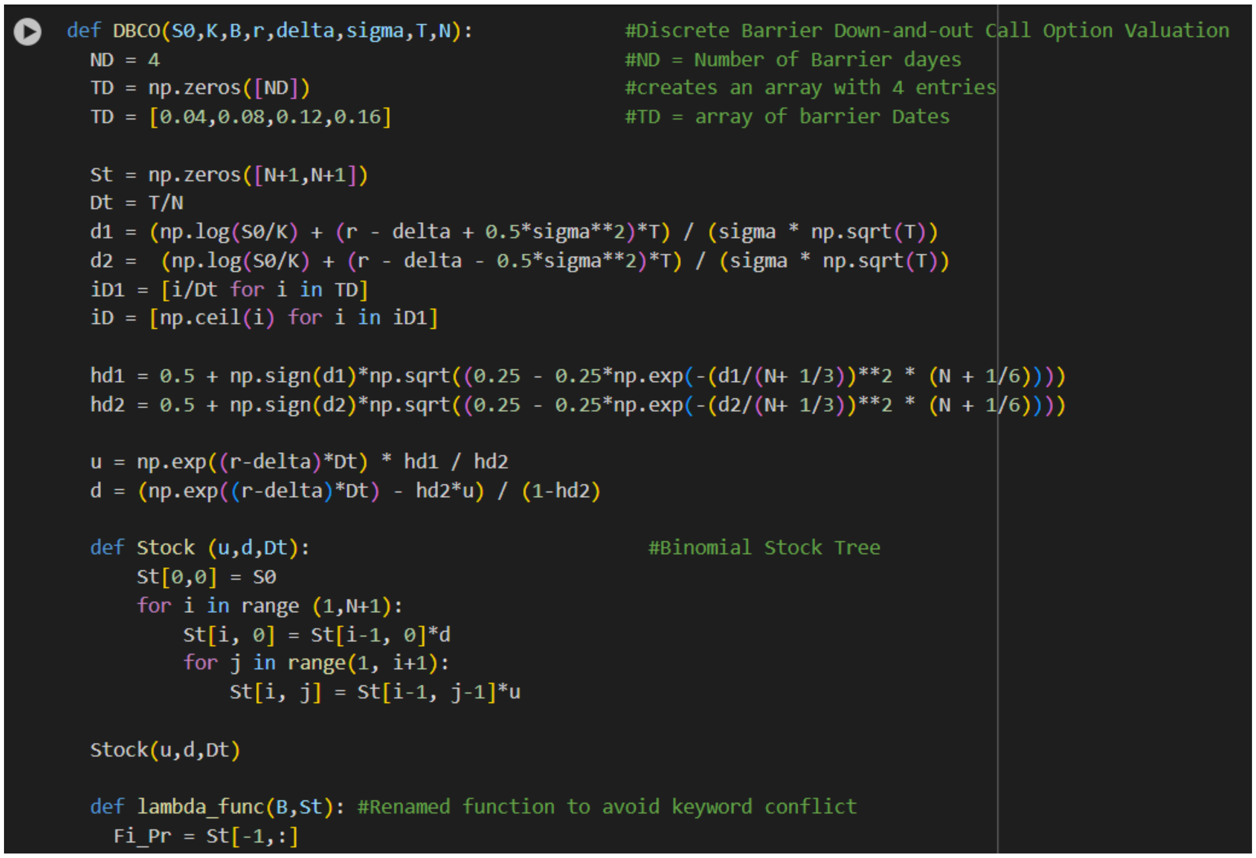
- The error **oscillates** as (N) increases.
- The **zigzag pattern** indicates that certain tree structures align better with the barrier than others.
- The **error remains relatively small**, suggesting good accuracy for high (N).

Error and Lambda Convergence

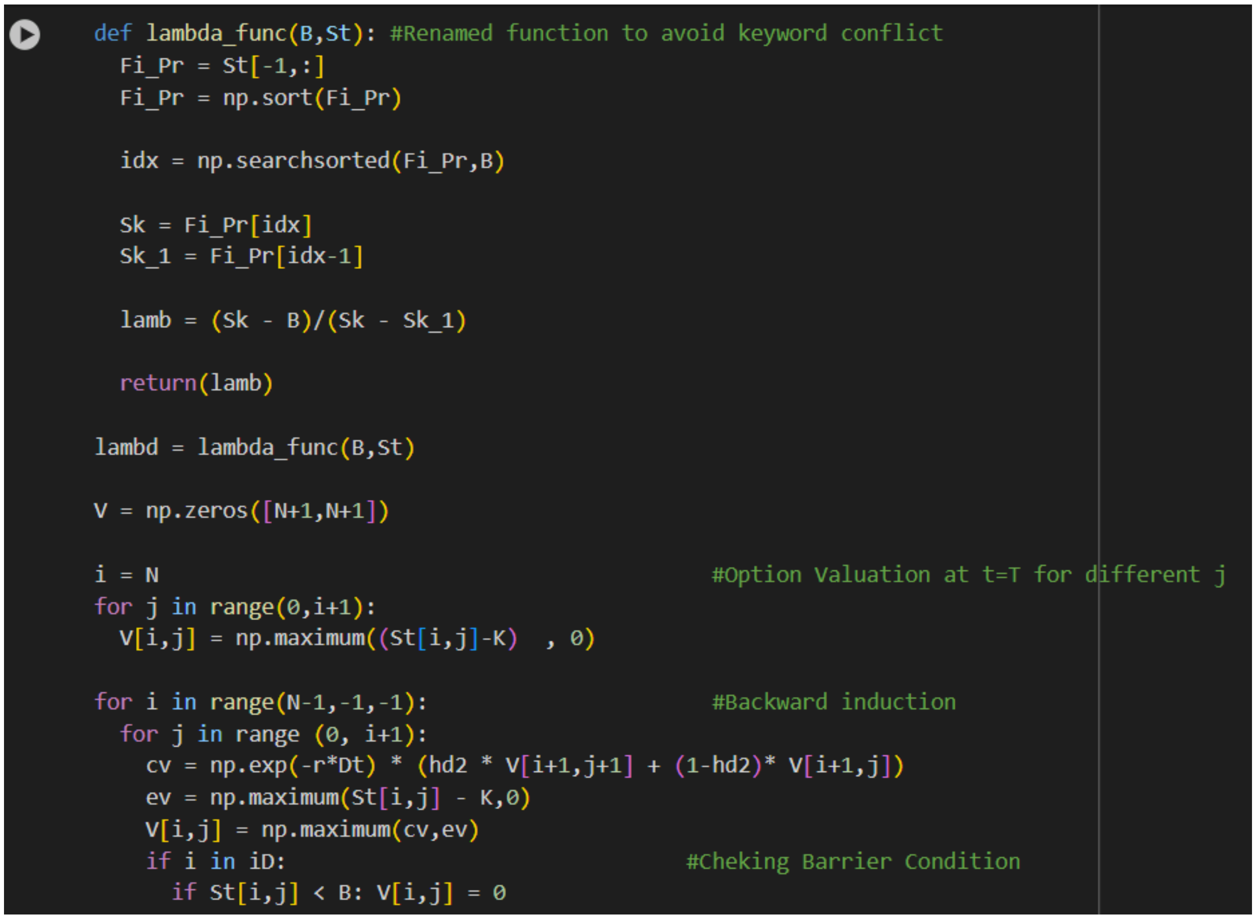
- The **red lambda curve** tracks the position of the closest stock price node to the barrier.
- **Spikes in error correspond to cases where nodes are poorly aligned with the barrier**.
- **The most accurate option values occur when lambda is near 0.5**, meaning the barrier is well-represented within the tree.

Conclusion

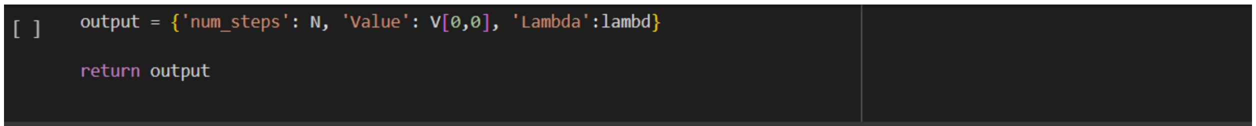
- **The Leisen & Reimer model performs well** for discrete barrier options, but accuracy depends on **how well the tree aligns with the barrier dates**.
- **Oscillations in error arise from discretization effects** in the tree structure.
- **Higher (N) values improve accuracy**, but optimal performance is achieved when **lambda values are well-balanced (near 0.5)**.
- Future improvements could involve **adaptive time-stepping** or **higher-order interpolation methods** to refine the barrier representation in the binomial tree.



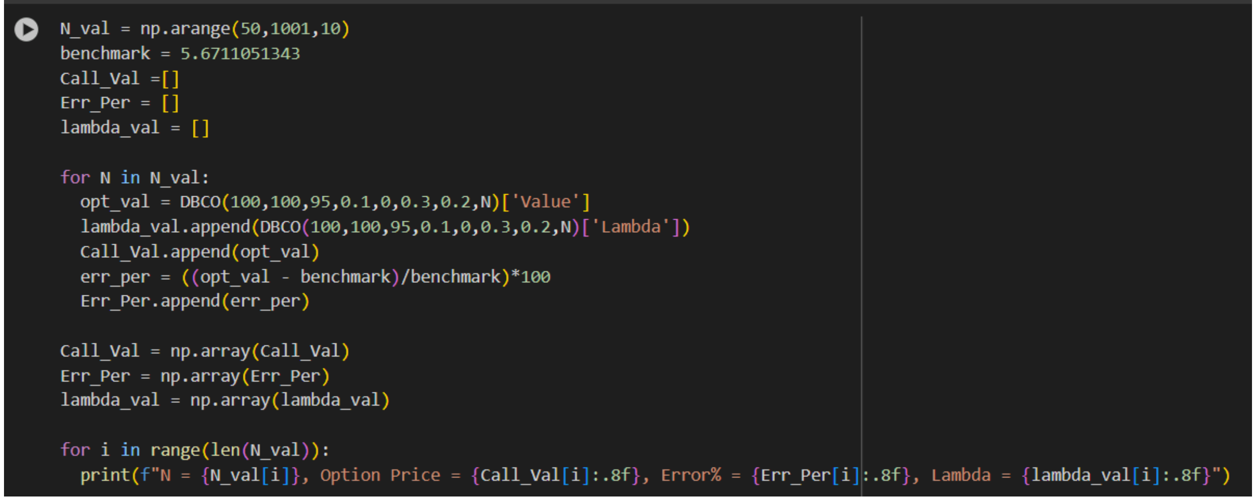
```python
def DBCO(S0,K,B,r,delta,sigma,T,N):                #Discrete Barrier Down-and-out Call Option Valuation
  ND = 4                                            #ND = Number of Barrier dayes
  TD = np.zeros([ND])                               #creates an array with 4 entries
  TD = [0.04,0.08,0.12,0.16]                        #TD = array of barrier Dates

  St = np.zeros([N+1,N+1])
  Dt = T/N
  d1 = (np.log(S0/K) + (r - delta + 0.5*sigma**2)*T) / (sigma * np.sqrt(T))
  d2 =  (np.log(S0/K) + (r - delta - 0.5*sigma**2)*T) / (sigma * np.sqrt(T))
  iD1 = [i/Dt for i in TD]
  iD = [np.ceil(i) for i in iD1]

  hd1 = 0.5 + np.sign(d1)*np.sqrt((0.25 - 0.25*np.exp(-(d1/(N+ 1/3))**2 * (N + 1/6))))
  hd2 = 0.5 + np.sign(d2)*np.sqrt((0.25 - 0.25*np.exp(-(d2/(N+ 1/3))**2 * (N + 1/6))))

  u = np.exp((r-delta)*Dt) * hd1 / hd2
  d = (np.exp((r-delta)*Dt) - hd2*u) / (1-hd2)

  def Stock (u,d,Dt):                                 #Binomial Stock Tree
      St[0,0] = S0
      for i in range (1,N+1):
          St[i, 0] = St[i-1, 0]*d
          for j in range(1, i+1):
              St[i, j] = St[i-1, j-1]*u

  Stock(u,d,Dt)

  def lambda_func(B,St): #Renamed function to avoid keyword conflict
    Fi_Pr = St[-1,:]
```



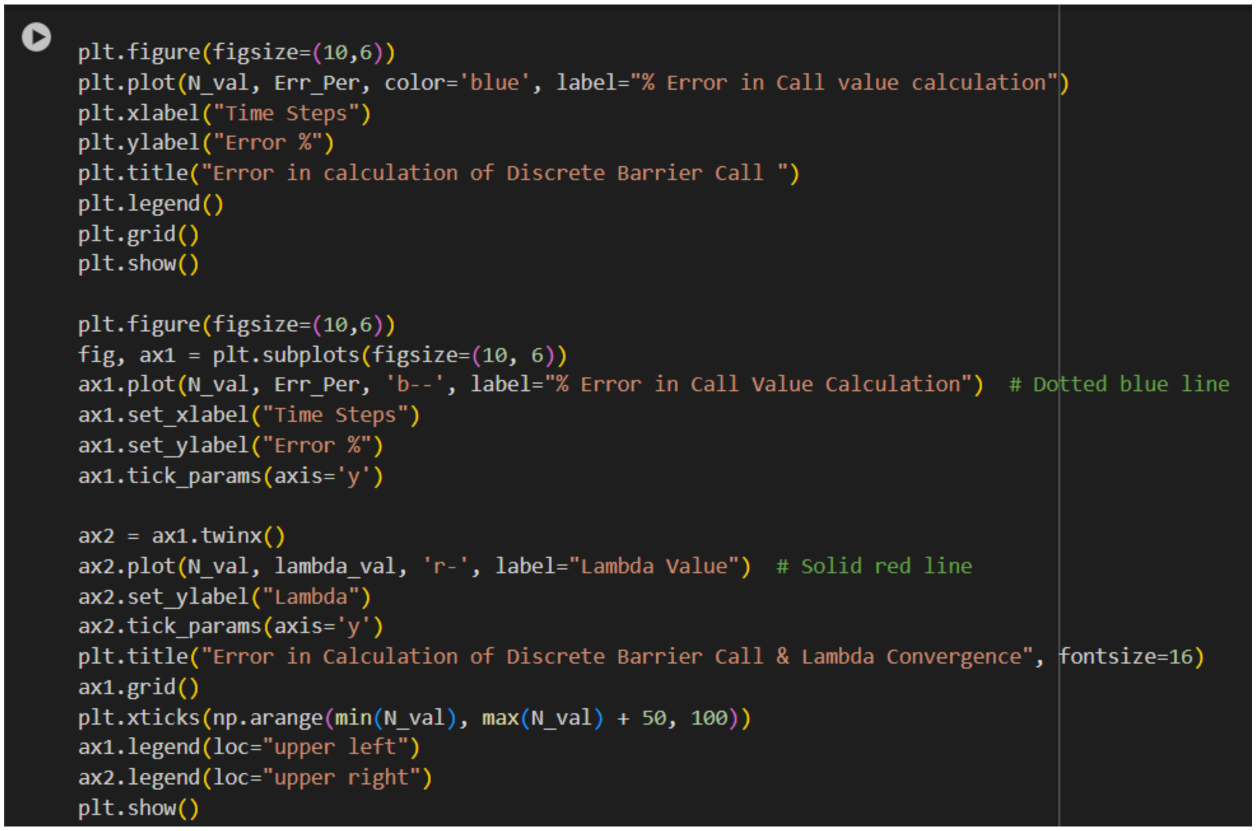
```python
  def lambda_func(B,St): #Renamed function to avoid keyword conflict
    Fi_Pr = St[-1,:]
    Fi_Pr = np.sort(Fi_Pr)

    idx = np.searchsorted(Fi_Pr,B)

    Sk = Fi_Pr[idx]
    Sk_1 = Fi_Pr[idx-1]

    lamb = (Sk - B)/(Sk - Sk_1)

    return(lamb)

  lambd = lambda_func(B,St)

  V = np.zeros([N+1,N+1])

  i = N                                           #Option Valuation at t=T for different j
  for j in range(0,i+1):
    V[i,j] = np.maximum((St[i,j]-K)  , 0)

  for i in range(N-1,-1,-1):                      #Backward induction
    for j in range (0, i+1):
      cv = np.exp(-r*Dt) * (hd2 * V[i+1,j+1] + (1-hd2)* V[i+1,j])
      ev = np.maximum(St[i,j] - K,0)
      V[i,j] = np.maximum(cv,ev)
      if i in iD:                               #Cheking Barrier Condition
        if St[i,j] < B: V[i,j] = 0
```



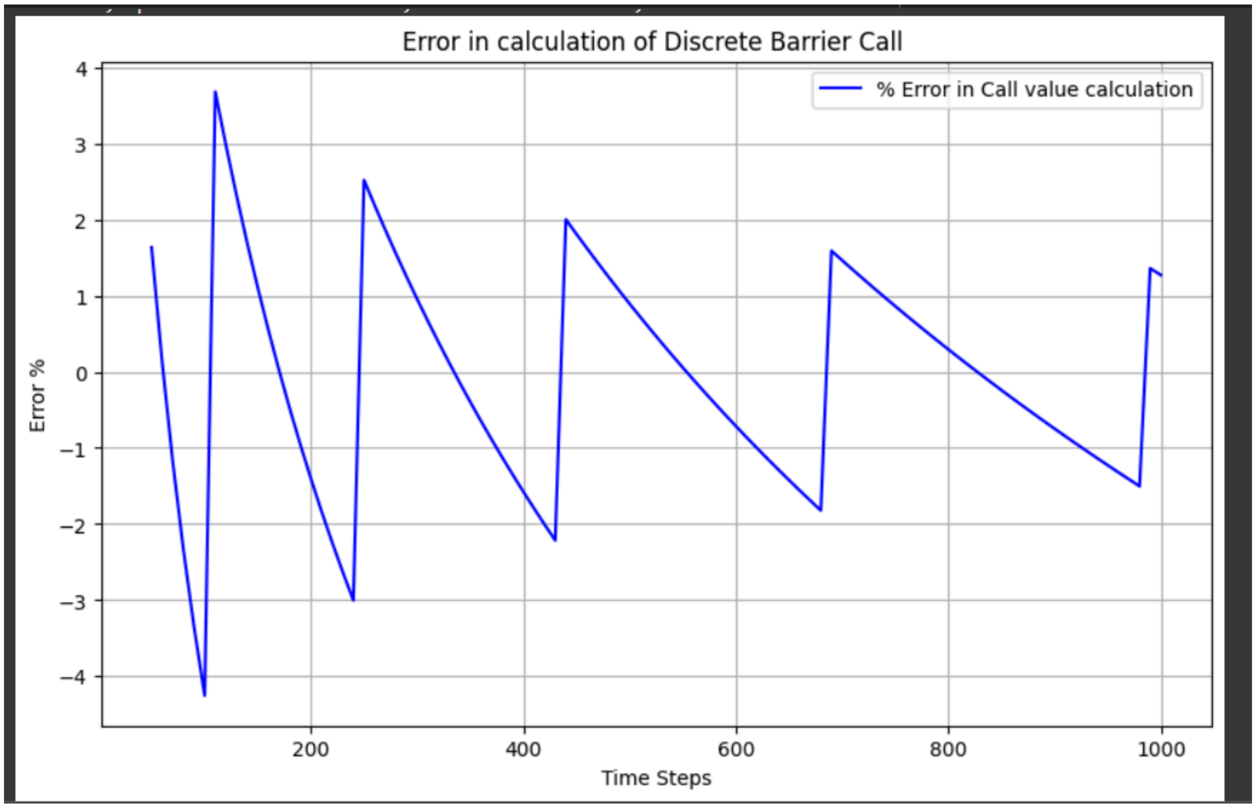
```python
[ ]   output = {'num_steps': N, 'Value': V[0,0], 'Lambda':lambd}

      return output
```

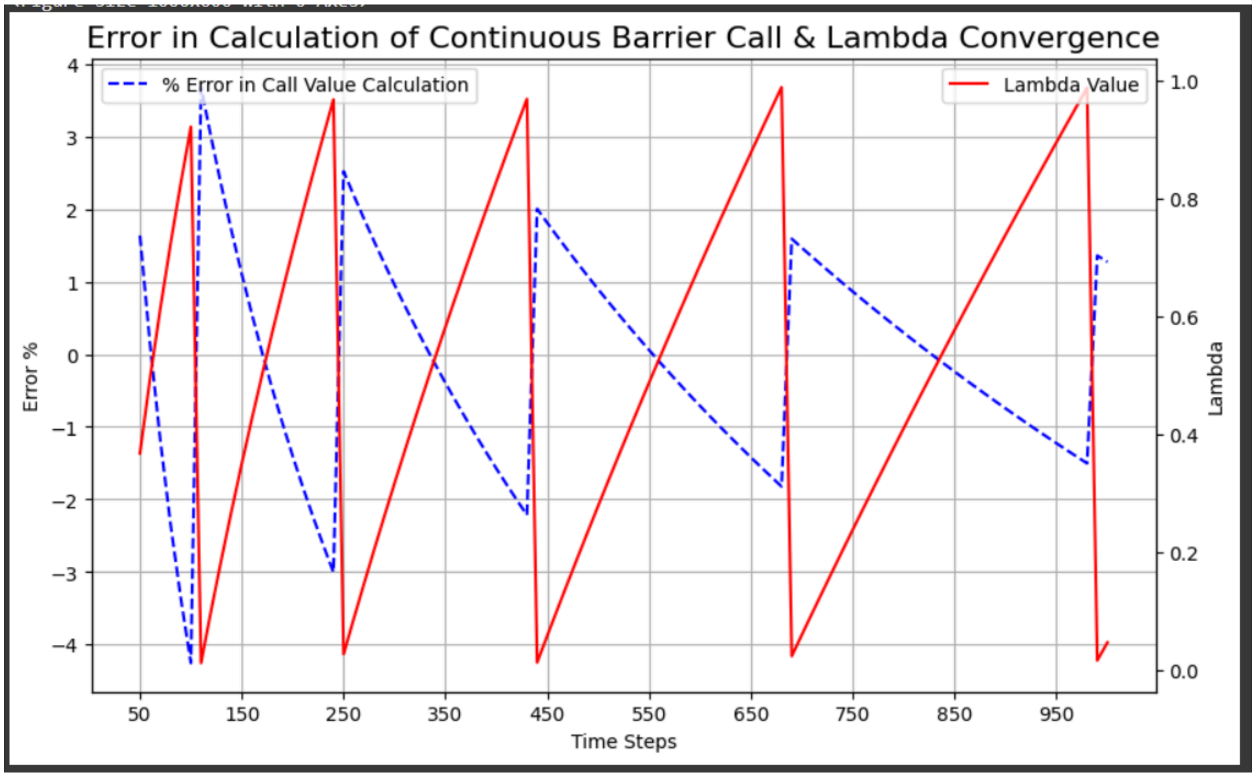


```python
N_val = np.arange(50,1001,10)
benchmark = 5.6711051343
Call_Val =[]
Err_Per = []
lambda_val = []

for N in N_val:
  opt_val = DBCO(100,100,95,0.1,0,0.3,0.2,N)['Value']
  lambda_val.append(DBCO(100,100,95,0.1,0,0.3,0.2,N)['Lambda'])
  Call_Val.append(opt_val)
  err_per = ((opt_val - benchmark)/benchmark)*100
  Err_Per.append(err_per)

Call_Val = np.array(Call_Val)
Err_Per = np.array(Err_Per)
lambda_val = np.array(lambda_val)

for i in range(len(N_val)):
  print(f"N = {N_val[i]}, Option Price = {Call_Val[i]:.8f}, Error% = {Err_Per[i]:.8f}, Lambda = {lambda_val[i]:.8f}")
```



```python
plt.figure(figsize=(10,6))
plt.plot(N_val, Err_Per, color='blue', label="% Error in Call value calculation")
plt.xlabel("Time Steps")
plt.ylabel("Error %")
plt.title("Error in calculation of Discrete Barrier Call ")
plt.legend()
plt.grid()
plt.show()

plt.figure(figsize=(10,6))
fig, ax1 = plt.subplots(figsize=(10, 6))
ax1.plot(N_val, Err_Per, 'b--', label="% Error in Call Value Calculation")  # Dotted blue line
ax1.set_xlabel("Time Steps")
ax1.set_ylabel("Error %")
ax1.tick_params(axis='y')

ax2 = ax1.twinx()
ax2.plot(N_val, lambda_val, 'r-', label="Lambda Value")  # Solid red line
ax2.set_ylabel("Lambda")
ax2.tick_params(axis='y')
plt.title("Error in Calculation of Discrete Barrier Call & Lambda Convergence", fontsize=16)
ax1.grid()
plt.xticks(np.arange(min(N_val), max(N_val) + 50, 100))
ax1.legend(loc="upper left")
ax2.legend(loc="upper right")
plt.show()
```




Error in Calculation of Continuous Barrier Call & Lambda Convergence
% Error in Call Value Calculation
Lambda Value
Error %
Lambda
Time Steps
4
3
2
1
0
−1
−2
−3
−4
1.0
0.8
0.6
0.4
0.2
0.0
50
150
250
350
450
550
650
750
850
950