Question 1)

**Code:**

1. # importing the necessary packages
2. **import** numpy as np
3. **import** scipy.linalg as linalg
5. ##############################################################################
7. **def** tridiagonalSolver(A, b):
8. # check if input matrix is of correct form
9. **if** (len(A) != len(b) **or**
10. np.any(A[2:,0] != 0) **or**
11. np.any(A[:-2,-1] != 0)):
12. **print**("Incorrect tridiagonal structure.")
13. **return**
15. Arows = len(A)
16. A = A.astype(float)
17. b = b.astype(float)
19. # making A into lower triangular matrix
20. **for** i **in** range(Arows-1, 0, -1):
21. b[i-1] = b[i-1] - b[i]\*A[i-1][i]/A[i][i]
22. A[i-1] = A[i-1] - A[i]\*A[i-1][i]/A[i][i]
24. # gauss elimination solving
25. # note, np.linalg.solve() also will work
26. x = np.zeros(Arows)
27. k = Arows-1
28. x[k] = b[k]/A[k][k]
29. **while** k >= 0:
30. x[k] = (b[k] - np.dot(A[k,k+1:],x[k+1:]))/A[k,k]
31. k = k-1
32. **return** x
34. **def** generalizedSolver(A, b):
35. # check if input matrix is of correct form
36. **if** (len(A) != len(b) **or**
37. np.any(A[2:,0] != 0) **or**
38. np.any(A[:-2,-1] != 0)):
39. **print**("Incorrect tridiagonal structure.")
40. **return**
42. A = A.astype(float)
43. b = b.astype(float)
45. # first getting rid of special delta value and epsilon value
46. b[0] = b[0] - b[1]\*A[0][2]/A[1][2]
47. b[-1] = b[-1] - b[-2]\*A[-1][-3]/A[-2][-3]
49. A[0] = A[0] - A[1]\*A[0][2]/A[1][2]
50. A[-1] = A[-1] - A[-2]\*A[-1][-3]/A[-2][-3]
52. # then run same tridiagonal solver on new matrix
53. x = tridiagonalSolver(A, b)
54. **return** x
56. **def** betterSolver(A, b):
57. # check if input matrix is of correct form
58. **if** (len(A) != len(b) **or**
59. np.any(A[2:,0] != 0) **or**
60. np.any(A[:-2,-1] != 0)):
61. **print**("Incorrect tridiagonal structure.")
62. **return**
64. # perform L (lower triangular) and U (upper triangular) decomposition
65. P,L,U = linalg.lu(A)
67. # solve
68. x = linalg.solve(L, b)
69. v = linalg.solve(U, x)
70. **return** v

Question 2) parts a,b,c,d

**Code:**

1. # importing the necessary packages
2. **import** numpy as np
3. **from** numpy **import** sqrt, exp, log
4. **from** scipy.stats **import** norm
6. ##############################################################################
8. # initial vars
9. S0 = 100.
10. sigma = 0.20
11. r = 0.03
12. T = 1
13. K = 115.
15. # on interval [0, 200]
16. # assume Smax = 2\*S0
17. Smax = 200.
18. ds = Smax/1000.      # space step
19. M = 1000
20. timestep = 20.  # change this according to question part
21. dt = T/timestep  # time step
22. N = int(T/dt)
24. # stock steps
25. s = np.array([ds\*j **for** j **in** range(M)])
27. # initial v
28. v = np.maximum(s - K, 0)
30. # regular alpha, beta, gamma matrix A version
31. alpha = 0.5  \* (sigma\*\*2\*s\*\*2/ds\*\*2 - r\*s/ds)
32. beta = -sigma\*\*2\*s\*\*2/ds\*\*2 - r
33. gamma = 0.5  \* (sigma\*\*2\*s\*\*2/ds\*\*2 + r\*s/ds)
35. # setting up matrix A
36. A = np.diag(beta) + np.diag(alpha[1:], -1) + np.diag(gamma[0:M-1], 1)
38. # boundary conditions for matrix A
39. A[0, :] = 0     # initial is 0
40. A[M-1, :] = 0   # linearity boundary
41. A[M-1, -3:] = np.array([1,-2,1])
43. # identity matrix
44. I = np.identity(1000)
46. # setting up main big matrix A\_tilda for price equation
47. A\_tilda = I + dt\*A
49. # EXPLICIT SCHEME
50. **for** i **in** range(N):
51. v = np.dot(A, v)
53. # closed form BS formula price
54. d1 = ( log(S0/K) + (r + (sigma\*\*2)/2)\*T ) / ( sigma\*sqrt(T) )
55. d2 = d1 - sigma\*sqrt(T)
56. BSCallPrice = S0\*norm.cdf(d1) - K\*exp(-r\*T)\*norm.cdf(d2)
57. **print**("Call Price explicitly via BS model formula = " + str(round(BSCallPrice,2)))
58. **print**("-------------------------------------------------")
60. **print**("Explicit finite difference method estimated price (for timestep = " + str(timestep) + ") is:")
61. **print**(v[(M+1)/2])

**Output:**

Call Price explicitly via BS model formula = 3.86

-------------------------------------------------

Explicit finite difference method estimated price (for timestep = 20.0) is:

0.0

Explicit finite difference method estimated price (for timestep = 50.0) is:

0.0

Explicit finite difference method estimated price (for timestep = 100.0) is:

Nan

Explicit finite difference method estimated price (for timestep = 500.0) is:

Nan

My implementation for Q2 would not give me any price result for the explicit finite difference scheme method.

This is perhaps because the price calculations for each v get larger and larger.

Question 3)

**Code:**

1. # importing the necessary packages
2. **import** numpy as np
3. **from** numpy **import** sqrt, exp, log
4. **from** scipy.stats **import** norm
5. **import** triSolver
7. ##############################################################################
9. # initial vars
10. S0 = 100.
11. sigma = 0.20
12. r = 0.03
13. T = 1.
14. K = 115.
16. # on interval [0, 200]
17. # assume Smax = 2\*S0
18. Smax = 200.
19. ds = Smax/1000.      # space step
20. M = 1000
21. timestep = 20.  # change this according to question part
22. dt = T/timestep  # time step
23. N = int(T/dt)
25. # stock steps
26. s = np.array([ds\*j **for** j **in** range(M)])
28. # initial v setup
29. v = np.maximum(s - K, 0)
31. # basic alpha, beta, gamma matrix A version
32. alpha = 0.5  \* (sigma\*\*2\*s\*\*2/ds\*\*2 - r\*s/ds)
33. beta = -sigma\*\*2\*s\*\*2/ds\*\*2 - r
34. gamma = 0.5  \* (sigma\*\*2\*s\*\*2/ds\*\*2 + r\*s/ds)
36. # setting up matrix A
37. A = np.diag(beta) + np.diag(alpha[1:], -1) + np.diag(gamma[0:M-1], 1)
39. # boundary conditions for A
40. A[0, :] = 0
41. A[M-1, :] = 0
42. A[M-1, -3:] = np.array([1,-2,1])
44. # identity matrix
45. I = np.identity(1000)
47. # setting up LHS and RHS matrices for price equation
48. leftB = I - 0.5\*dt\*A
49. rightB = I + 0.5\*dt\*A
51. # Crank-Nicolson scheme
52. **for** k **in** range(N):
53. b\_tilda = np.dot(rightB, v)
54. v = triSolver.betterSolver(leftB, b\_tilda)
56. # closed form BS formula price
57. d1 = ( log(S0/K) + (r + (sigma\*\*2)/2)\*T ) / ( sigma\*sqrt(T) )
58. d2 = d1 - sigma\*sqrt(T)
59. BSCallPrice = S0\*norm.cdf(d1) - K\*exp(-r\*T)\*norm.cdf(d2)
60. **print**("Call Price explicitly via BS model formula = " + str(BSCallPrice))
61. **print**("-------------------------------------------------")
63. **print**("Crank-Nicolson finite difference method estimated price (for timestep = " + str(timestep) + ") is:")
64. **print**(v[(M+1)/2])

**Output:**

Call Price explicitly via BS model formula = 3.85958238114

-------------------------------------------------

Crank-Nicolson finite difference method estimated price (for timestep = 20.0) is:

3.85920868022

Crank-Nicolson finite difference method estimated price (for timestep = 50.0) is:

3.85925024133

Crank-Nicolson finite difference method estimated price (for timestep = 100.0) is:

3.85925591002

Crank-Nicolson finite difference method estimated price (for timestep = 500.0) is:

3.85925772885

As we can clearly tell, the Crank-Nicolson finite difference scheme is quite accurate and very close to the closed-form BS formula price.