Question 1)
Code:

```
1.  # importing the necessary packages
2.  import numpy as np
3.  import scipy.linalg as linalg
4.
5.  ###############################################################################
6.
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
14.
15.     Arows = len(A)
16.     A = A.astype(float)
17.     b = b.astype(float)
18.
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]
23.
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
33.
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
41.
42.     A = A.astype(float)
43.     b = b.astype(float)
44.
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]
48.
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]
51.
52.     # then run same tridiagonal solver on new matrix
53.     x = tridiagonalSolver(A, b)
54.     return x
55.
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
63.
64.     # perform L (lower triangular) and U (upper triangular) decomposition
65.     P,L,U = linalg.lu(A)
66.
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:

```
# importing the necessary packages
import numpy as np
from numpy import sqrt, exp, log
from scipy.stats import norm

################################################################################

# initial vars
S0 = 100.
sigma = 0.20
r = 0.03
T = 1
K = 115.

# on interval [0, 200]
# assume Smax = 2*S0
Smax = 200.
ds = Smax/1000.      # space step
M = 1000
timestep = 20.  # change this according to question part
dt = T/timestep  # time step
N = int(T/dt)

# stock steps
s = np.array([ds*j for j in range(M)])

# initial v
v = np.maximum(s - K, 0)

# regular alpha, beta, gamma matrix A version
alpha = 0.5  * (sigma**2*s**2/ds**2 - r*s/ds)
beta = -sigma**2*s**2/ds**2 - r
gamma = 0.5  * (sigma**2*s**2/ds**2 + r*s/ds)

# setting up matrix A
A = np.diag(beta) + np.diag(alpha[1:], -1) + np.diag(gamma[0:M-1], 1)

# boundary conditions for matrix A
A[0, :] = 0     # initial is 0
A[M-1, :] = 0   # linearity boundary
A[M-1, -3:] = np.array([1,-2,1])

# identity matrix
I = np.identity(1000)

# setting up main big matrix A_tilda for price equation
A_tilda = I + dt*A

# EXPLICIT SCHEME
for i in range(N):
    v = np.dot(A, v)

# closed form BS formula price
d1 = ( log(S0/K) + (r + (sigma**2)/2)*T ) / ( sigma*sqrt(T) )
d2 = d1 - sigma*sqrt(T)
BSCallPrice = S0*norm.cdf(d1) - K*exp(-r*T)*norm.cdf(d2)
print("Call Price explicitly via BS model formula = " + str(round(BSCallPrice,2)))
print("-------------------------------------------------")
```

```
59.
60. print("Explicit finite difference method estimated price (for timestep = " + str(timest
    ep) + ") 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:

```
# importing the necessary packages
import numpy as np
from numpy import sqrt, exp, log
from scipy.stats import norm
import triSolver

################################################################################

# initial vars
S0 = 100.
sigma = 0.20
r = 0.03
T = 1.
K = 115.

# on interval [0, 200]
# assume Smax = 2*S0
Smax = 200.
ds = Smax/1000.      # space step
M = 1000
timestep = 20.  # change this according to question part
dt = T/timestep  # time step
N = int(T/dt)

# stock steps
s = np.array([ds*j for j in range(M)])

# initial v setup
v = np.maximum(s - K, 0)

# basic alpha, beta, gamma matrix A version
alpha = 0.5  * (sigma**2*s**2/ds**2 - r*s/ds)
beta = -sigma**2*s**2/ds**2 - r
gamma = 0.5  * (sigma**2*s**2/ds**2 + r*s/ds)

# setting up matrix A
A = np.diag(beta) + np.diag(alpha[1:], -1) + np.diag(gamma[0:M-1], 1)

# boundary conditions for A
A[0, :] = 0
A[M-1, :] = 0
A[M-1, -3:] = np.array([1,-2,1])

# identity matrix
I = np.identity(1000)

# setting up LHS and RHS matrices for price equation
leftB = I - 0.5*dt*A
rightB = I + 0.5*dt*A

# Crank-Nicolson scheme
for k in range(N):
    b_tilda = np.dot(rightB, v)
    v = triSolver.betterSolver(leftB, b_tilda)

# closed form BS formula price
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("-------------------------------------------------")
62.
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.