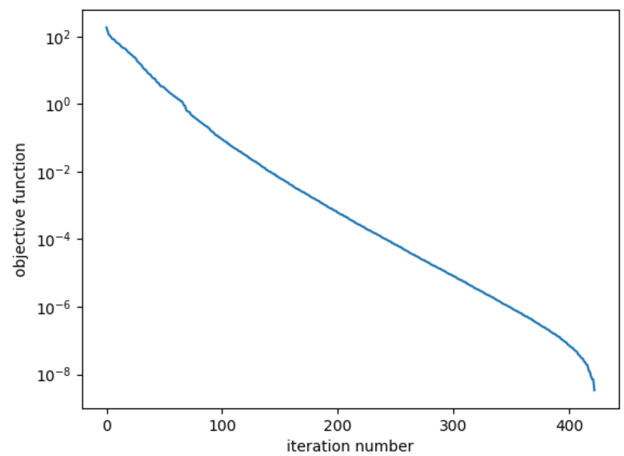
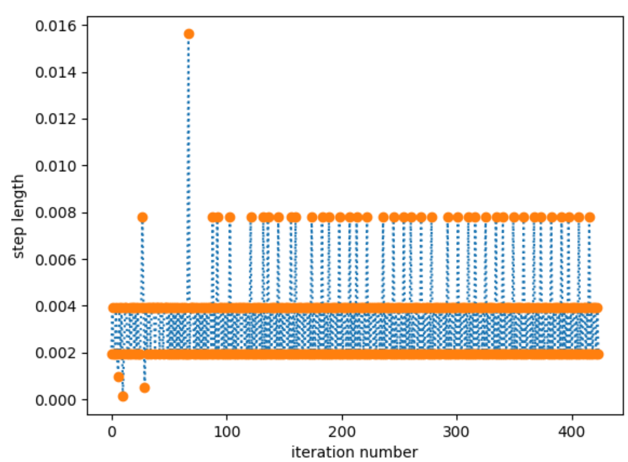
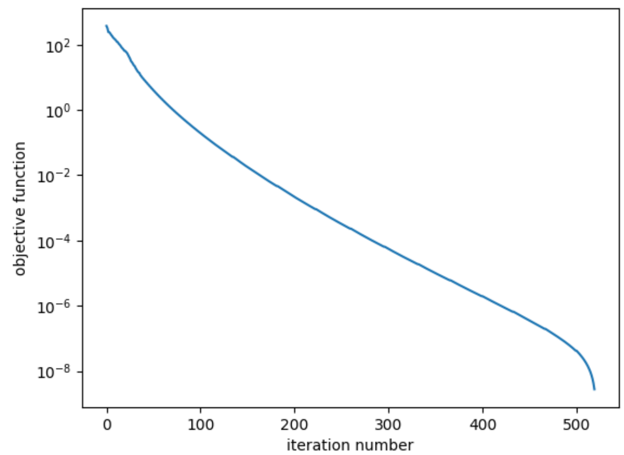
CompSci 206 PS5
Xiaoying Li
12/05/2021

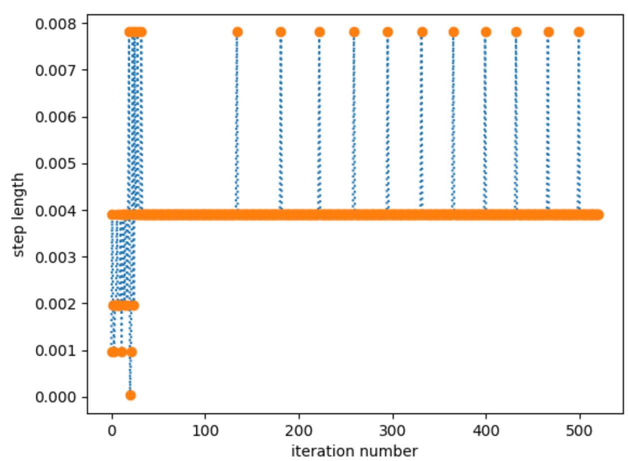
CO 9.30
(a)
Below are some instances of different backtracking parameters α and β, and different sizes.
Instance1: $m = 200, n = 100, \alpha = 0.01, \beta = 0.5$



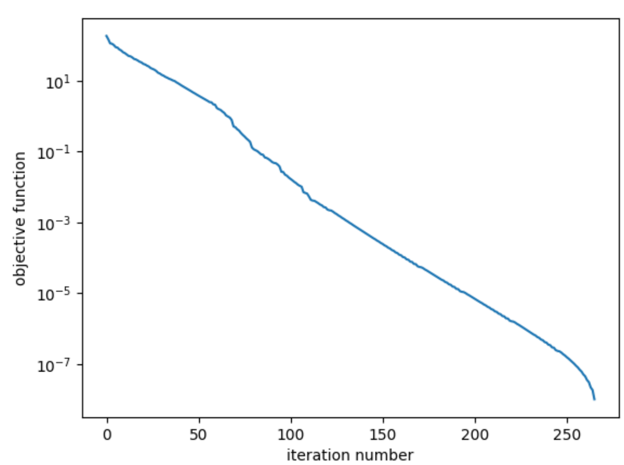


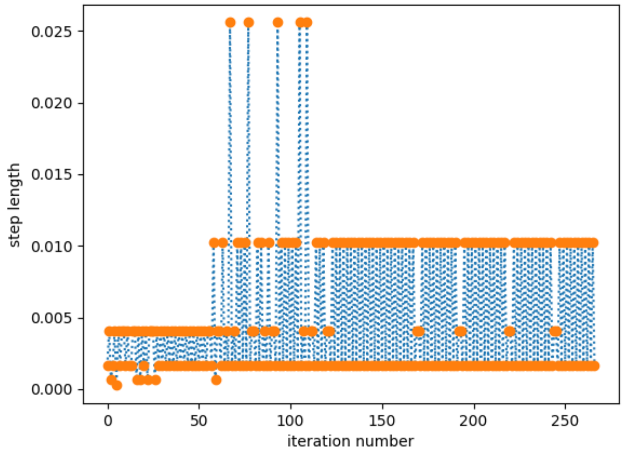
Instance 2: $m = 300, n = 200, \alpha = 0.01, \beta = 0.5$



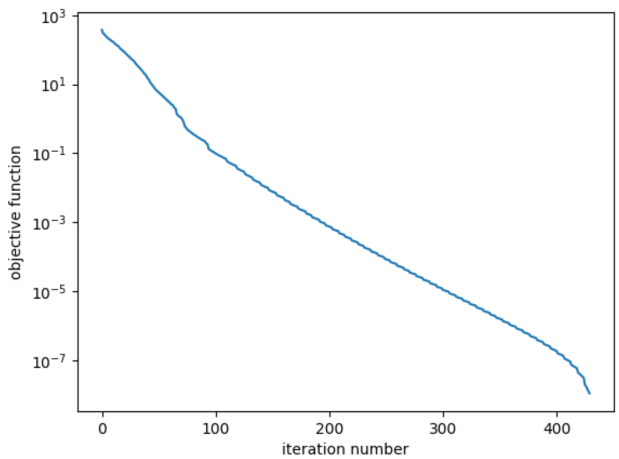


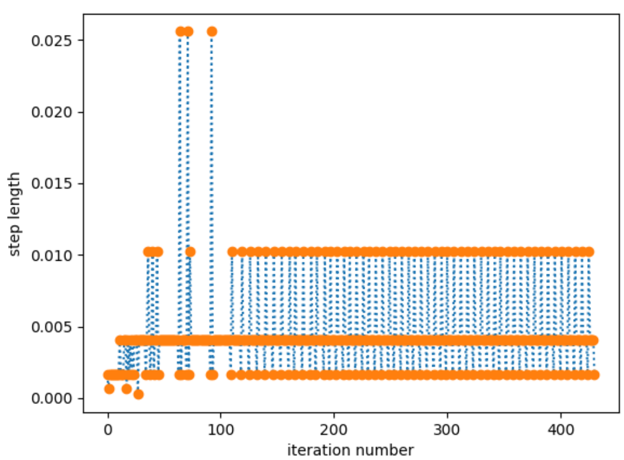
Instance 3: $m = 200, n = 100, \alpha = 0.05, \beta = 0.3$





Instance 4: $m = 300, n = 200, \alpha = 0.05, \beta = 0.3$





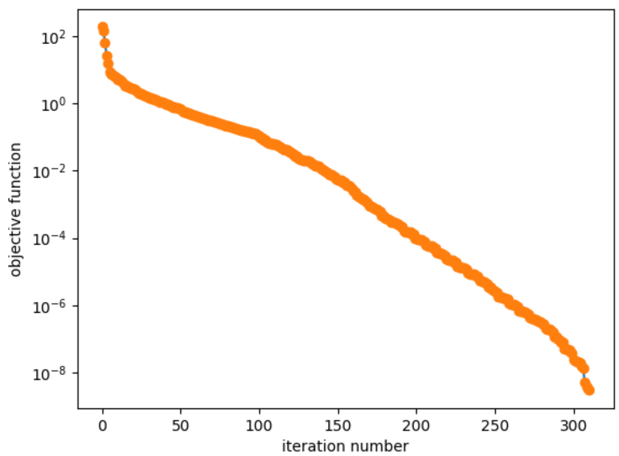
Python implementation:

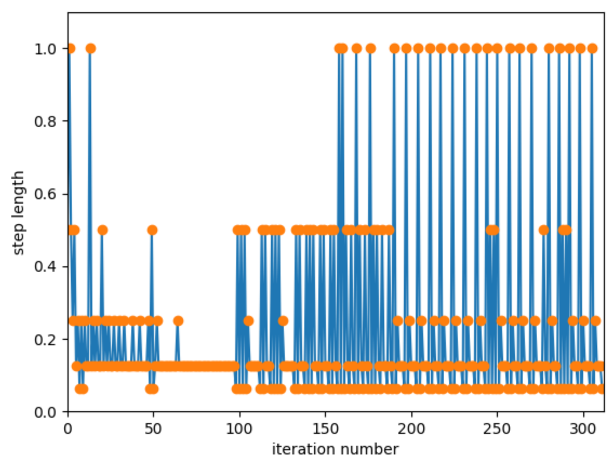
```
import numpy as np
from matplotlib import pyplot as plt

np.random.seed(1)
m = 200
n = 100
alpha = 0.01
beta = 0.5
max_iteration = 1000
newton_tol = 1e-8
gradient_tol = 1e-3
A = np.random.randn(m, n)

x = np.zeros((n, 1))
value = 0
values = []
d = 0
gradient = 0
hessian = 0
steps = []

for i in range(max_iteration):
    value = -np.sum(np.log(1 - A.dot(x))) - np.sum(np.log(1 + x)) -
np.sum(np.log(1 - x))
    values.append(value)
    d = 1 / (1 - A.dot(x))
    d_list = d.T.tolist()[0]
    gradient = A.T.dot(d) - 1 / (1 + x) + 1 / (1 - x)
    v = -gradient
    fprime = gradient.T.dot(v)
    if np.linalg.norm(gradient) < gradient_tol:
        break
    t = 1
    while (np.max(A.dot(x + t * v)) >= 1) or (np.max(np.abs(x + t * v)) >=
1):
        t = beta * t
    while (-sum(np.log(1 - A.dot(x + t * v))) - sum(np.log(1 - (x + t * v) **
2))) > value + alpha * t * fprime:
        t = beta * t
    x = x + t * v
    steps.append(t)

optimal_value = values[-1]
plt.figure(1)
plt.semilogy(range(len(values) - 2), values[0:-2] - optimal_value, '-')
plt.xlabel('iteration number')
plt.ylabel('objective function')
plt.show()
plt.figure(2)
plt.plot(range(len(steps)), steps, ':', range(len(steps)), steps, 'o')
plt.xlabel('iteration number')
plt.ylabel('step length')
plt.show()
```

(b)

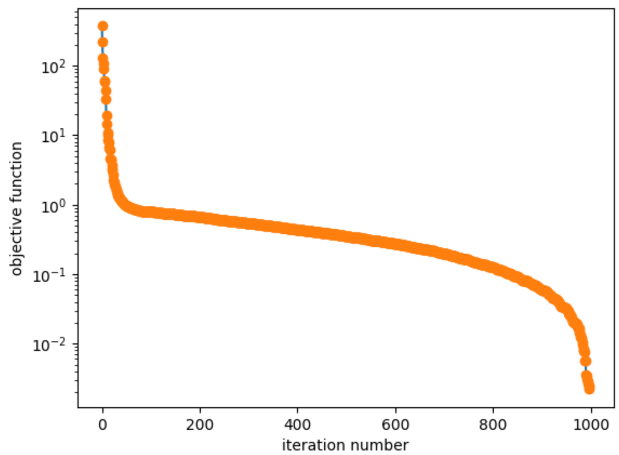
Below are some instances of different backtracking parameters α and β, and different sizes.

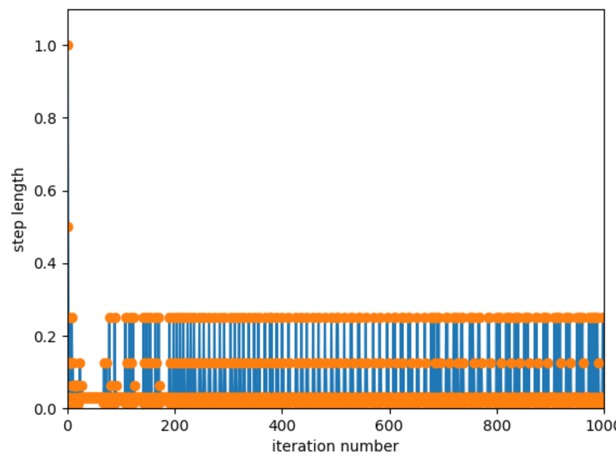
Instance1: $m = 200, n = 100, \alpha = 0.01, \beta = 0.5$



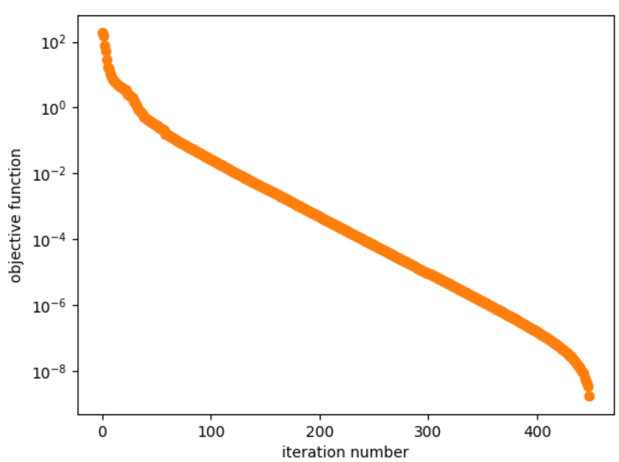


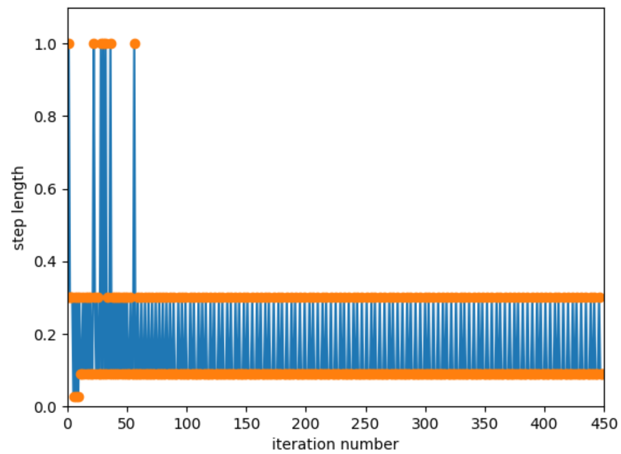
Instance 2: $m = 300, n = 200, \alpha = 0.01, \beta = 0.5$



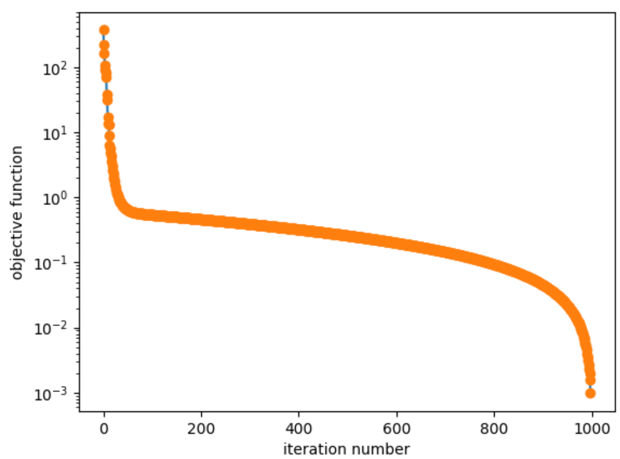


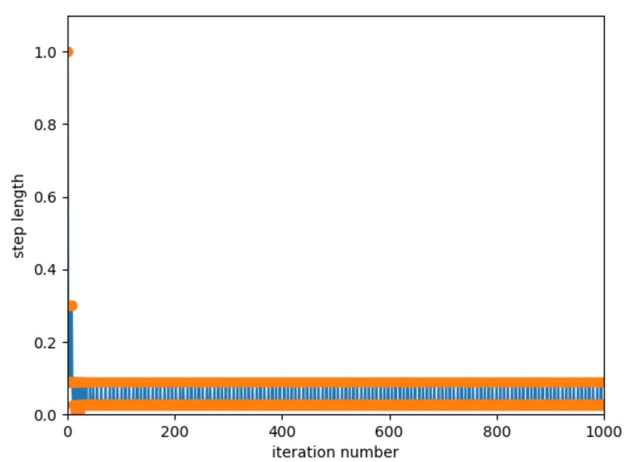
Instance 3: $m = 200, n = 100, \alpha = 0.05, \beta = 0.3$





Instance 4: $m = 300, n = 200, \alpha = 0.05, \beta = 0.3$





Python implementation:

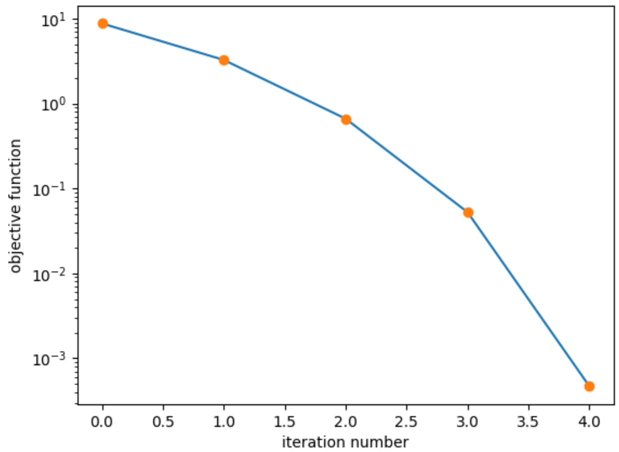
```
import numpy as np
from matplotlib import pyplot as plt

np.random.seed(1)
m = 200
n = 100
alpha = 0.01
beta = 0.5
max_iteration = 1000
newton_tol = 1e-8
gradient_tol = 1e-3
A = np.random.randn(m, n)

x = np.zeros((n, 1))
value = 0
values = []
d = 0
gradient = 0
hessian = 0
steps = []

for i in range(max_iteration):
    value = -np.sum(np.log(1 - A.dot(x))) - np.sum(np.log(1 + x)) -
np.sum(np.log(1 - x))
    values.append(value)
    d = 1 / (1 - A.dot(x))
    d_list = d.T.tolist()[0]
    gradient = A.T.dot(d) - 1 / (1 + x) + 1 / (1 - x)
    hessian = np.matmul(A.T, np.diag(d_list) ** 2).dot(A) + np.diag(1 / (1 +
x) ** 2 + 1 / (1 - x) ** 2)
    v = -np.linalg.inv(hessian).dot(gradient)
    fprime = gradient.T.dot(v)
    if np.abs(fprime) < newton_tol:
        break
    t = 1
    while (np.max(A.dot(x + t * v)) >= 1) or (np.max(np.abs(x + t * v)) >=
1):
        t = beta * t
    while (-sum(np.log(1 - A.dot(x + t * v))) - sum(np.log(1 - (x + t * v) **
2))) > value + alpha * t * fprime:
        t = beta * t
    x = x + t * v
    steps.append(t)

optimal_value = values[-1]
plt.figure(3)
plt.semilogy(range(len(values) - 2), values[0:-2] - optimal_value, '-',
range(len(values) - 2), values[0:-2] - optimal_value, 'o')
plt.xlabel('iteration number')
plt.ylabel('objective function')
plt.show()
plt.figure(4)
plt.plot(range(len(steps)), steps, '-', range(len(steps)), steps, 'o')
plt.axis([0, len(steps), 0, 1.1])
plt.xlabel('iteration number')
```

```
plt.ylabel('step length')
plt.show()
```

CO 10.15

(a)

The bellowing plot shows the objective function value versus iteration number for the given example.



Python implementation:

```
import numpy as np
from matplotlib import pyplot as plt

max_iteration = 100
alpha = 0.01
beta = 0.5
newton_tol = 1e-7
p = 30
n = 100
A = np.random.randn(p, n)
x = np.random.rand(n, 1)
values = []

for i in range(max_iteration):
    val = x.T.dot(np.log(x))
    values.append(val[0][0])
    grad = 1 + np.log(x)
    x_ = 1 / x
    x_list = x_.T.tolist()[0]
    hess = np.diag(x_list)
    mat1 = np.hstack((hess, A.T))
    mat2 = np.hstack((A, np.zeros((p, p))))
    mat3 = np.vstack((mat1, mat2))
    mat4 = np.vstack((grad, np.zeros((p, 1))))
    sol = -np.linalg.inv(mat3).dot(mat4)
    v = sol[0:n]
    fprime = grad.T.dot(v)
    if np.abs(fprime) < newton_tol:
```

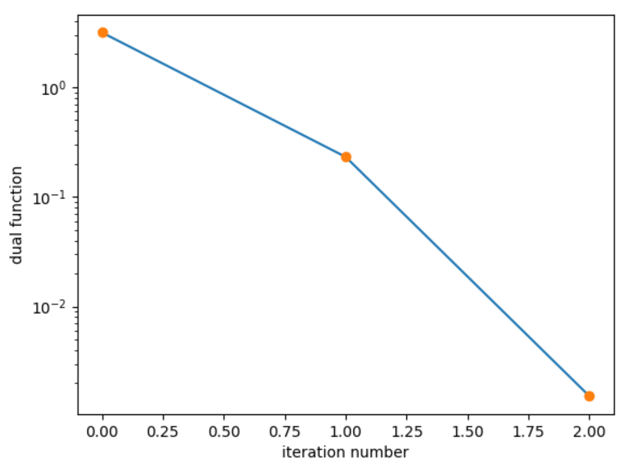
```
        break
    t = 1
    while np.min(x + t * v) <= 0:
        t = beta * t
    while (x + t * v).T.dot(np.log(x + t * v)) >= val + t * alpha * fprime:
        t = beta * t
    x = x + t * v

optimal_value = values[-1]
print(values)
plt.figure(1)
plt.semilogy(range(len(values) - 2), values[0:-2] - optimal_value, '-',
range(len(values) - 2), values[0:-2] - optimal_value, 'o')
plt.xlabel('iteration number')
plt.ylabel('objective function')
plt.show()
```

(c)

In dual Newton method, the dual problem is

$$maximize - b^T v - \sum_{i=1}^{n} e^{-a_i^T v - 1}, a_i = ith\ column\ of\ A$$

The bellowing plot shows the dual function value versus iteration number for the given example.



Python implementation:

```
import numpy as np
from matplotlib import pyplot as plt

np.random.seed(0)
max_iteration = 100
alpha = 0.01
beta = 0.5
newton_tol = 1e-8
p = 30
n = 100
A = np.random.randn(p, n)
x = np.random.rand(n, 1)
```

```
nu = np.zeros((p, 1))
values = []
b = A.dot(x)

for i in range(max_iteration):
    exp_vec = np.exp(-A.T.dot(nu)-1)
    val = b.T.dot(nu) + np.sum(exp_vec)
    values.append(val[0][0])
    grad = b - A.dot(exp_vec)
    exp_vec_list = exp_vec.T.tolist()[0]
    hess = A.dot(np.diag(exp_vec_list)).dot(A.T)
    v = -np.linalg.inv(hess).dot(grad)
    fprime = grad.T.dot(v)
    if np.abs(fprime) < newton_tol:
        break
    t = 1
    while b.T.dot(nu + t * v) + np.sum(np.exp(-A.T.dot(nu + t * v)-1)) > val
+ t * alpha * fprime:
        t = beta * t
    nu = nu + t * v

optimal_value = values[-1]
plt.figure(2)
plt.semilogy(range(len(values) - 2), values[0:-2] - optimal_value, '-',
range(len(values) - 2), values[0:-2] - optimal_value, 'o')
plt.xlabel('iteration number')
plt.ylabel('objective function')
plt.show()
```

The computational efforts of standard Newton method and dual Newton method are the same.
In the standard Newton method, the problem is solved by coefficient matrix

$$\begin{bmatrix} \nabla^2 f(x) & A^T \\ A & 0 \end{bmatrix}, with\ \nabla^2 f(x) = diag(x)^{-1}$$

Block elimination reduces the equation to one with coefficient matrix

$$A\ diag(x)\ A^T$$

In the dual Newton method, the problem is solved with coefficient matrix

$$-\nabla^2 g(v) = ADA^T, where\ D\ diagonal\ with\ D_{ii} = e^{-a_i^T v - 1}$$

Therefore, in both methods, the main computation in each iteration is a linear system of the form

$$A^T D A_v = -g$$

And D is diagonal with positive diagonal elements.

CO 11.22

From 8.16, we obtain,

$$minimize - \sum_{i=1}^{n} \log(u_i - l_i)$$

$$subjuct\ to\ A^+ u - A^- l \leq b, with\ implicit\ contraint\ u > l$$

$$a_{ij}^{+} = \max\{a_{ij}, 0\}, a_{ij}^{-} = \max\{-a_{ij}, 0\}$$

For the function

$$\psi(l, u) = -t\sum_{i=1}^{n} \log(u_i - l_i) - \sum_{i=1}^{n} \log\left((b - A^{+}u + A^{-}l)_i\right)$$

the gradient and Hessian are

$$\nabla\psi(l, u) = t\begin{bmatrix} I \\ -I \end{bmatrix} diag(u - l)^{-1}1 + \begin{bmatrix} -A^{-T} \\ A^{+T} \end{bmatrix} diag(b - A^{+}u + A^{-}l)^{-1}1$$

$$\nabla^2\psi(l, u) = t\begin{bmatrix} I \\ -I \end{bmatrix} diag(u - l)^{-1}[I \quad -I] + \begin{bmatrix} -A^{-T} \\ A^{+T} \end{bmatrix} diag(b - A^{+}u + A^{-}l)^{-1}[-A^{-} \quad A^{+}]$$

Python implementation:

```
import numpy as np

max_iteration = 200
alpha = 0.01
beta = 0.5
newton_tol = 1e-8
MU = 20
TOL = 1e-4
A = [[0, -1],
    [2, -4],
    [2, 1],
    [-4, 4],
    [-4, 0]]
Am = [[0, 1],
      [0, 4],
      [0, 0],
      [4, 0],
      [4, 0]]
Ap = [[0, 0],
      [2, 0],
      [2, 1],
      [0, 4],
      [0, 0]]
b = 1
n = 2
Am = np.array(Am)
Ap = np.array(Ap)
r = np.max(Ap.dot(np.ones((n, 1))) + Am.dot(np.ones((n, 1))))
u = (0.5 / r) * np.ones((n, 1))
l = -(0.5 / r) * np.ones((n, 1))
t = 1
for i in range(max_iteration):
    y = b + Am.dot(l) - Ap.dot(u)
    val = -t * np.sum(np.log(u-1)) - np.sum(np.log(y))
    grad = t * np.vstack((1/(u-1), -1/(u-1))) + np.vstack((-Am.T, -
Ap.T)).dot(1/y)
    diag_a = 1 / ((u-1) ** 2)
    diag_l = diag_a.T.tolist()[0]
    diag_mat1 = np.diag(diag_l)
    diag_y = 1 / (y ** 2)
    diag_yl = diag_y.T.tolist()[0]
```

```
    diag_mat2 = np.diag(diag_yl)
    mat1 = np.hstack((diag_mat1, -diag_mat1))
    mat2 = np.hstack((-diag_mat1, diag_mat1))
    mat3 = np.vstack((mat1, mat2))
    mat4 = np.vstack((-Am.T, Ap.T))
    mat5 = np.hstack((-Am, Ap))
    hess = t * mat3 + mat4.dot(diag_mat2).dot(mat5)
    step = np.linalg.inv(hess).dot(grad)
    fprime = grad.T.dot(step)
    if np.abs(fprime) < newton_tol:
        gap = 2 * m / t
        if gap < TOP:
            break
        t = MU * t
    else:
        dl = step[0:n]
        du = step[n:2 * n]
        dy = Am.dot(dl) - Ap.dot(du)
        tls = 1
        mat_m = np.vstack((u - 1 + tls*(du - dl), y + tls*dy))
        while np.min(mat_m) <= 0:
            tls = beta * tls
        while -t * np.sum(np.log(u - 1 + tls * (du - dl))) - np.sum(np.log(y
+ tls * dy)) >= val + tls * alpha * fprime:
            tls = beta * tls
        l = l + tls * dl
        u = u + tls * du
```