1 Problem 2 Take Home Final Exam

```python
# Importing the necessary libraries
import matplotlib.pyplot as plt
import numpy as np
from numpy.linalg import norm

np.random.seed(999) # Seed random number generator for consistency

#Initialize the matrix dimensions
n = 100
m = 200

# Create a random problem
A = np.random.randn(m,n)

# Pick the stopping condition
eta = 1e-4
eps = 1e-8
i_max = 1000
```

1.1 Part a - Gradient Descent

```python
# Function goes here
def f(x):
        return -sum(np.log(1-np.dot(A,x)))-sum(np.log(1-x))-sum(np.log(1+x))

# Have to differentiate and hard code derivitive
def grad_f(x):
        s1 = 1/(1-np.dot(A,x))
        s2 = np.dot(np.transpose(A),s1)
        return s2+1/(1-x)-1/(1+x)
```

1.1.1 Algorithm for Backtracking Line Search

while f(x + t delta x) > f(x) + alpha t gradient(f(x)) delta x: t = beta t

```python
# Backtracking method for updating t
def backtrack(x,dx,alpha,beta):
        t=1.0
        # Make sure updating to feasible x, use approximation to ensure in feasible doma
        s = x+np.dot(t,dx)
        while (np.amax(np.dot(A,s))>1) or (np.amax(abs(s))>1):
                t=beta*t
                s = x+t*dx
        # Backtrack in feasible domain
        s2 = np.dot(np.transpose(grad_f(x)),dx)
        while f(s) > f(x)+alpha*t*s2:
                t=beta*t
                s = x+t*dx
                s2 = np.dot(np.transpose(grad_f(x)),dx)
        return t
```

1.1.2 Algorithm for gradient descent

1. Dx = -1 * gradient f(x)
2. Line search - choose step size t via backtracking method
3. Update x = x + t Dx

```
def gradient_descent(iterations,alpha,beta):
        y = np.array([])
        s = np.array([])
        # Use zero as initial guess
        x = np.zeros(n)
        x = x.reshape(n,1) # Make sure x is a vector
        # Repeat
        for i in range(iterations):
                # Step 1
                y = np.append(y,f(x))
                dx = -1*grad_f(x)

                # Step 2
                t = backtrack(x,-grad_f(x),alpha,beta)
                s = np.append(s,t)

                # Step 3
                x = x + t*dx

                p = f(x)

                if norm(grad_f(x)) < eta:
                        break

        return y,s,p
```

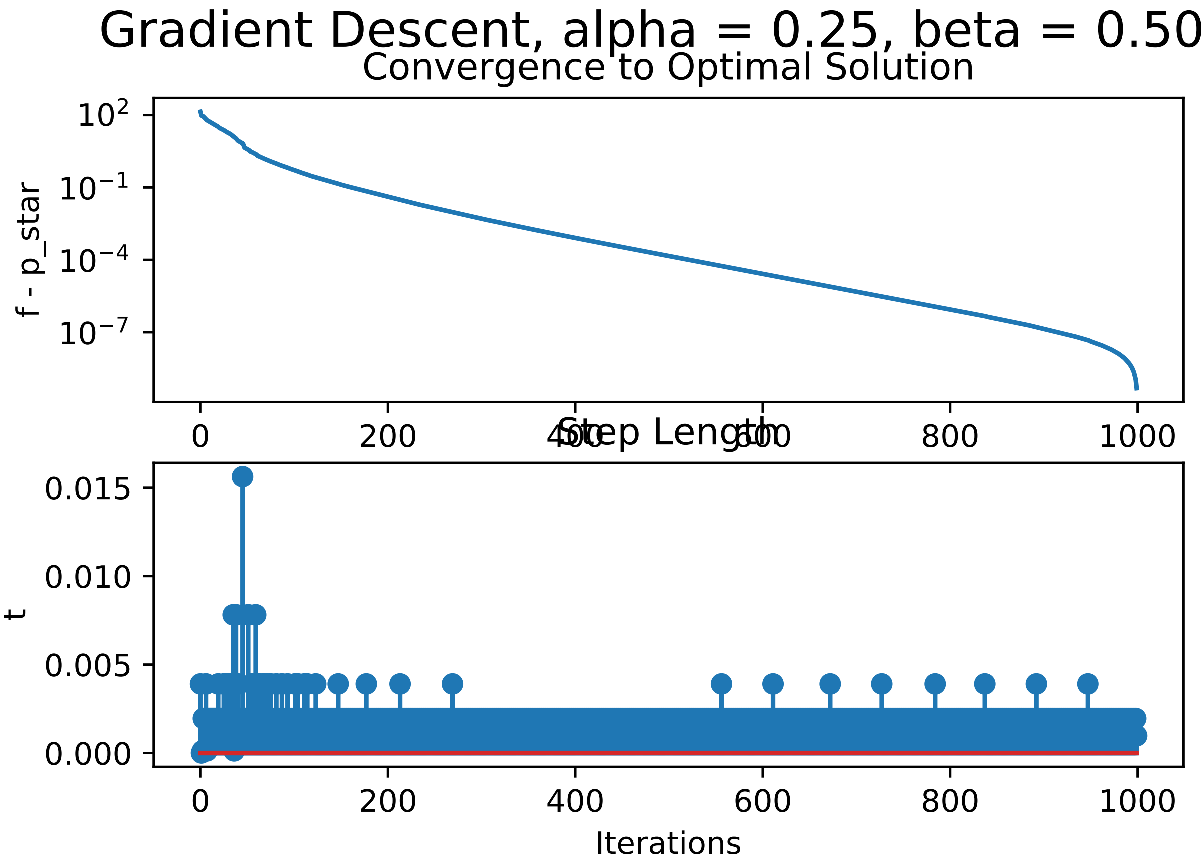
Making a plot function to play with alpha and beta more easily

```
def gradient_descent_plot(alpha,beta):
        # Run gradient descent algorithm
        y,s,p = gradient_descent(i_max,alpha,beta)

        fig,ax = plt.subplots(2,1)

        ax[0].semilogy(y-p)
        ax[0].set_title('Convergence to Optimal Solution')
        ax[0].set_xlabel('Iterations')
        ax[0].set_ylabel('f - p_star')

        ax[1].stem(s)
        ax[1].set_title('Step Length')
        ax[1].set_xlabel('Iterations')
        ax[1].set_ylabel('t')

        fig.suptitle('Gradient Descent, alpha = %1.2f, beta = %1.2f'%(alpha,beta),fonts

        return fig,ax
```

1.1.3 Figures and analysis

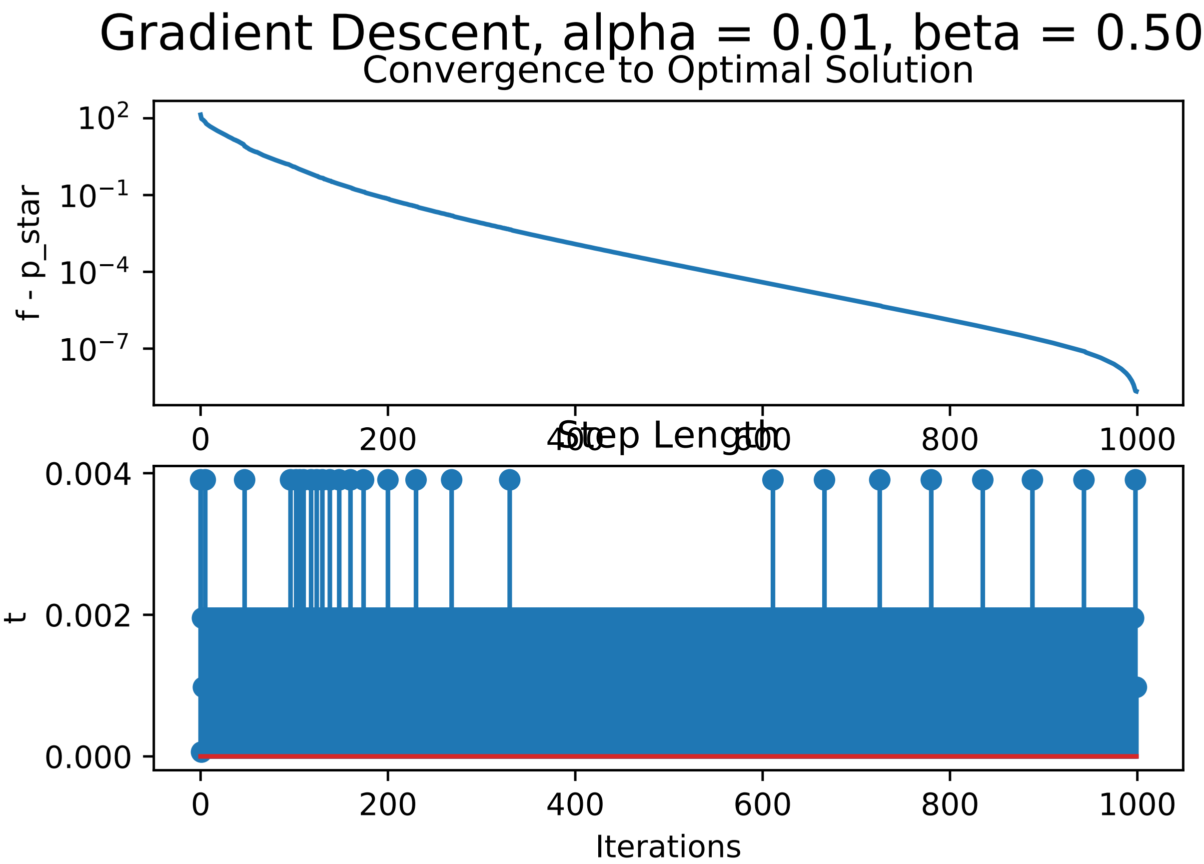
Below is the code to perform gradient descent with alpha=0.25, beta=0.5

```
#fig,ax = gradient_descent_plot(alpha,beta)
fig0,ax0 = gradient_descent_plot(0.25,0.5)
plt.savefig('fig1.png')
```



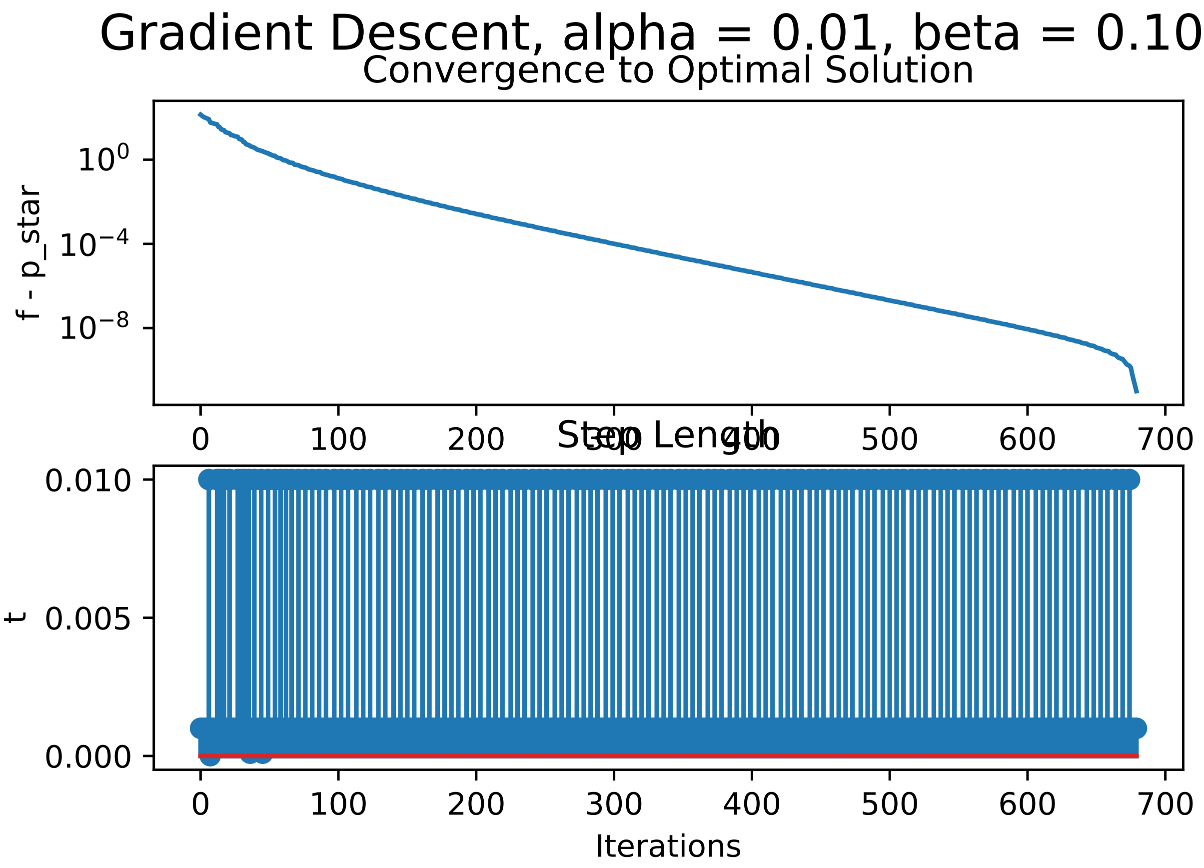
Below is the code to perform gradient descent with alpha=0.1, beta=0.5

```
fig1,ax1 = gradient_descent_plot(0.01,0.5)
plt.savefig('fig2.png')
```



The decrease in alpha eliminates some of the larger step size changes, and yields a smoother but slightly less accurate convergence in the bounds of the iterations Let's try playing with beta a little bit now

```
fig2,ax2 = gradient_descent_plot(0.01,0.1)
plt.savefig('fig3.png')
```



Great! Our convergence has gotten even more accurate. Notice also the step size increase.

1.2 Part b - repeat using netwon's method

```
def hess_f(x):
        s1 = 1/(1-np.dot(A,x))
        s2 = np.diag(np.power(s1,2)[:,0])
        s3 = np.dot(s2,A)
        s4 = np.dot(np.transpose(A),s3)
        s5 = np.power(1+x,2)
        s6 = np.diag(1/s5)
        s7 = np.power(1-x,2)
        s8 = np.diag(1/s7)
        return s4+s6+s8

def newton_method(iterations,alpha,beta,eps):
        y = np.array([])
        s = np.array([])
        x = np.zeros(n)
        x = x.reshape(n,1) # Make sure x is a vector
        # Repeat
        # Step 1
        hf = hess_f(x)
        gf = grad_f(x)
        xnt = -np.linalg.solve(hf,gf)
        dec = -np.dot(np.transpose(gf),xnt)
        p = f(x)
        print(dec)
        for i in range(iterations):
```

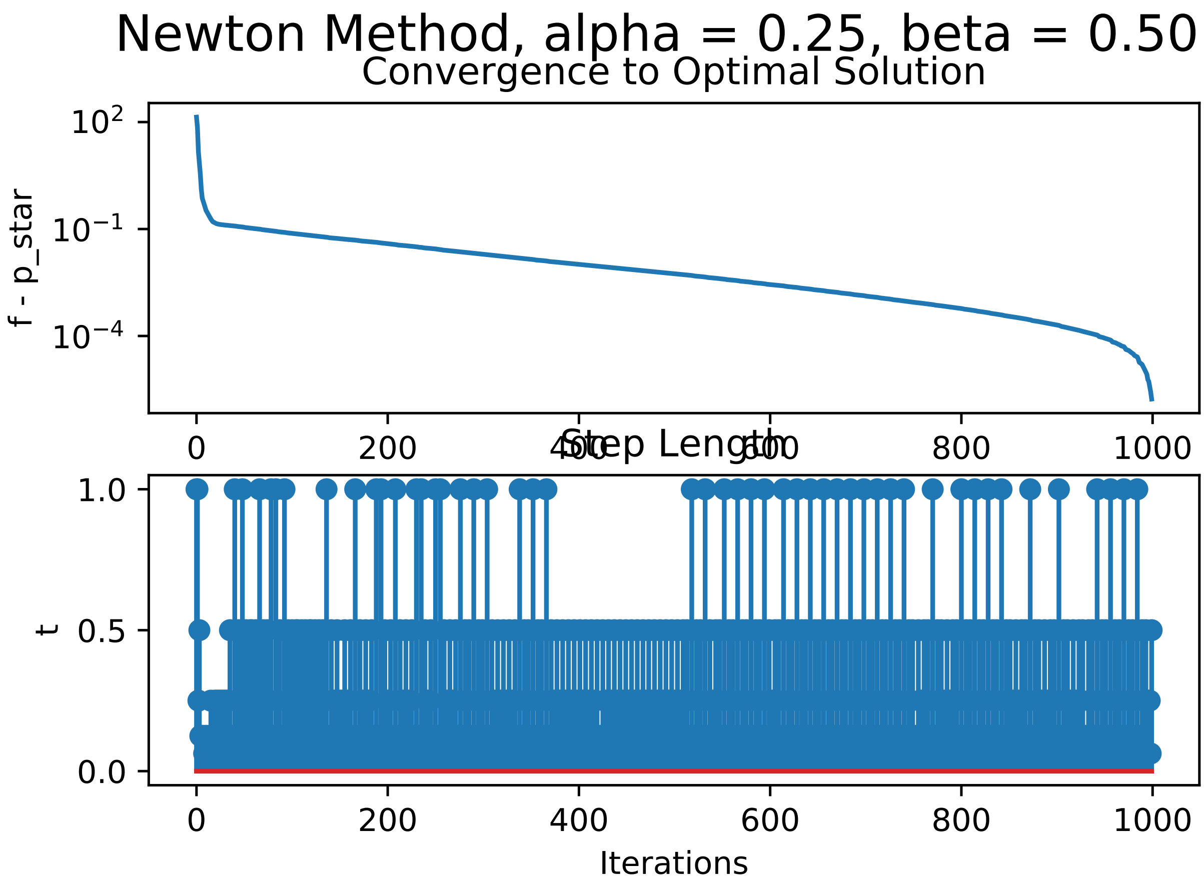
```
        # Step 1
        y = np.append(y,f(x))

        # Step 2
        if dec/2 <= eps:
            break


        t = backtrack(x,xnt,alpha,beta)
        s = np.append(s,t)

        # Step 3
        x = x + t*xnt

        hf = hess_f(x)
        gf = grad_f(x)
        xnt = -np.linalg.solve(hf,gf)
        dec = -np.dot(np.transpose(gf),xnt)

        p = f(x)

    return y,s,p

def newton_method_plot(alpha,beta,eps):
    # Run gradient descent algorithm
    y,s,p = newton_method(i_max,alpha,beta,eps)

    fig,ax = plt.subplots(2,1)

    ax[0].semilogy(y-p)
    ax[0].set_title('Convergence to Optimal Solution')
    ax[0].set_xlabel('Iterations')
    ax[0].set_ylabel('f - p_star')

    ax[1].stem(s)
    ax[1].set_title('Step Length')
    ax[1].set_xlabel('Iterations')
    ax[1].set_ylabel('t')

    fig.suptitle('Newton Method, alpha = %1.2f, beta = %1.2f'%(alpha,beta),fontsize=

    return fig,ax
```

1.2.1 Figures and analysis

Below is the code to perform newton method with alpha=0.25, beta=0.5

```
#fig,ax = newton_method_plot(alpha,beta)
fig3,ax3 = newton_method_plot(0.25,0.5,eps)
plt.savefig('fig4.png')
```

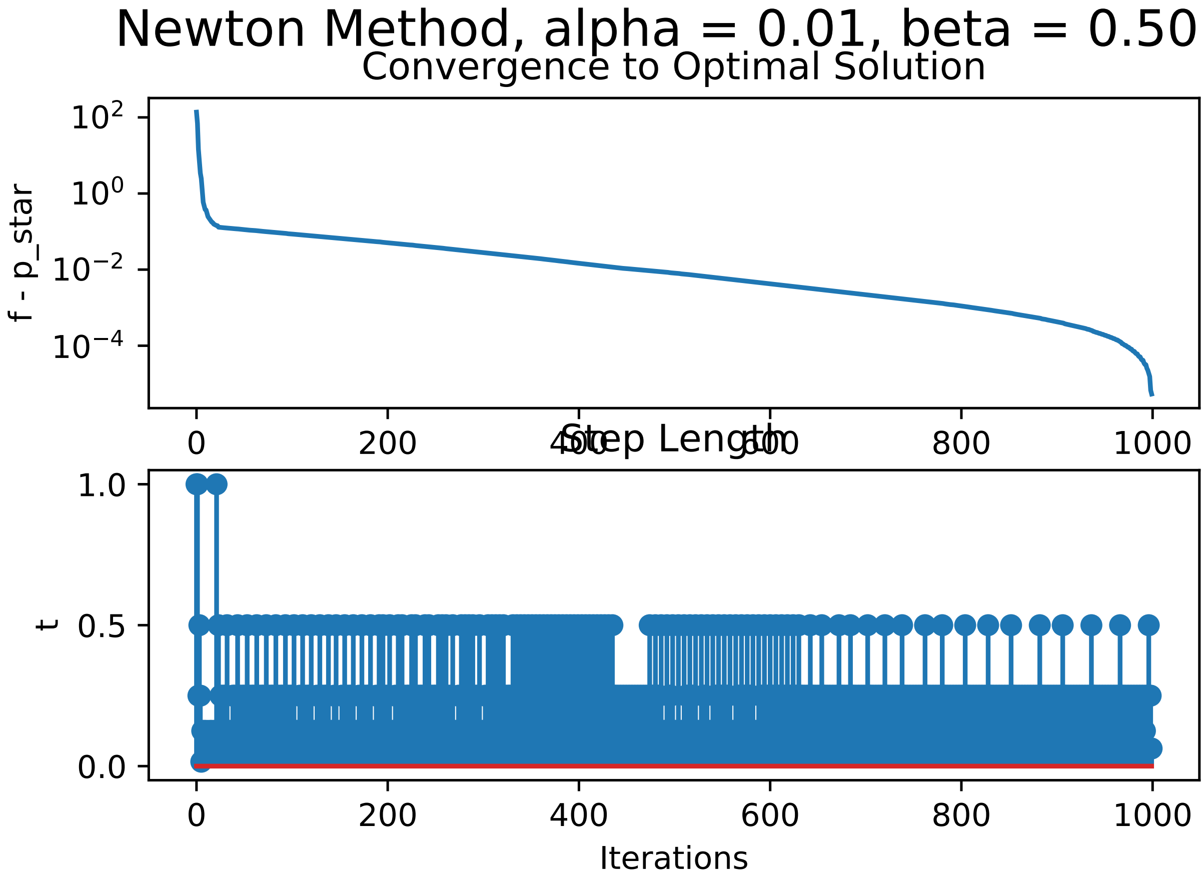
```
[[101.82381699]]
```



Below is the code to perform newton with alpha=0.1, beta=0.5

```
fig4,ax4 = newton_method_plot(0.01,0.5,eps)
plt.savefig('fig5.png')
```

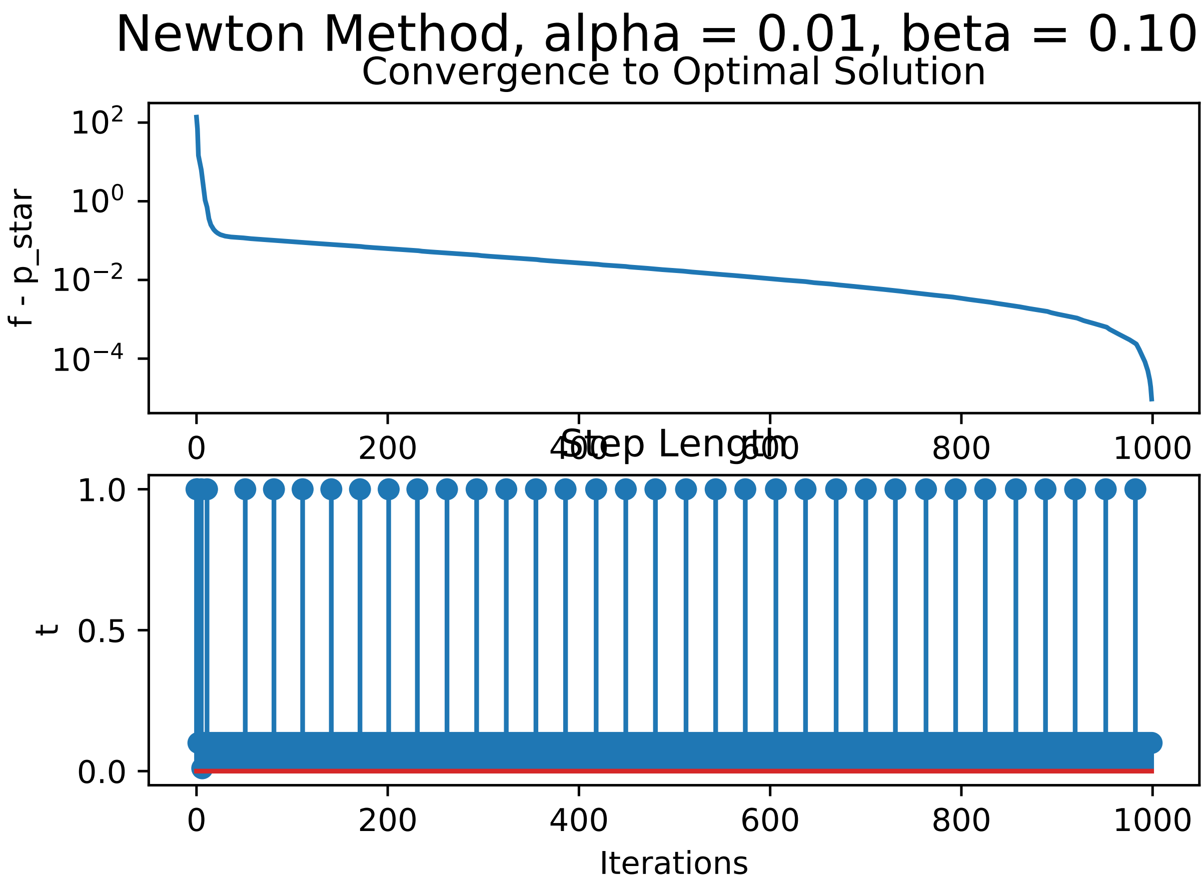
```
[[101.82381699]]
```



Let's try playing with beta a little bit now

```
fig5,ax5 = newton_method_plot(0.01,0.1,eps)
plt.savefig('fig6.png')
```

```
[[101.82381699]]
```



Notice how in comparison to the gradient descent method, the newton method converges much more quickly than the gradient descent