optimization

November 29, 2022

1 Write a script that implement the GD algorithm, with the following structure:

Input: f: the function f(x) we want to optimize. It is supposed to be a Python function, not an array. grad_f: the gradient of f(x). It is supposed to be a Python function, not an array. x0: an n-dimensional array which represents the initial iterate. kmax: an integer. The maximum possible number of iterations (to avoid infinite loops) tolf: small float. The relative tollerance of the algorithm. Convergence happens if ||grad_f(x_k)||*2 < tolf ||grad_f(x_0)||*2 tolx: small float. The tollerance in the input domain. Convergence happens if ||x*{k}* - *x*{k-1}||_2 < tolx. Pay attention to the first iterate. Output: x: an array that contains the value of x_k FOR EACH iterate x_k (not only the latter). k: an integer. The number of iteration needed to converge. k < kmax. f_val: an array that contains the value of f(x_k) FOR EACH iterate x_k. grads: an array that contains the value of grad_f(x_k) FOR EACH iterate x_k. err: an array the contains the value of ||grad_f(x_k)||_2 FOR EACH iterate x_k.

[71]:
```python
import matplotlib.pyplot as plt
import numpy
import numpy as np
```

[72]:
```python
def plot(x, errf, x_true=False, back=False, alpha=0.2):
    k = len(x)
    title = f"{'x*= ' + str(x_true)  if x_true else ''} x_c={np.round(x[-1], 2)} N. of iteration: {k}, backtracking: {'yes' if back else 'no'} {'alpha: ' + str(alpha) if not back else ''}"
    plt.title(title)
    plt.plot(errf)
    legend = ["error"]
    if x_true:
        x_errors = np.zeros(x.shape)
        for i, x_k in enumerate(x):
            x_errors[i] = np.linalg.norm(x_k - x_true)
        plt.plot(x_errors)
        legend.append("X error")
    #plt.subplot(2, 2, 2)
    #plt.plot(points, grads)
    #plt.subplot(2, 2, 3)
    #plt.plot(points, err)
```

```python
    plt.legend(legend)
    plt.show()


def backtracking(f, grad_f, x):
    """
    This function is a simple implementation of the backtracking algorithm for
    the GD (Gradient Descent) method.

    f: function. The function that we want to optimize.
    grad_f: function. The gradient of f(x).
    x: ndarray. The actual iterate x_k.
    """
    alpha = 1
    c = 0.8
    tau = 0.25

    while f(x - alpha * grad_f(x)) > f(x) - c * alpha * np.linalg.
 ↪norm(grad_f(x), 2) ** 2:
        alpha = tau * alpha

        if alpha < 1e-3:
            break
    return alpha


def GD(f, grad_f, x0, tolf, tolx, kmax, alpha=0.2, back=False):

    x0 = np.array(x0)
    shape = (kmax, *x0.shape)
    # output
    x = np.zeros(shape)
    f_val = np.zeros(shape)
    grads = np.zeros(shape)
    err = np.zeros(shape)

    x_tol = tolx
    f_tol = tolf
    x_old = x0
    k = 0

    while k < kmax and x_tol >= tolx and f_tol >= tolf:
        if back:
            alpha = backtracking(f, grad_f, x_old)
        x_k = x_old - alpha * np.array(grad_f(x_old))
        x_tol = np.linalg.norm(x_k-x_old)
```

```
        f_tol = np.linalg.norm(f(x_k))

        # Update arrays
        x[k] = x_k
        f_val[k] = f(x_k)
        grads[k] = grad_f(x_k)
        err[k] = np.linalg.norm(grads[k])
        x_old = x_k
        k = k+1

    return x[:k], f_val[:k], grads[:k], err[:k]
```

1.1 Test the algorithm above on the following functions:

[73]:
```
tolf = 1e-4
tolx = 1e-4
kmax = 100
alphas = [0.1, 0.01]
```

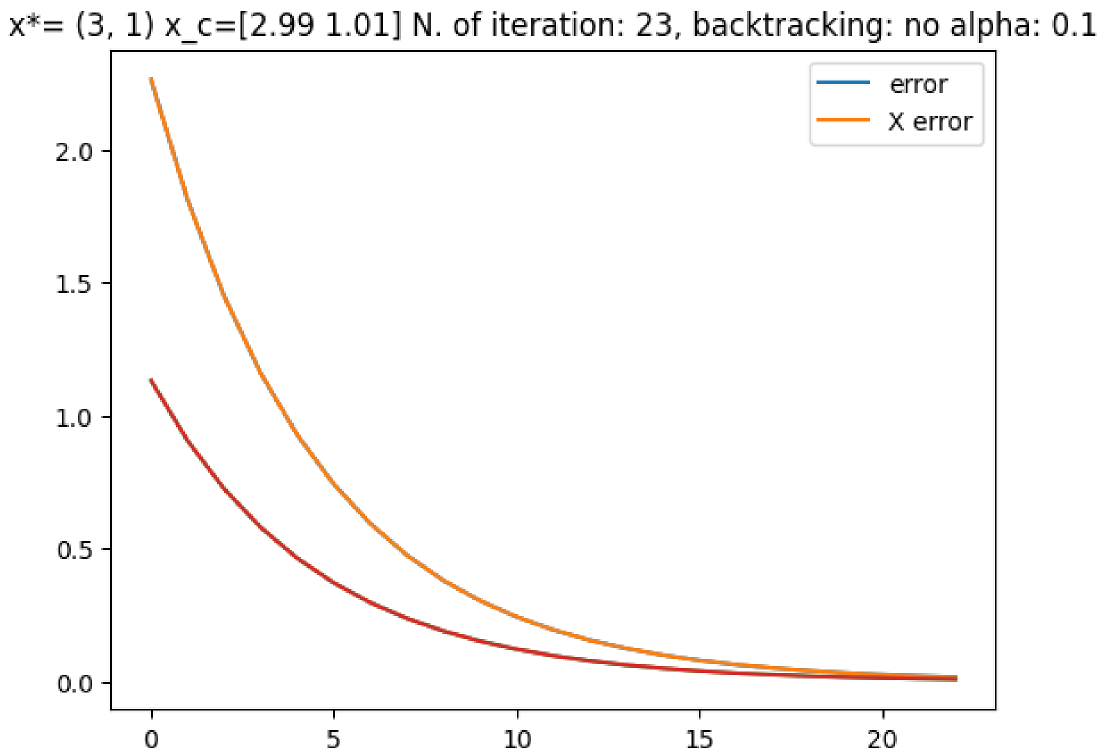
1.1.1 Function 1

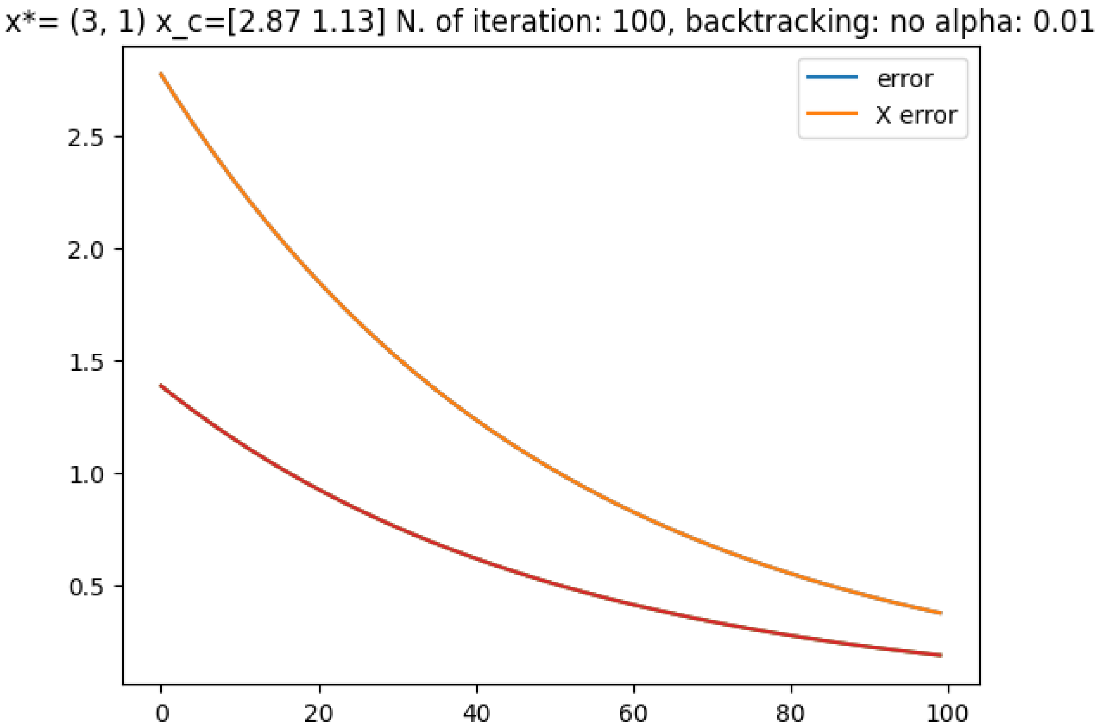
$$f(x_1, x_2) = (x_1 - 3)^2 + (x_2 - 1)^2$$

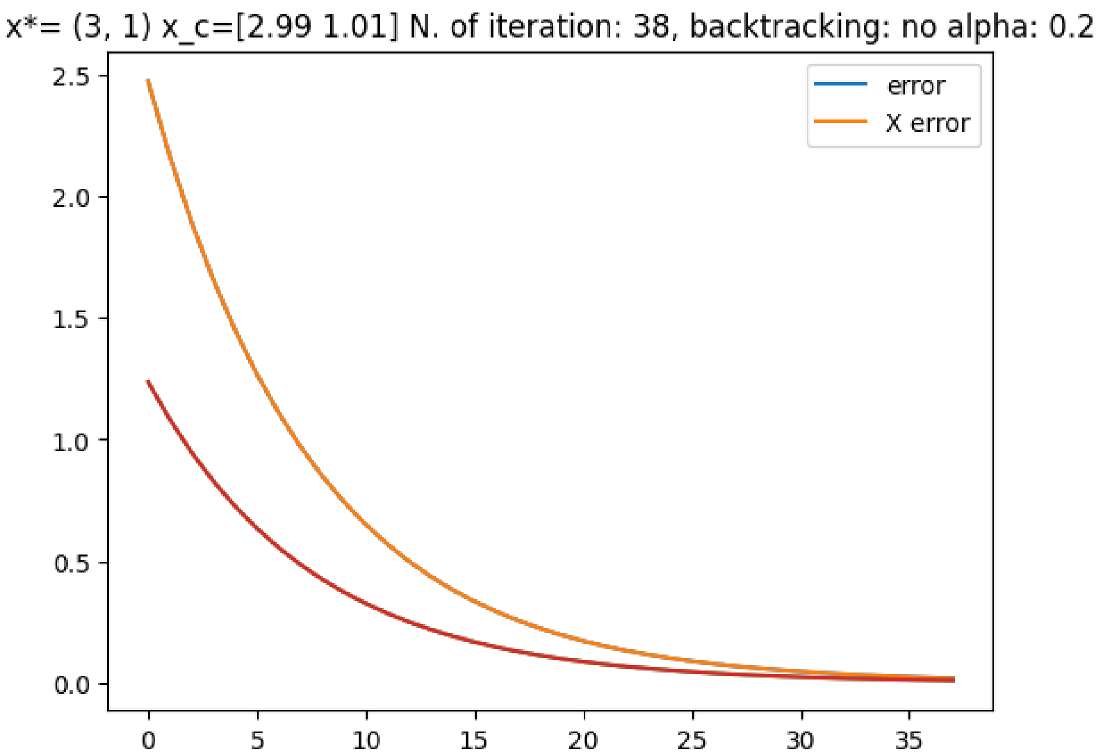
for which the true optimum is $x^* = (3, 1)^T$

[74]:
```
def f1(x):
    x1, x2 = x
    return (x1 - 3)**2 + (x2 - 1)**2



def grad_f1(x):
    x1, x2 = x
    return np.array((2*(x1-3), 2*(x2-1)))



x0 = (2, 2)
x_true1 = (3, 1)



def test_function(f, grad_f, x0, kmax, x_true=False, f5=False):
    for alpha in alphas:
        x, f_val, grads, err = GD(f, grad_f, x0, tolf, tolx, kmax, alpha)
        plot(x, err, x_true, alpha=alpha)
        if f5:
            x_ = np.linspace(-3, 3, 1000)
            plt.plot(x_, f(x_))
            plt.plot(x, f_val, "bo")
```

```
            plt.show()
    x, f_val, grads, err = GD(f, grad_f, x0, tolf, tolx, kmax, back=True)
    plot(x, err, x_true)

    if f5:
        x_ = np.linspace(-3, 3, 1000)
        plt.plot(x_, f(x_))
        plt.plot(x, f_val, "bo")
        plt.show()


test_function(f1, grad_f1, x0, kmax, x_true1)
```



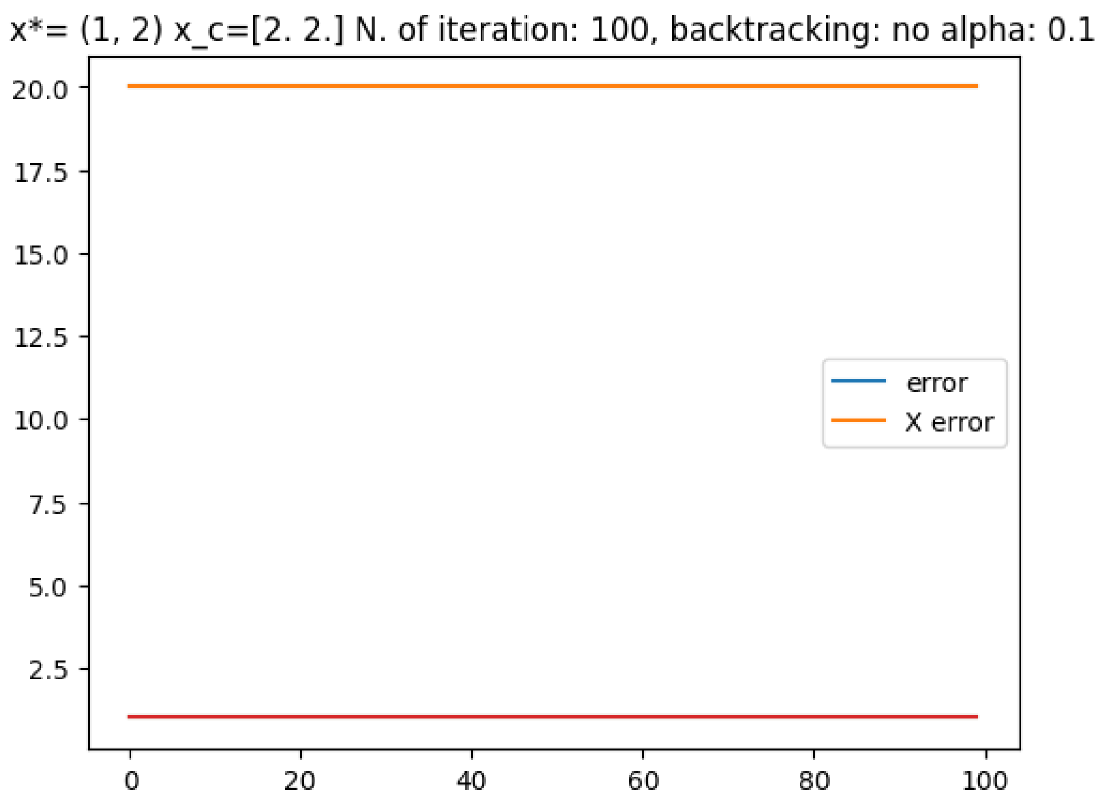

x*= (3, 1) x_c=[2.87 1.13] N. of iteration: 100, backtracking: no alpha: 0.01
error
X error
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100

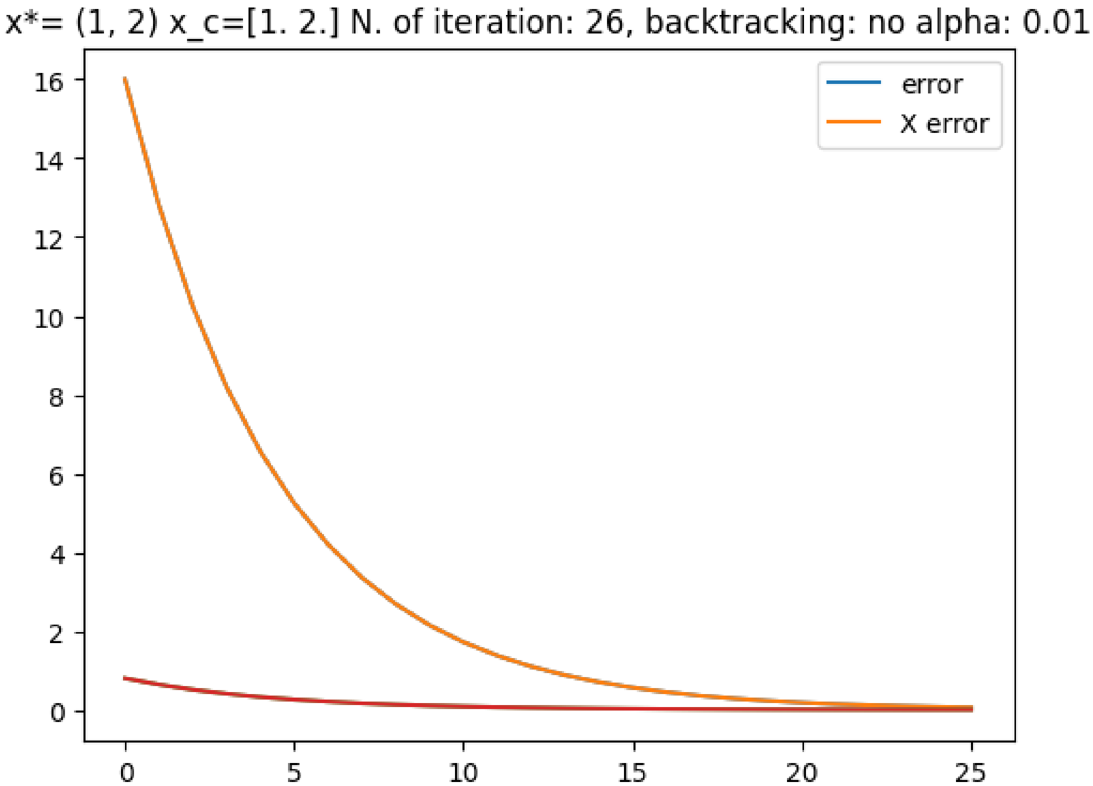

x*= (3, 1) x_c=[2.99 1.01] N. of iteration: 38, backtracking: no alpha: 0.2
error
X error
2.5
2.0
1.5
1.0
0.5
0.0
0
5
10
15
20
25
30
35

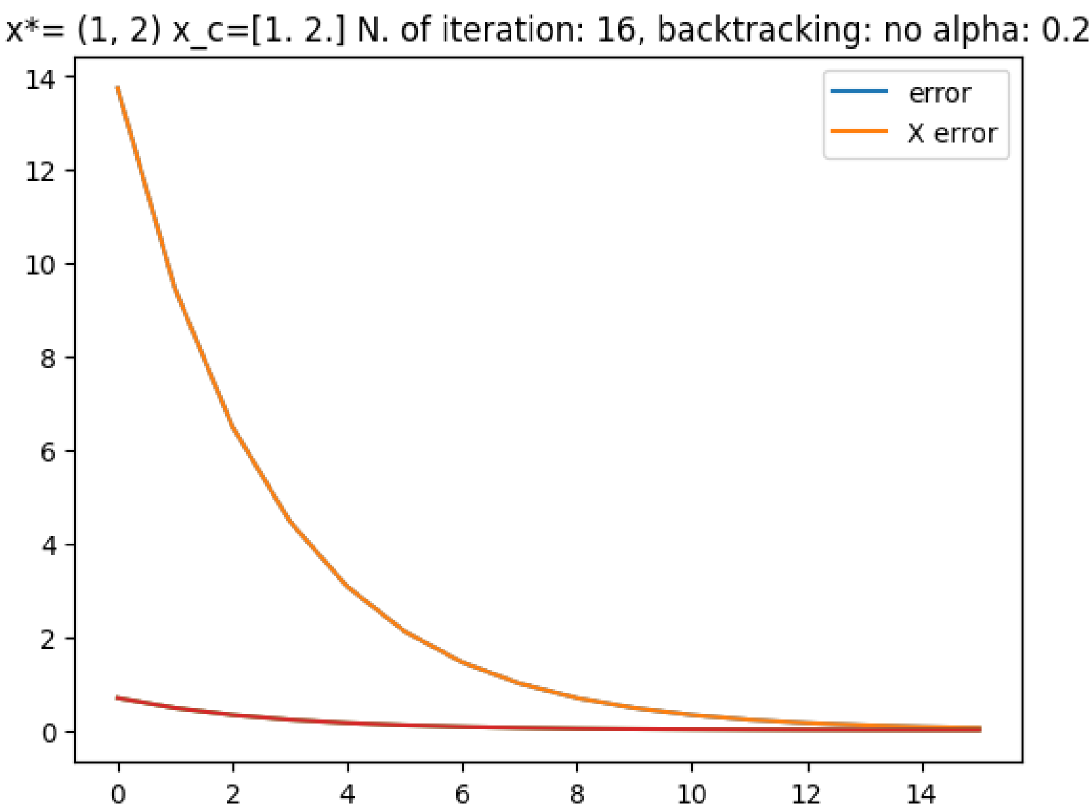
1.2 Function 2

$$f(x_1, x_2) = 10(x_1 - 1)^2 + (x_2 - 2)^2$$

[75]:
```python
def f2(x):
    x1, x2 = x
    return 10*(x1 - 1)**2 + (x2 - 2)**2
def grad_f2(x):
    x1, x2 = x
    return np.array((20*(x1-1), 2*(x2-2)))


x0 = (2, 2)
x_true2 = (1,2)

test_function(f2, grad_f2, x0, kmax, x_true2)
```




x*= (1, 2) x_c=[1. 2.] N. of iteration: 26, backtracking: no alpha: 0.01
error
X error
16
14
12
10
8
6
4
2
0
0
5
10
15
20
25



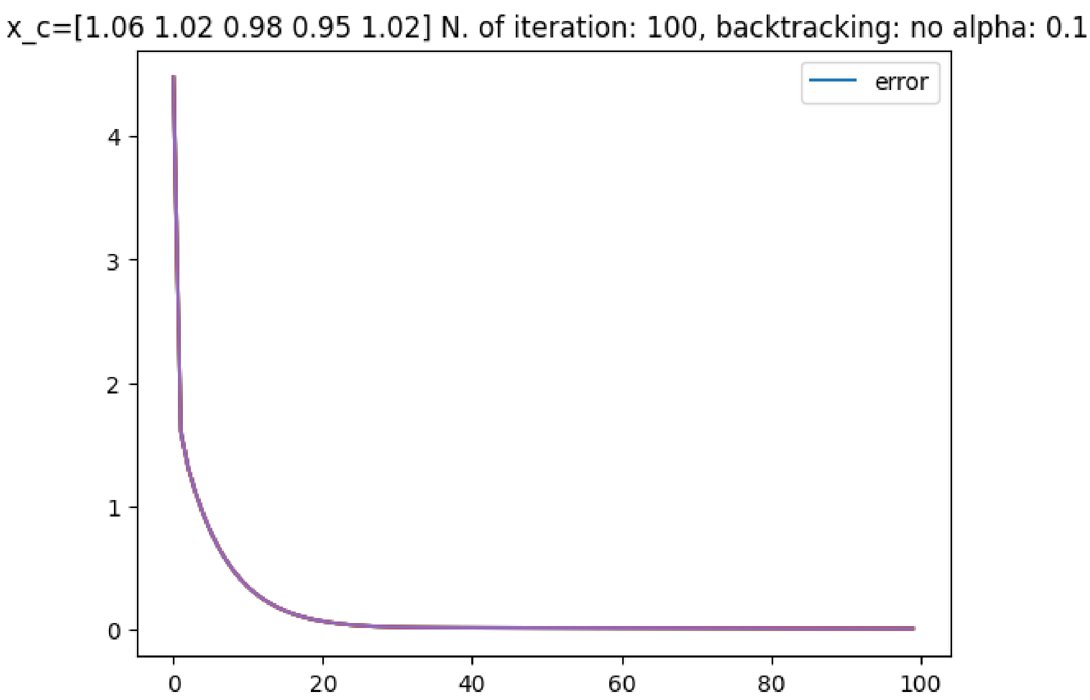
1.3 Function 3

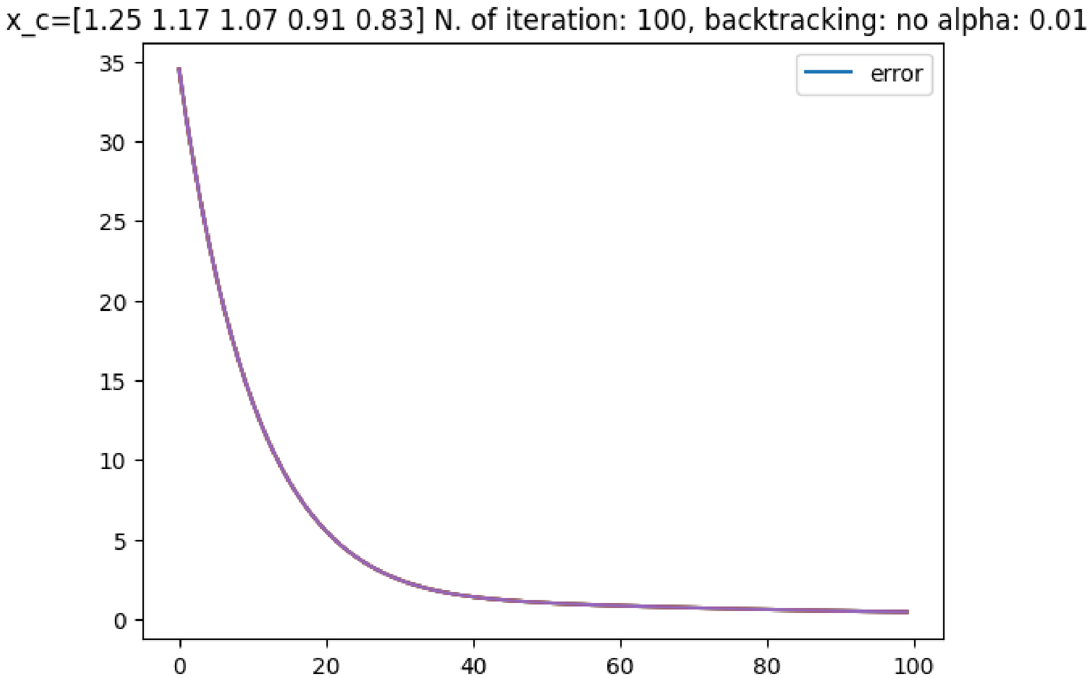
$$f(x) = \frac{1}{2}||Ax - b||_2^2$$

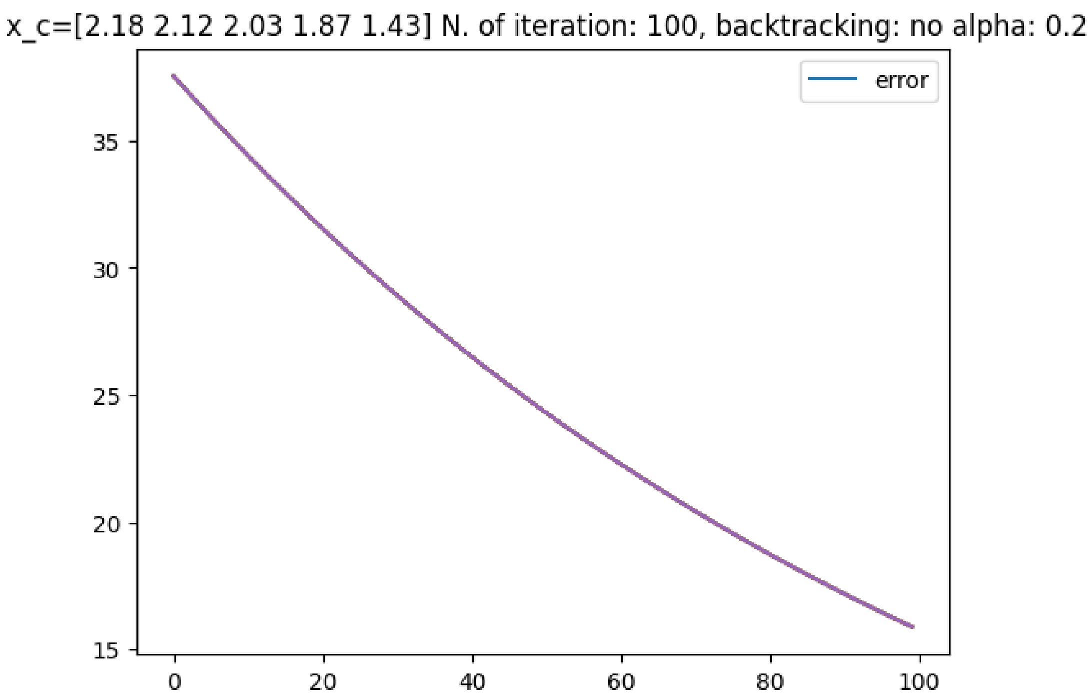
[76]:
```
def f3(x):
    x = np.array(x)
    x = np.reshape(x,(1, len(x)))
    n, m = x.shape
    x_true = np.ones((1, n))
    v = np.linspace(0, 1, n, endpoint=True)
    A = numpy.vander(v)
    b = A @ x_true
    return 1/2 * np.linalg.norm(A @ x - b, 2)**2

def grad_f3(x):
    n = len(x)
    v = np.linspace(0,1,n)
    A = np.vander(v)
    x_true = np.ones(n).T
    b = A @ x_true
```

```
    return np.array(A.T@(A@x-b))


N = [5, 10, 15]

for n in N:
    x0 = [3 for i in range(n)]
    print("N = ", n)
    test_function(f3, grad_f3, x0, kmax)
```

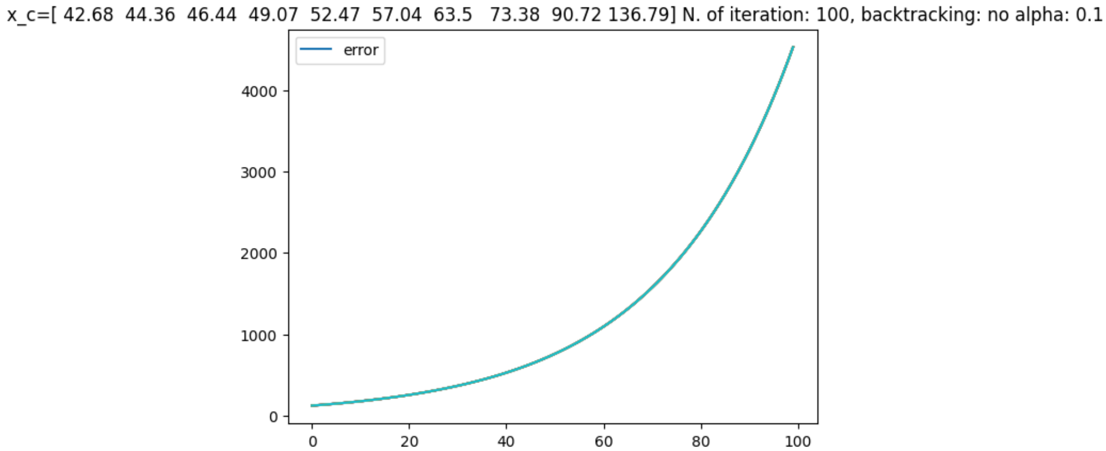
```
N =  5
```

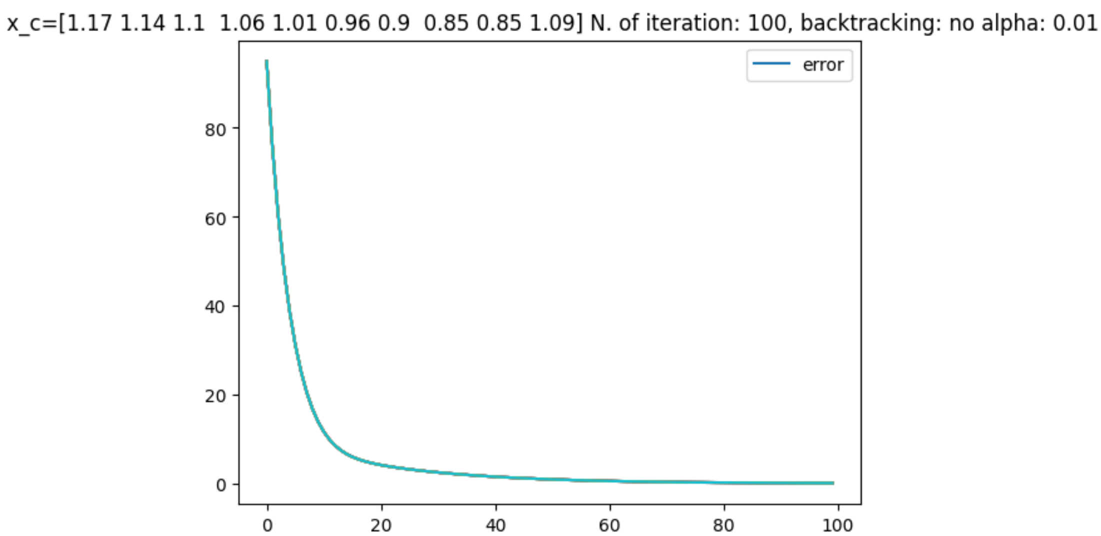


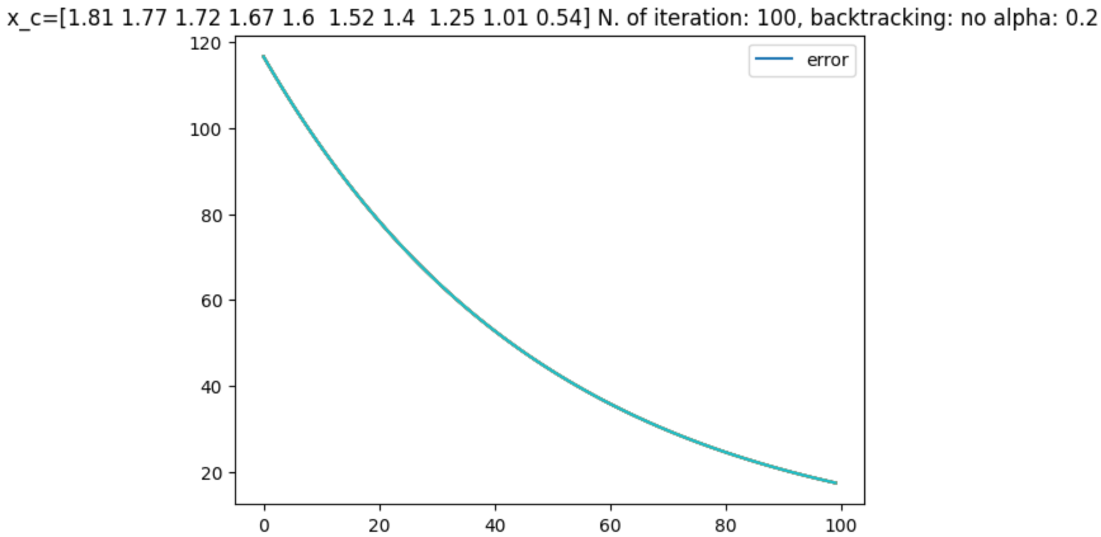

x_c=[1.25 1.17 1.07 0.91 0.83] N. of iteration: 100, backtracking: no alpha: 0.01
error
35
30
25
20
15
10
5
0
0
20
40
60
80
100


x_c=[2.18 2.12 2.03 1.87 1.43] N. of iteration: 100, backtracking: no alpha: 0.2
error
35
30
25
20
15
0
20
40
60
80
100

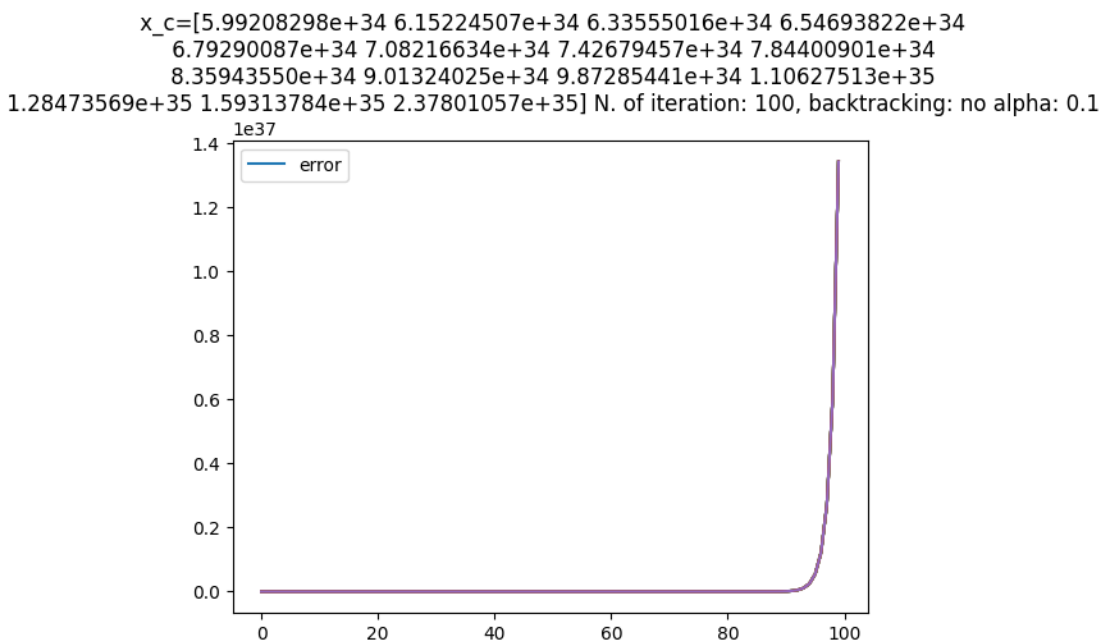
```
N =  10
```

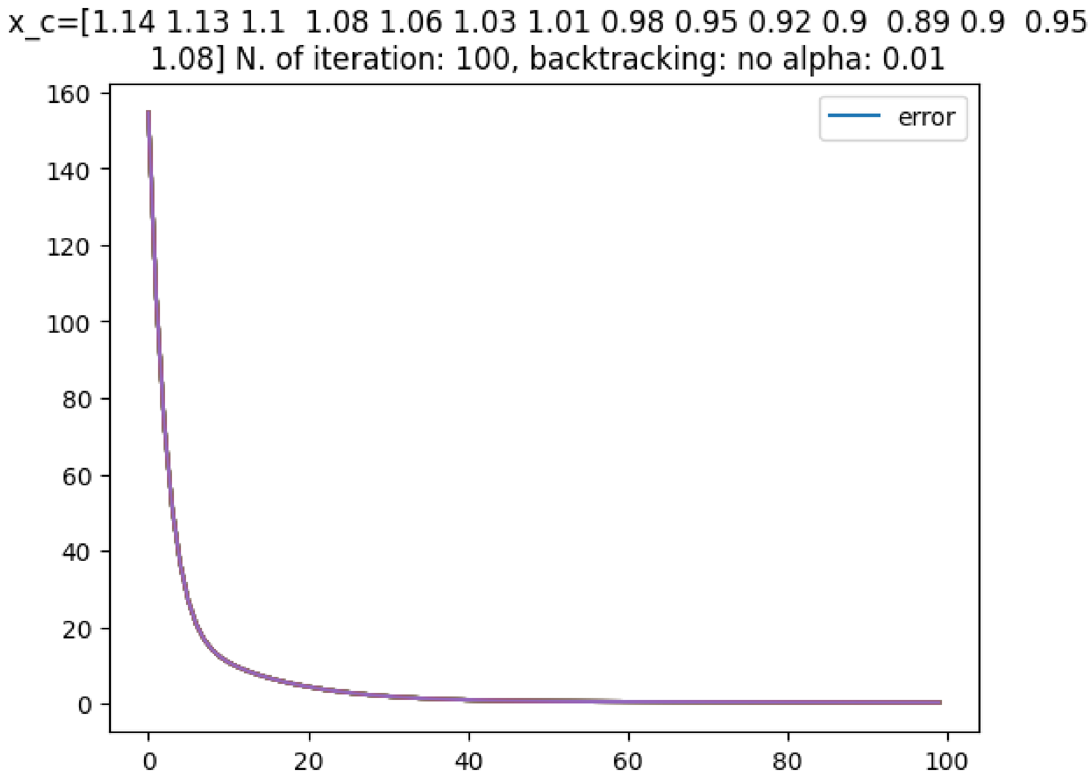

x_c=[42.68 44.36 46.44 49.07 52.47 57.04 63.5   73.38 90.72 136.79] N. of iteration: 100, backtracking: no alpha: 0.1
error

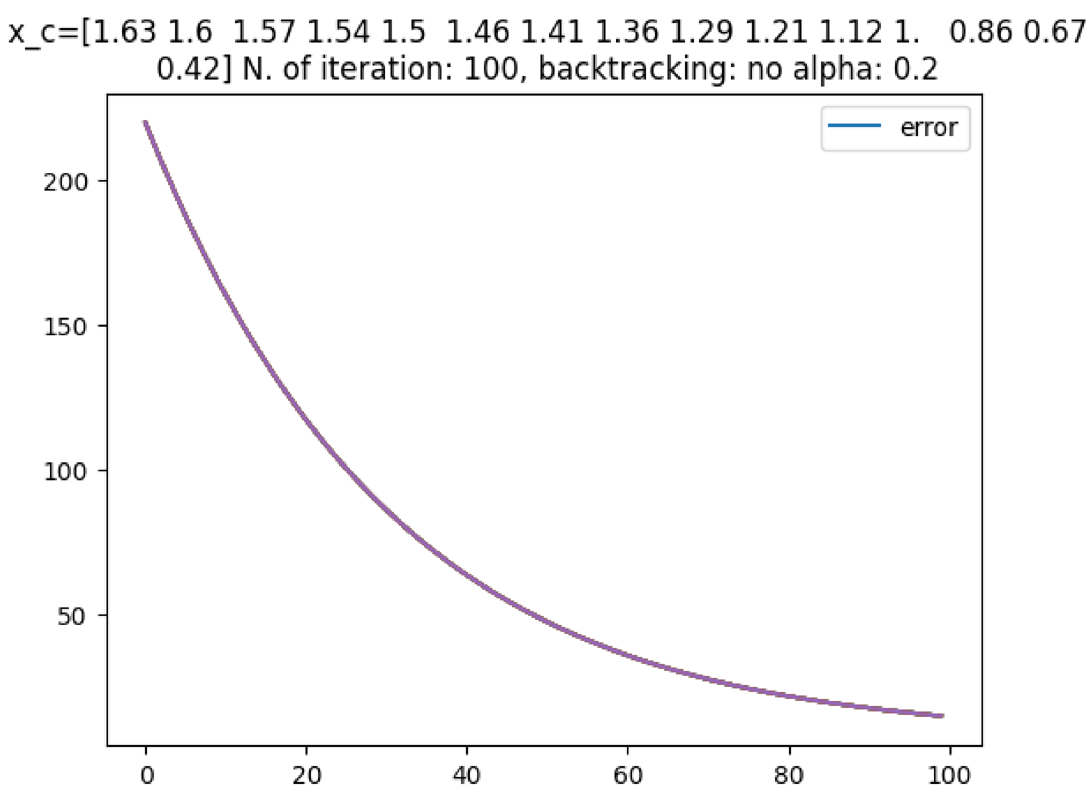

x_c=[1.17 1.14 1.1 1.06 1.01 0.96 0.9 0.85 0.85 1.09] N. of iteration: 100, backtracking: no alpha: 0.01
error


x_c=[1.81 1.77 1.72 1.67 1.6 1.52 1.4 1.25 1.01 0.54] N. of iteration: 100, backtracking: no alpha: 0.2
error
120
100
80
60
40
20
0
20
40
60
80
100

N = 15


x_c=[5.99208298e+34 6.15224507e+34 6.33555016e+34 6.54693822e+34
6.79290087e+34 7.08216634e+34 7.42679457e+34 7.84400901e+34
8.35943550e+34 9.01324025e+34 9.87285441e+34 1.10627513e+35
1.28473569e+35 1.59313784e+35 2.37801057e+35] N. of iteration: 100, backtracking: no alpha: 0.1
1e37
error
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100


x_c=[1.14 1.13 1.1 1.08 1.06 1.03 1.01 0.98 0.95 0.92 0.9 0.89 0.9 0.95
1.08] N. of iteration: 100, backtracking: no alpha: 0.01
error
0
20
40
60
80
100
120
140
160

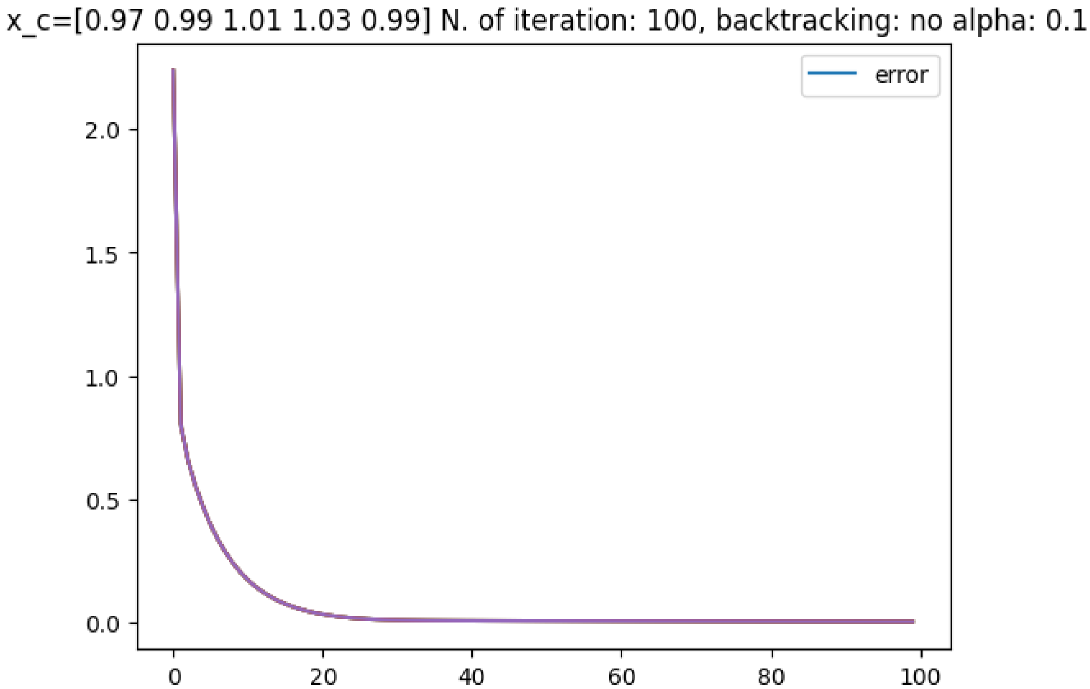

x_c=[1.63 1.6 1.57 1.54 1.5 1.46 1.41 1.36 1.29 1.21 1.12 1.   0.86 0.67
0.42] N. of iteration: 100, backtracking: no alpha: 0.2
error
0
20
40
60
80
100
50
150
200

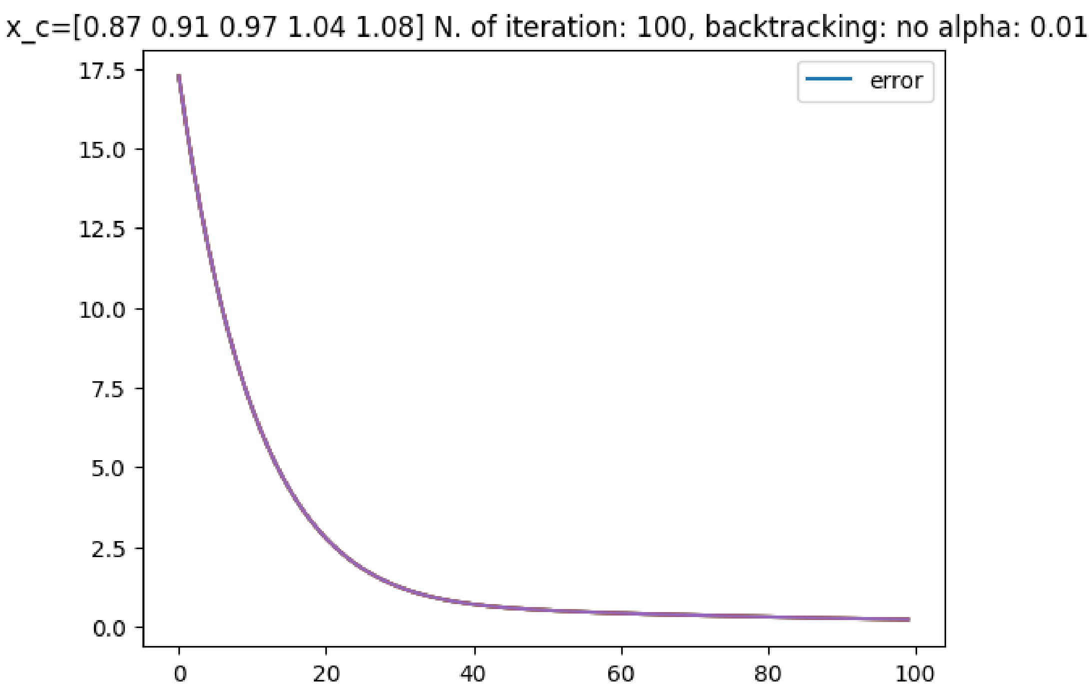
1.4 Function 4

$$f(x) = \frac{1}{2}||Ax - b||_2^2 + \frac{\lambda}{2}||x||_2^2$$

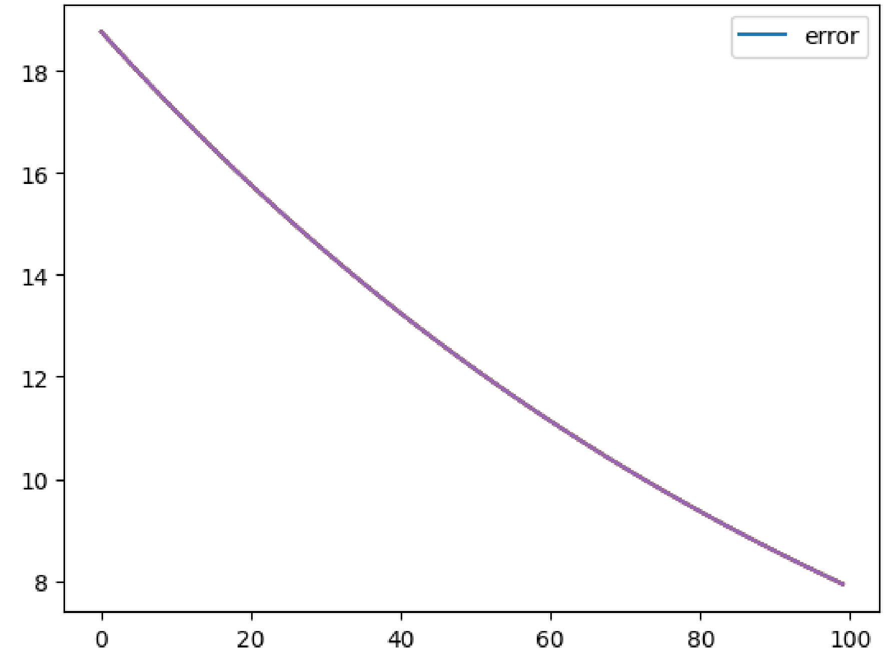
[77]:
```
def f4_builder(lmb):
    def f4(x):
        x = np.array(x)
        x = np.reshape(x, (1, len(x)))
        n, m = x.shape
        x_true = np.ones((1, n))
        v = np.linspace(0, 1, n, endpoint=True)
        A = numpy.vander(v)
        b = A @ x_true
        return 1/2 * np.linalg.norm(A @ x - b, 2)**2 + lmb/2 * np.linalg.
 ↪norm(x)**2
    return f4

def grad_f4_builder(lmb):
    def grad_f4(x):
        n = len(x)
        v = np.linspace(0,1,n)
        A = np.vander(v)
        x_true = np.ones(n).T
        b = A @ x_true
        return np.array(A.T@(A@x-b)) + lmb*x
    return grad_f4


n = 5
lmbs = np.linspace(0, 1, 3)
x0 = [0 for i in range(n)]


for lmb in lmbs:
    print("Lambda: ", lmb)
    f4 = f4_builder(lmb)
    grad_f4 = grad_f4_builder(lmb)
    test_function(f4, grad_f4, x0, kmax)
```

```
Lambda:  0.0
```


x_c=[0.97 0.99 1.01 1.03 0.99] N. of iteration: 100, backtracking: no alpha: 0.1
error
2.0
1.5
1.0
0.5
0.0
0
20
40
60
80
100

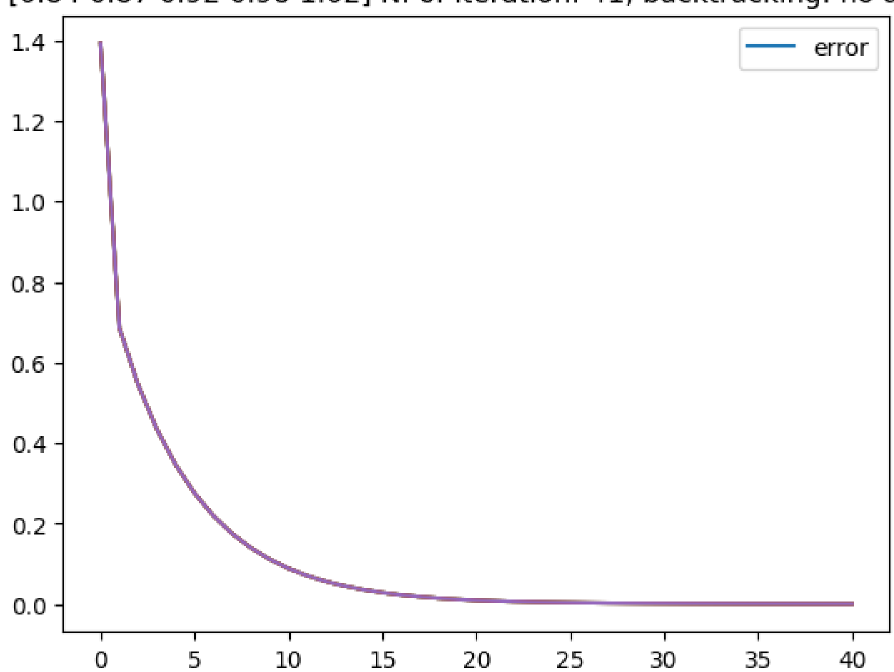

x_c=[0.87 0.91 0.97 1.04 1.08] N. of iteration: 100, backtracking: no alpha: 0.01
error
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
20
40
60
80
100

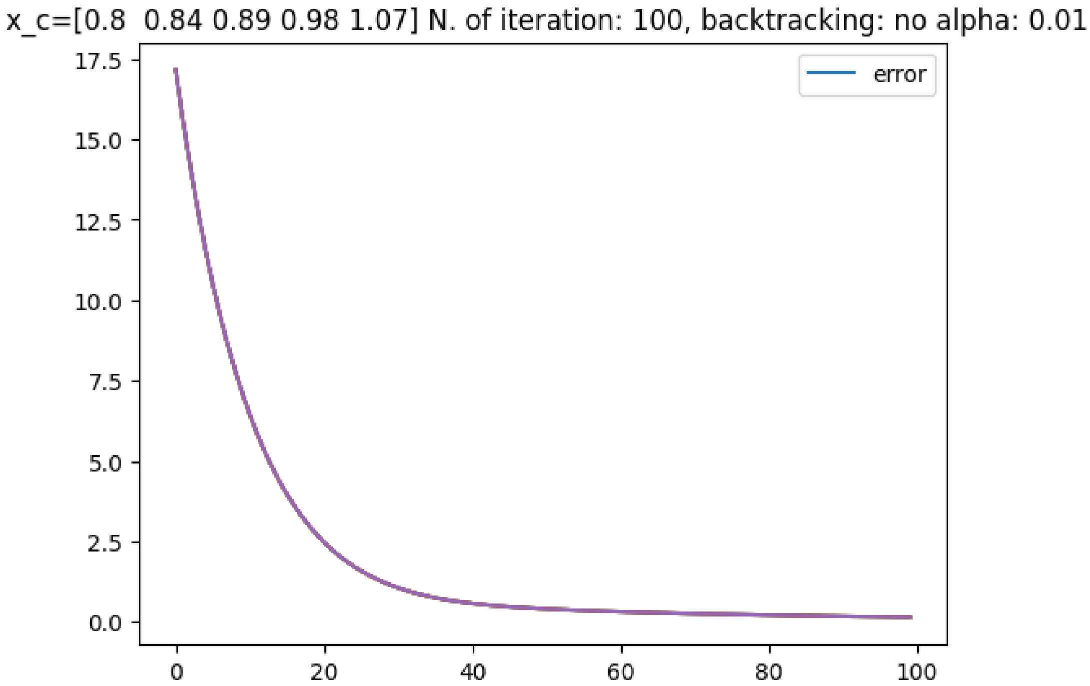
x_c=[0.41 0.44 0.49 0.57 0.79] N. of iteration: 100, backtracking: no alpha: 0.2

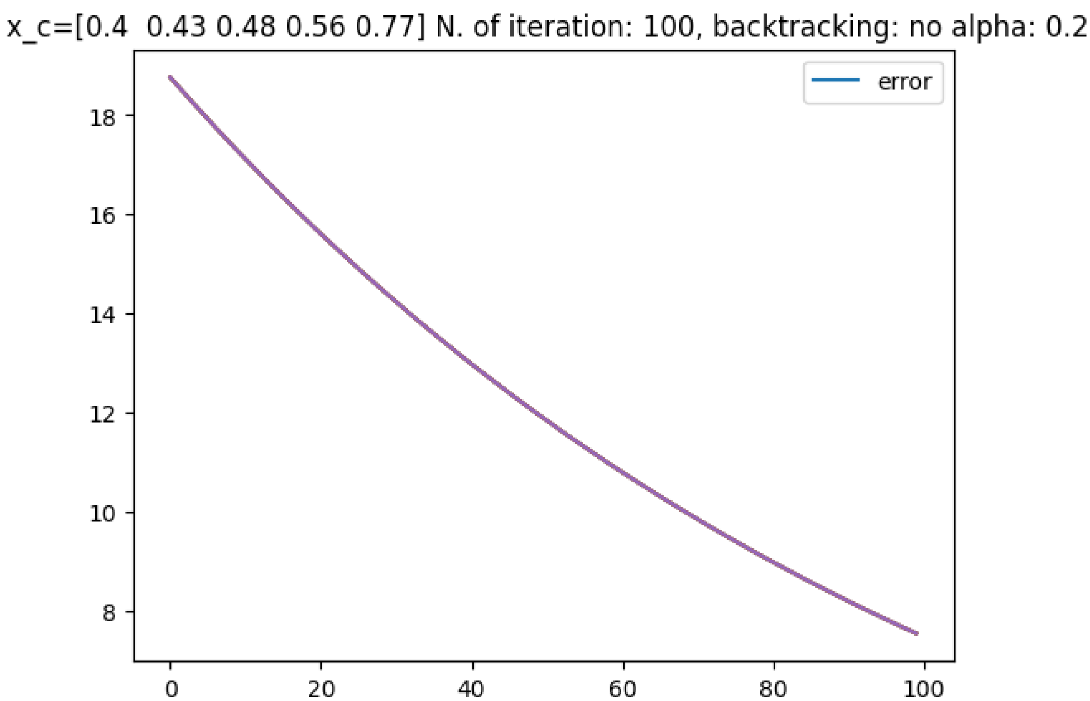



```
Lambda:  0.5
```

x_c=[0.84 0.87 0.92 0.98 1.02] N. of iteration: 41, backtracking: no alpha: 0.1

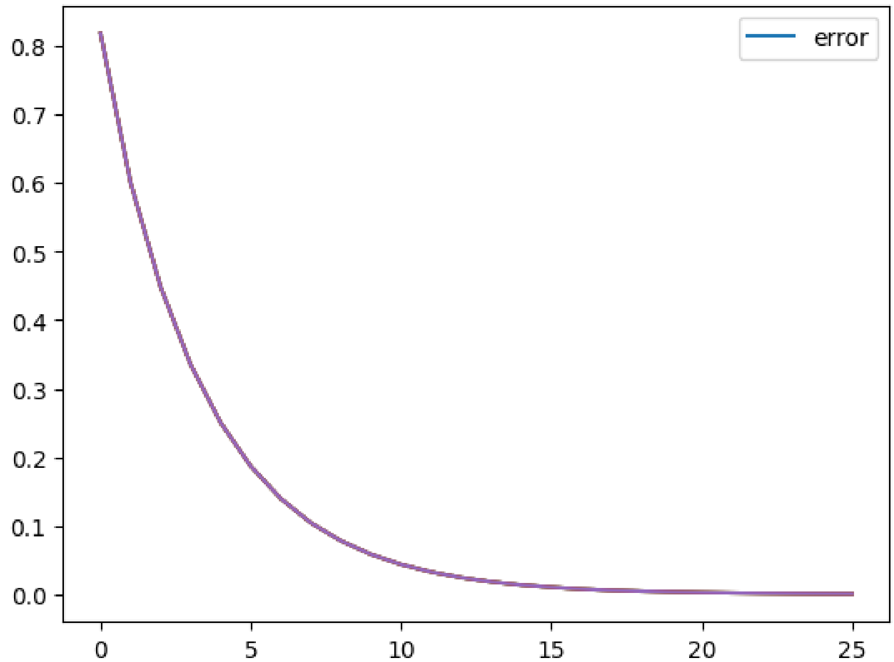



x_c=[0.8 0.84 0.89 0.98 1.07] N. of iteration: 100, backtracking: no alpha: 0.01
error
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
20
40
60
80
100

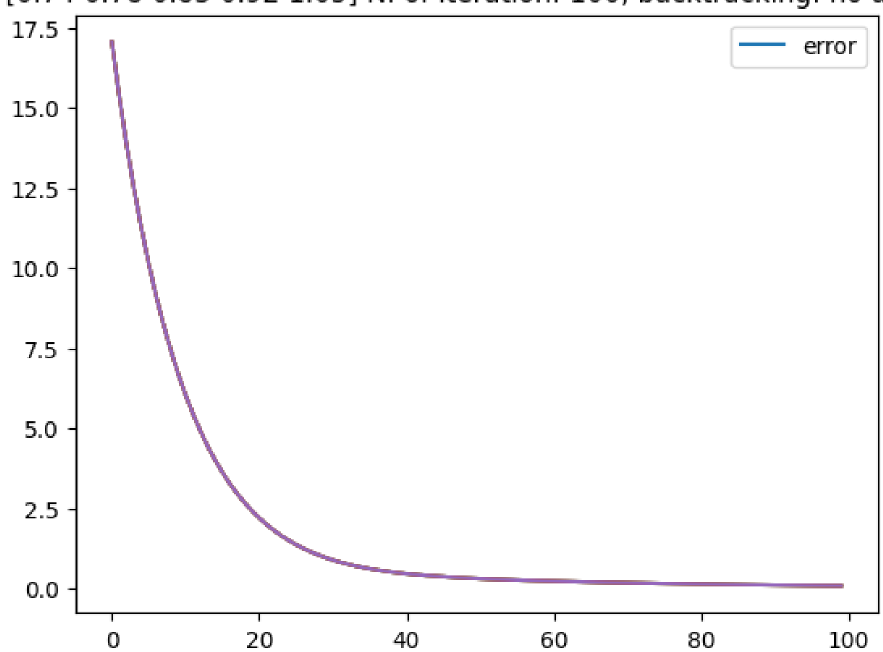

x_c=[0.4 0.43 0.48 0.56 0.77] N. of iteration: 100, backtracking: no alpha: 0.2
error
18
16
14
12
10
8
0
20
40
60
80
100

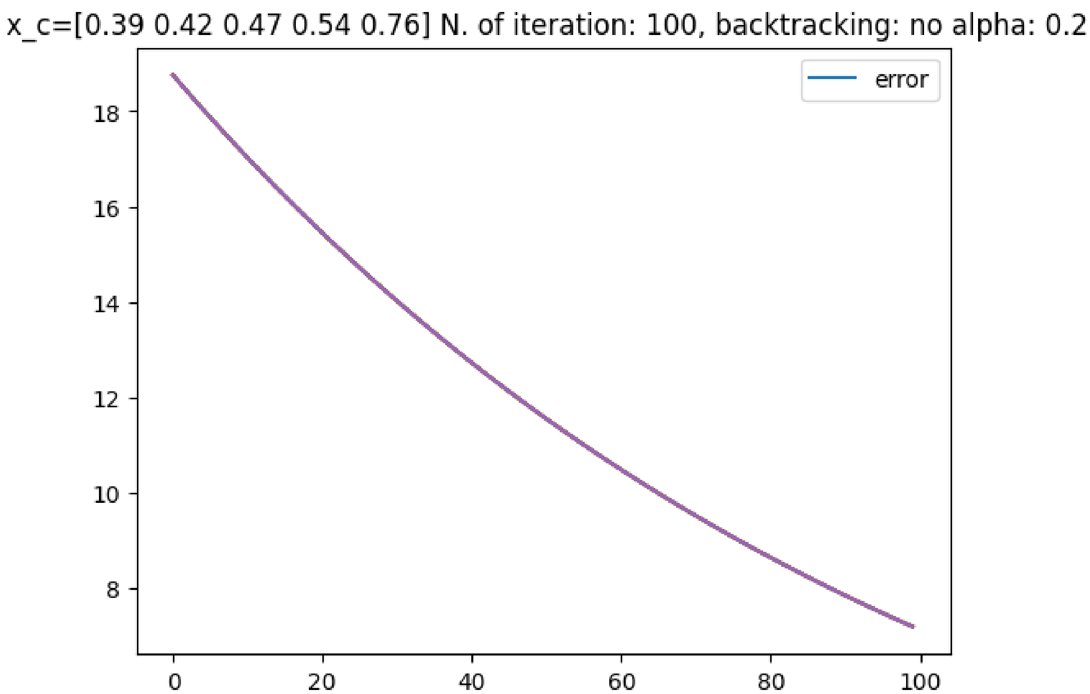
```
Lambda:  1.0
```

x_c=[0.76 0.79 0.84 0.92 1.02] N. of iteration: 26, backtracking: no alpha: 0.1



x_c=[0.74 0.78 0.83 0.92 1.05] N. of iteration: 100, backtracking: no alpha: 0.01

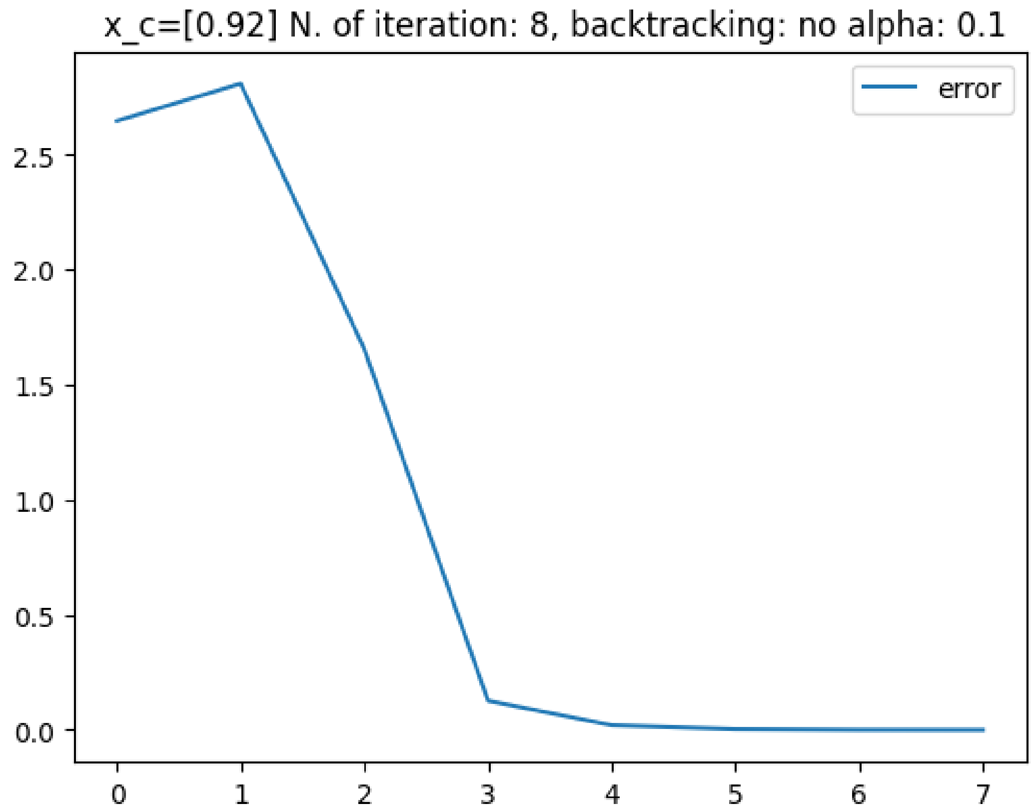


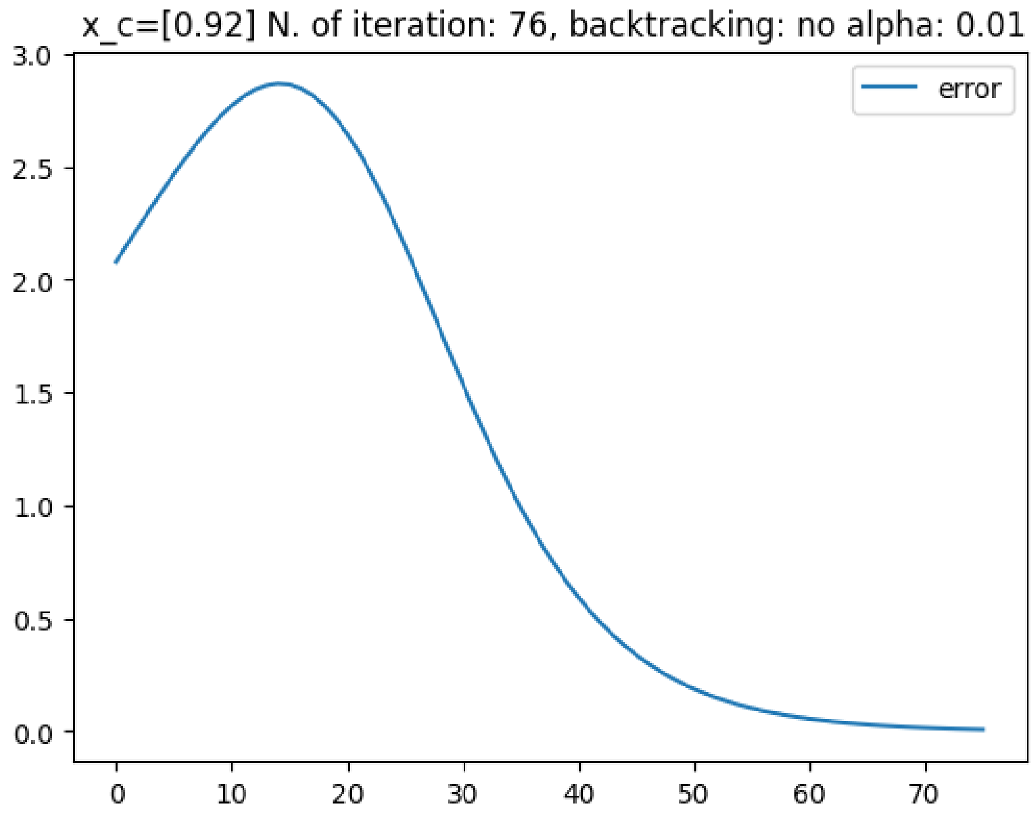


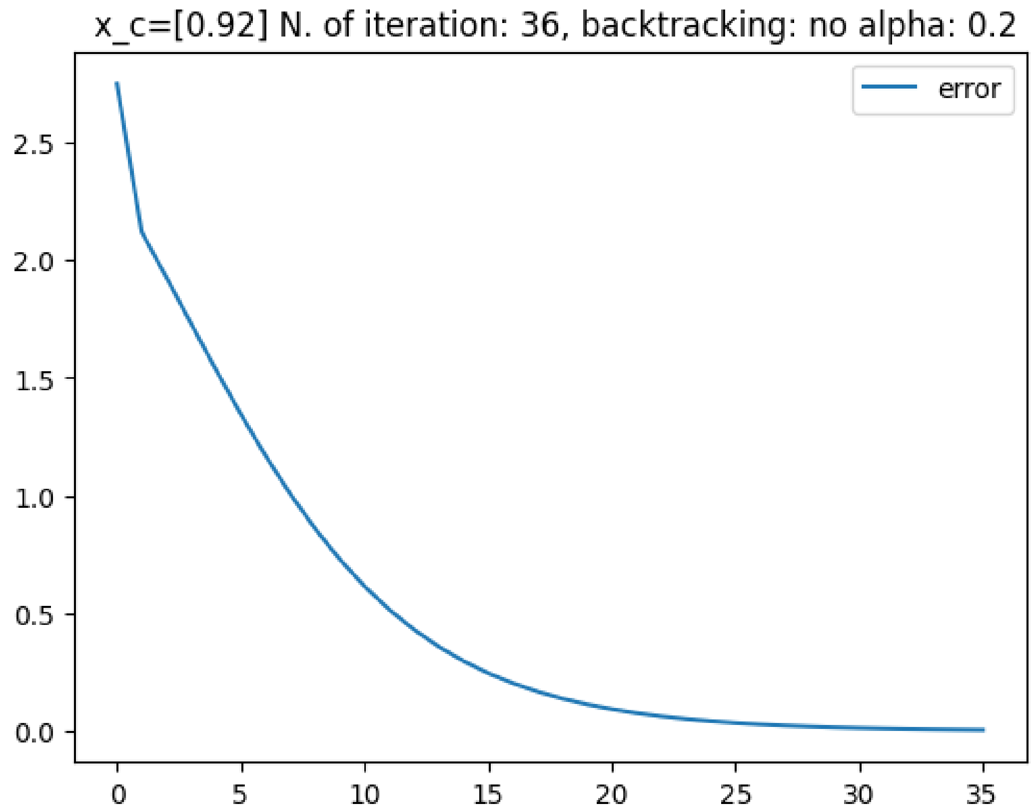
1.4.1 Function 5

$$f(x) = x^4 + x^3 - 2x^2 - 2x$$

[78]:
```
def f5(x):
    return np.power(x,4) + np.power(x, 3) - 2*np.power(x,2) - 2*x

def grad_f5(x):
    return np.array(4*np.power(x, 3) + 3*np.power(x,2) - 4*x - 2)


N = np.arange(5, 20, 5)

x0 = [0.]
test_function(f5, grad_f5, x0, kmax)
```

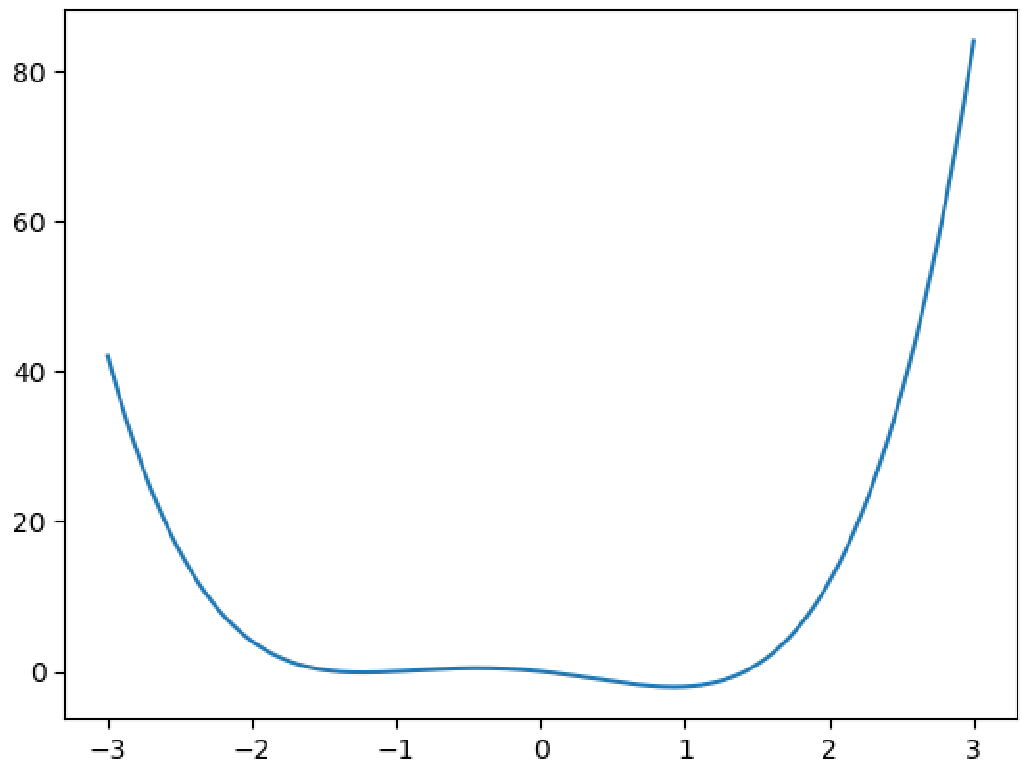

x_c=[0.92] N. of iteration: 8, backtracking: no alpha: 0.1
error
2.5
2.0
1.5
1.0
0.5
0.0
0
1
2
3
4
5
6
7


x_c=[0.92] N. of iteration: 76, backtracking: no alpha: 0.01
error
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
10
20
30
40
50
60
70

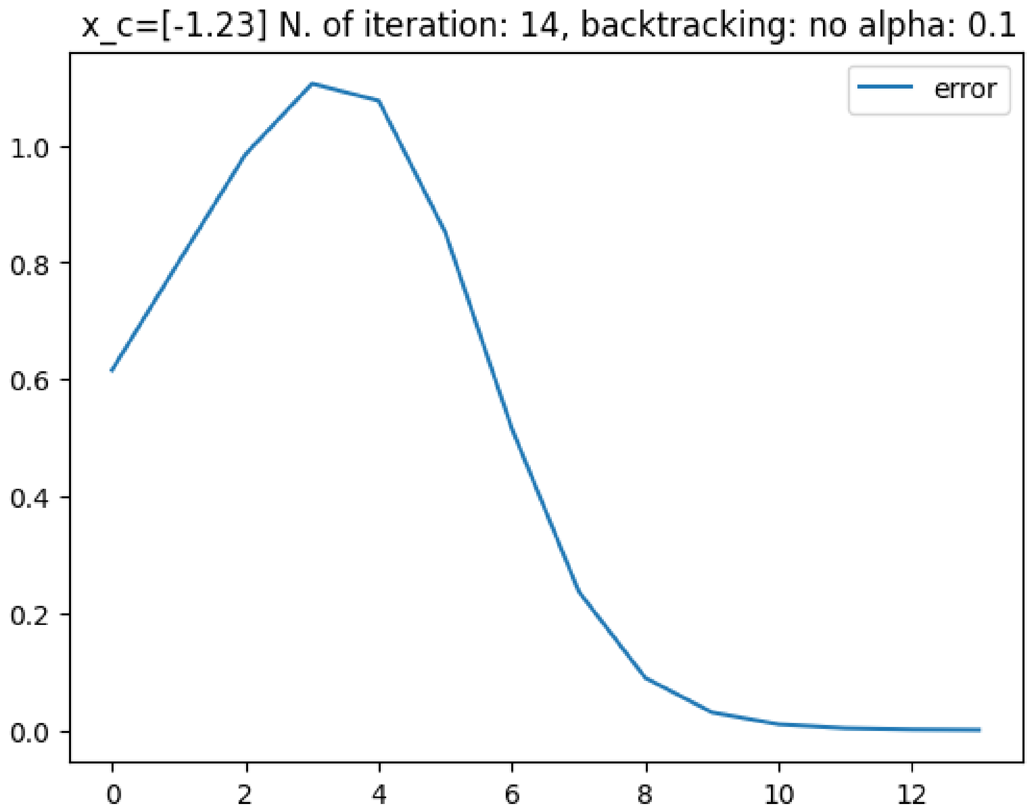


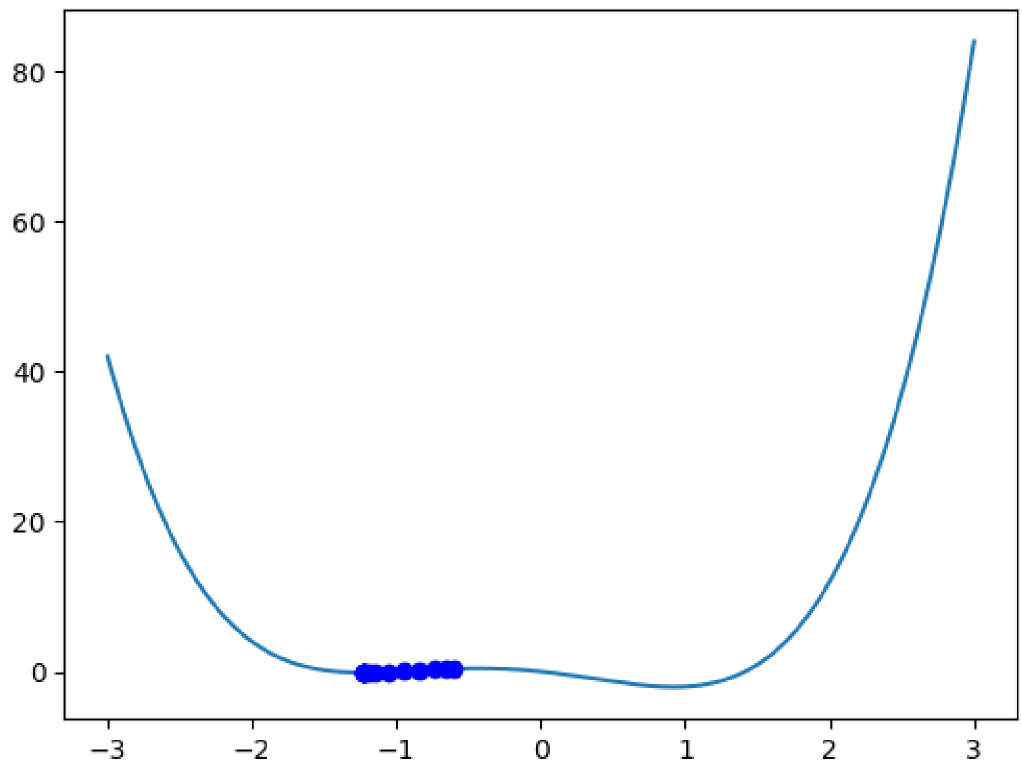
Only for the non-convex function defined in 5, plot it in the interval $[-3, 3]$ and test the convergence point of GD with different values of x0 and different step-sizes. Observe when the convergence point is the global minimum and when it stops on a local minimum or maximum.

[79]:
```
x_5 = np.linspace(-3, 3, 1000)
plt.plot(x_5, f5(x_5))
plt.show()

starting_points = [-2, 0, 2]

for x0 in starting_points:
    x0 = np.array([x0])
    test_function(f5, grad_f5, x0, kmax, f5=True)
```

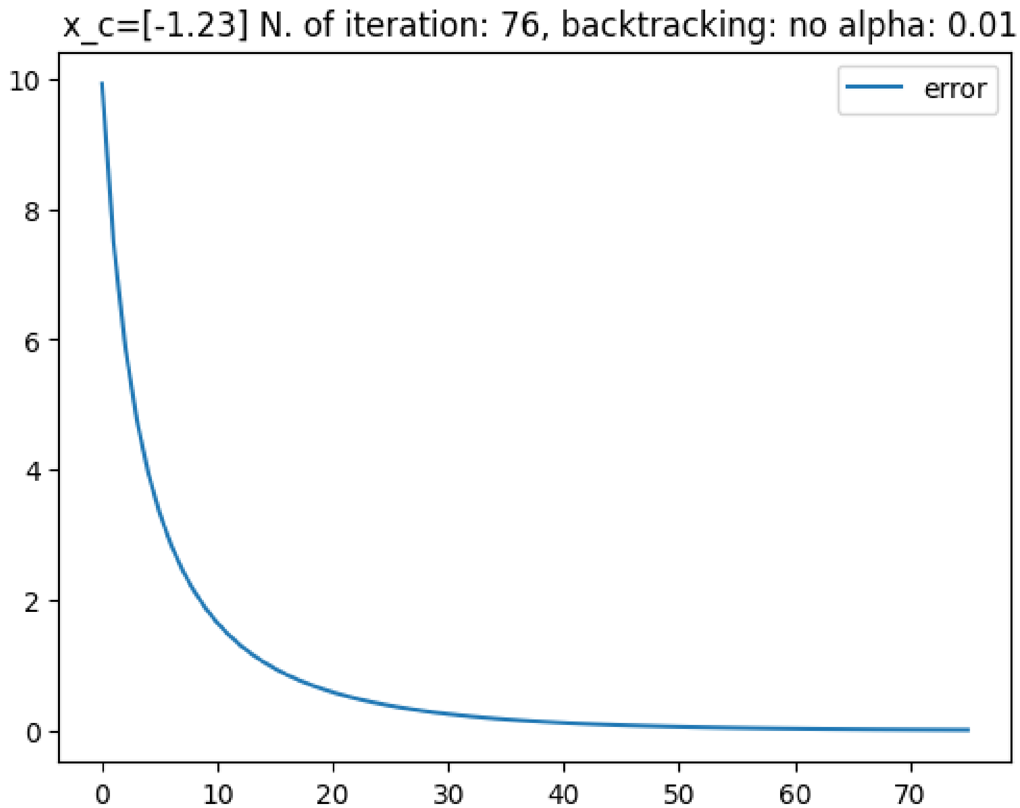


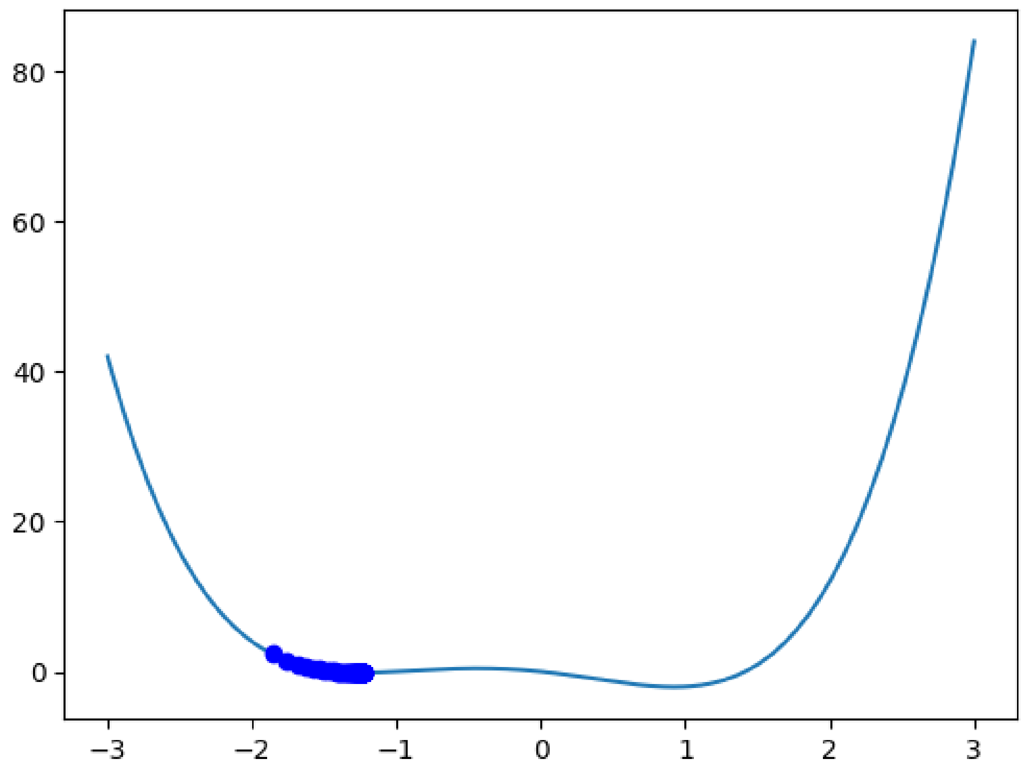
80
60
40
20
0
−3
−2
−1
0
1
2
3

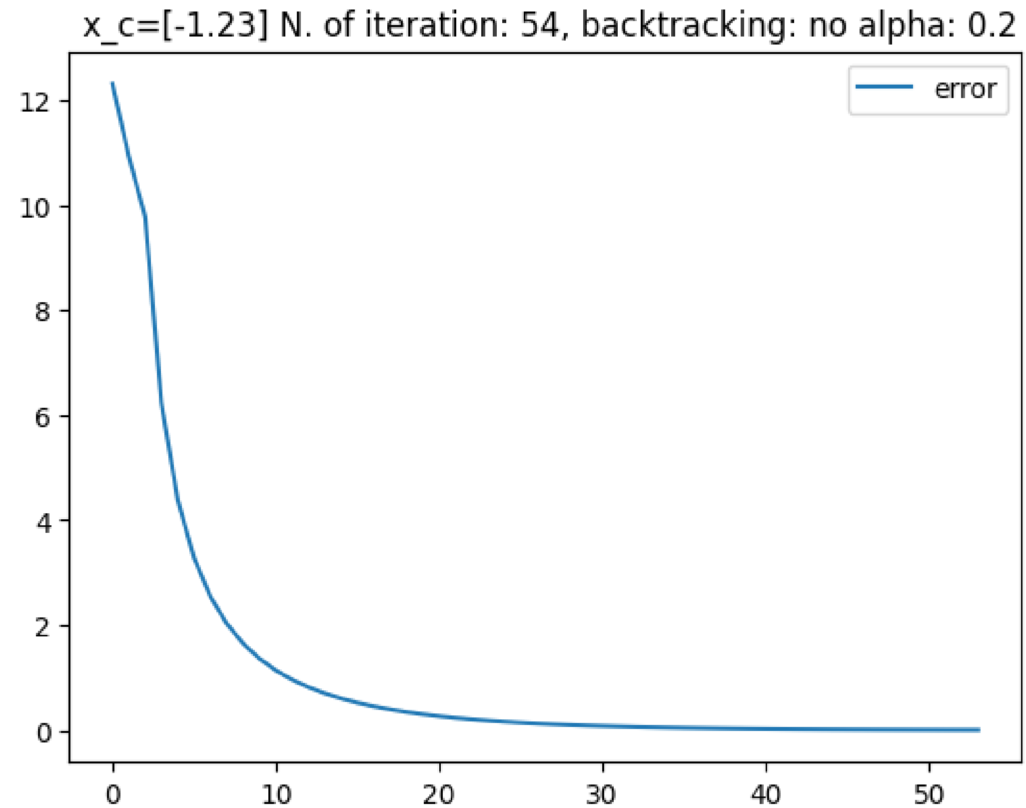

x_c=[-1.23] N. of iteration: 14, backtracking: no alpha: 0.1
error
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
12

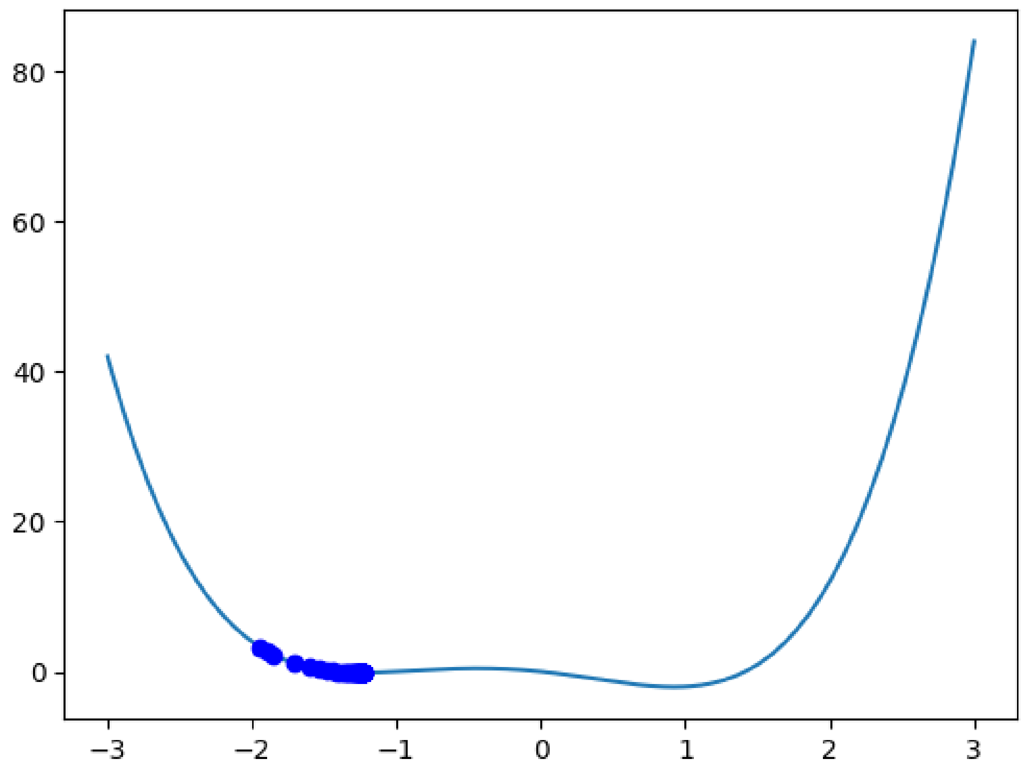


80
60
40
20
0
−3
−2
−1
0
1
2
3

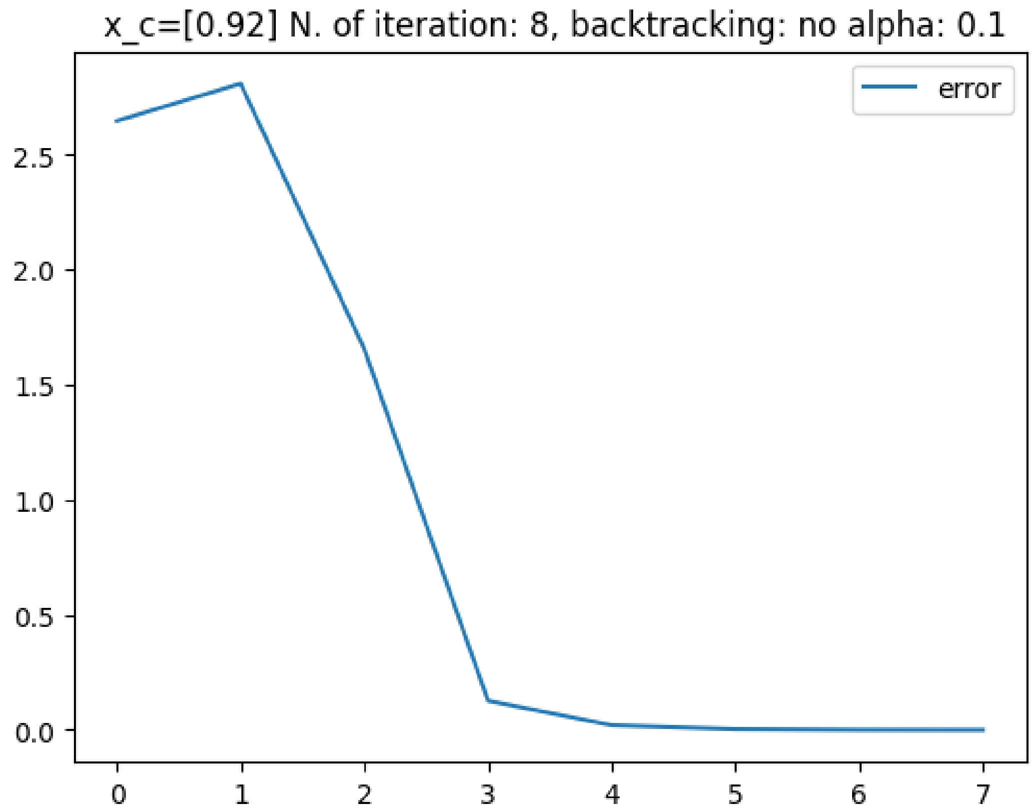

x_c=[-1.23] N. of iteration: 76, backtracking: no alpha: 0.01
error
10
8
6
4
2
0
0
10
20
30
40
50
60
70

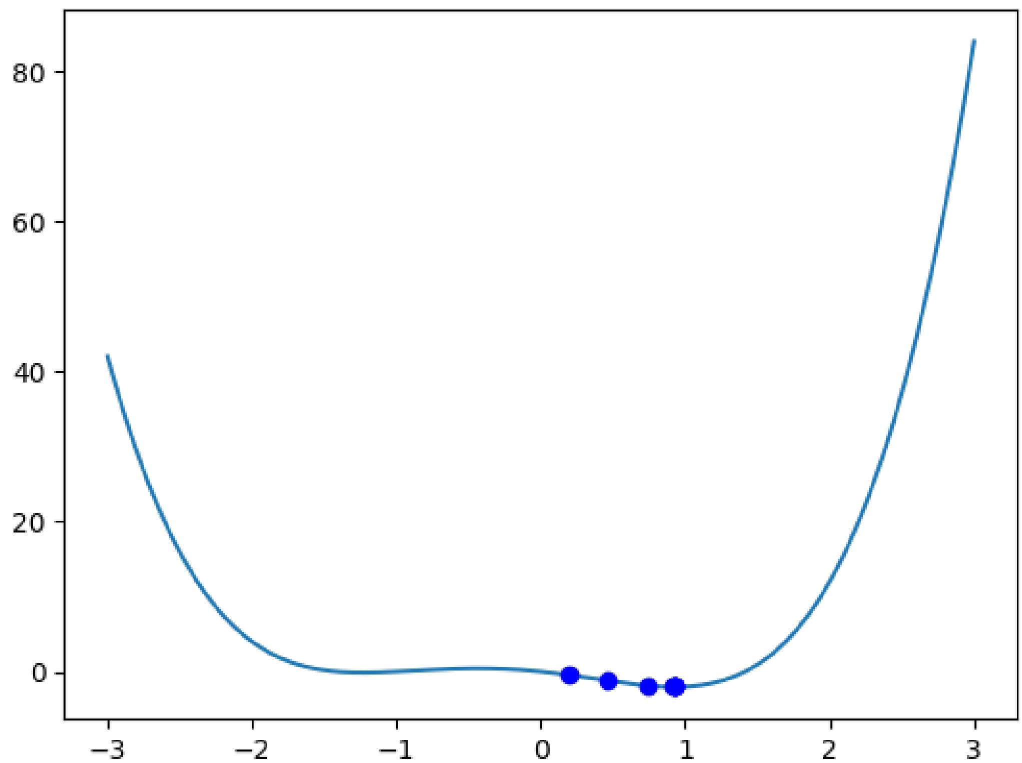

80
60
40
20
0
−3
−2
−1
0
1
2
3

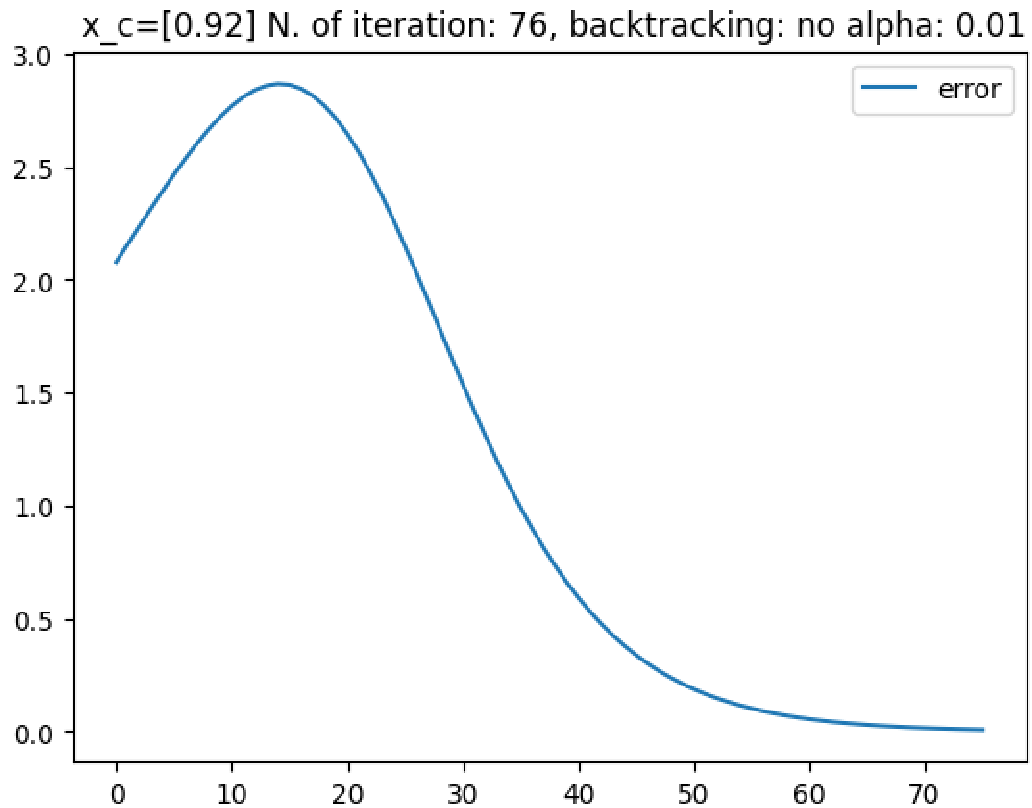

x_c=[-1.23] N. of iteration: 54, backtracking: no alpha: 0.2
error
0
2
4
6
8
10
12
0
10
20
30
40
50

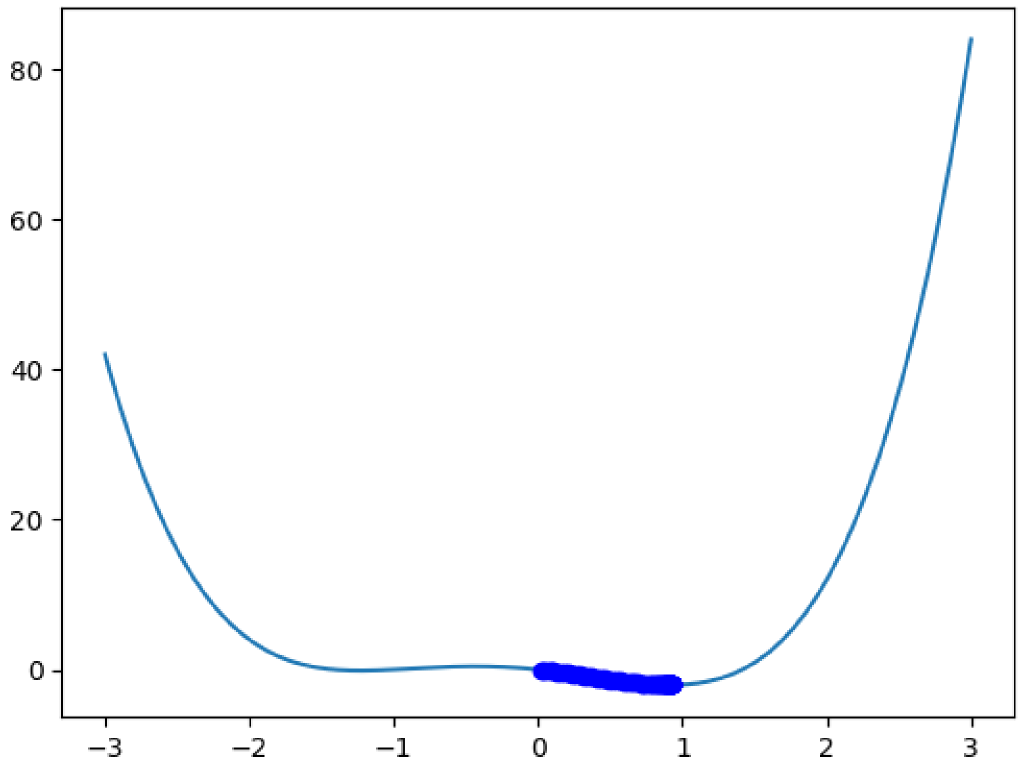


80
60
40
20
0
−3
−2
−1
0
1
2
3

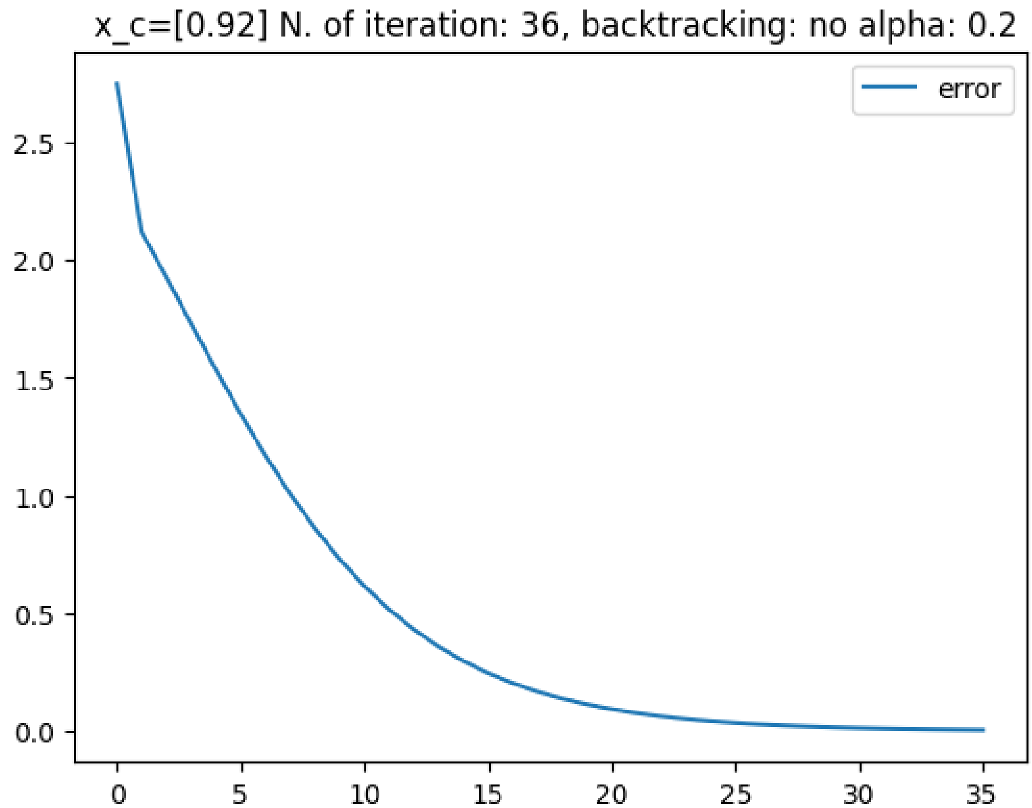

x_c=[0.92] N. of iteration: 8, backtracking: no alpha: 0.1
error
2.5
2.0
1.5
1.0
0.5
0.0
0
1
2
3
4
5
6
7

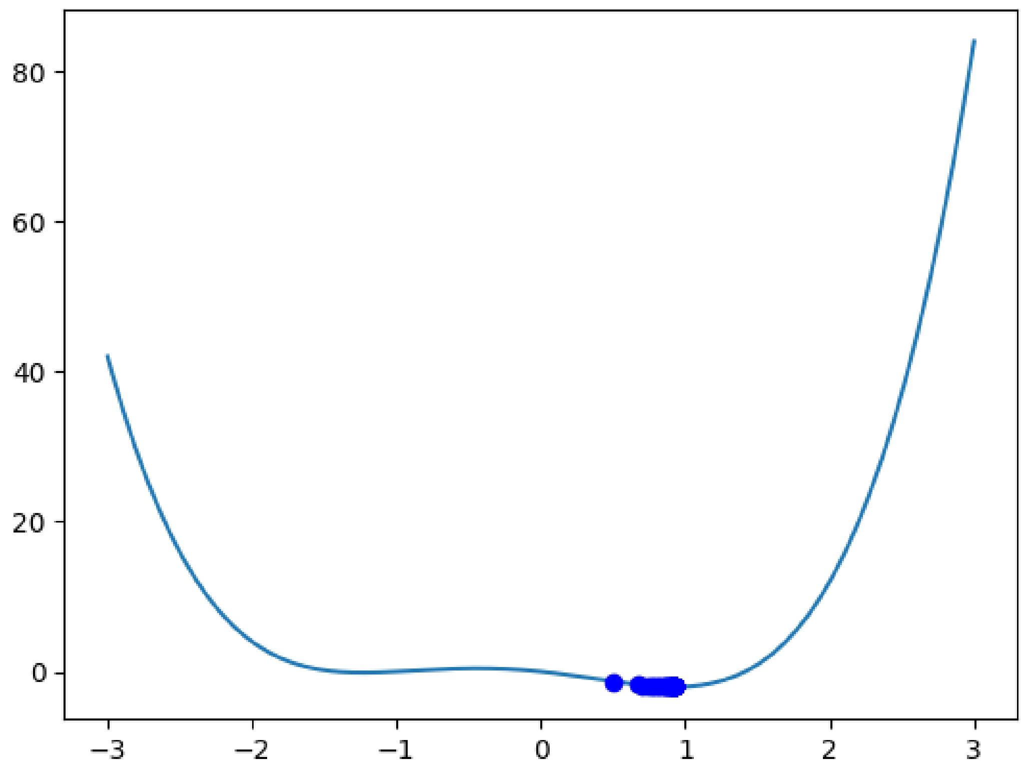

80
60
40
20
0
−3
−2
−1
0
1
2
3

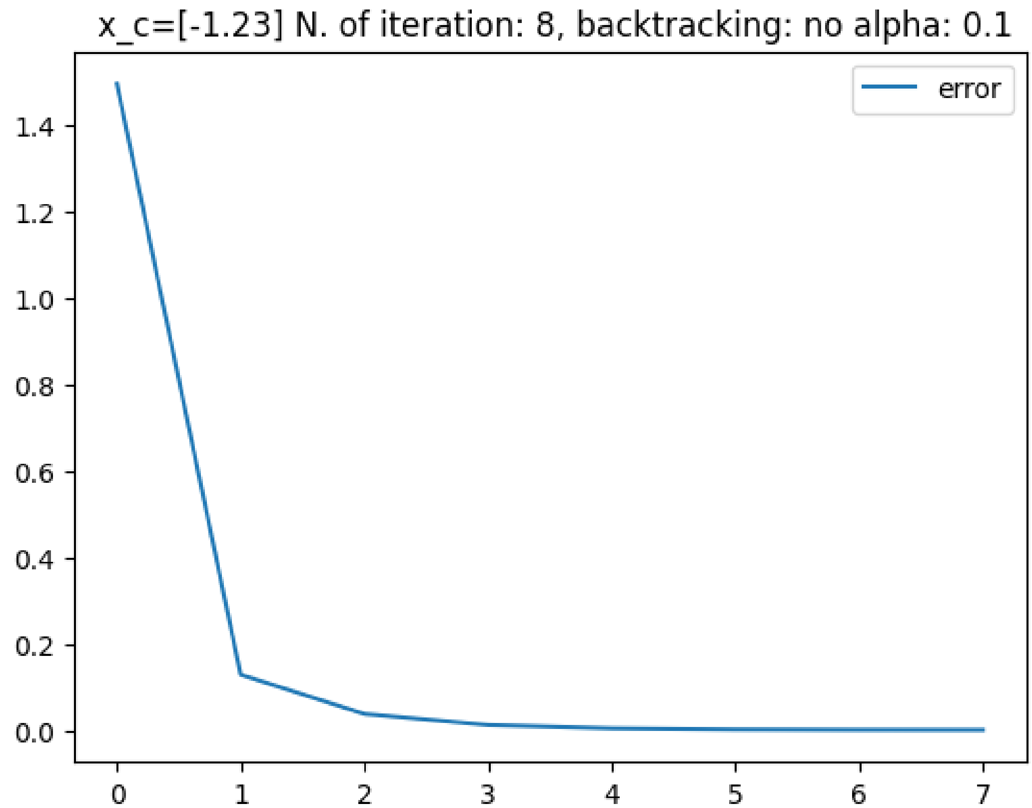

x_c=[0.92] N. of iteration: 76, backtracking: no alpha: 0.01
error
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
10
20
30
40
50
60
70

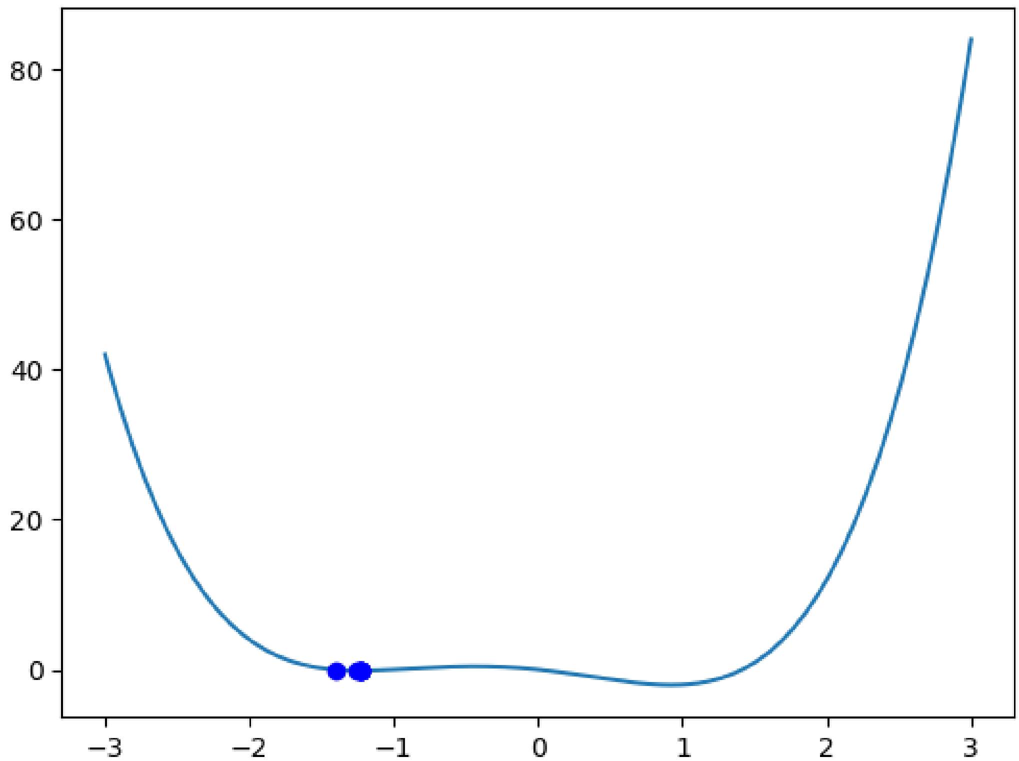


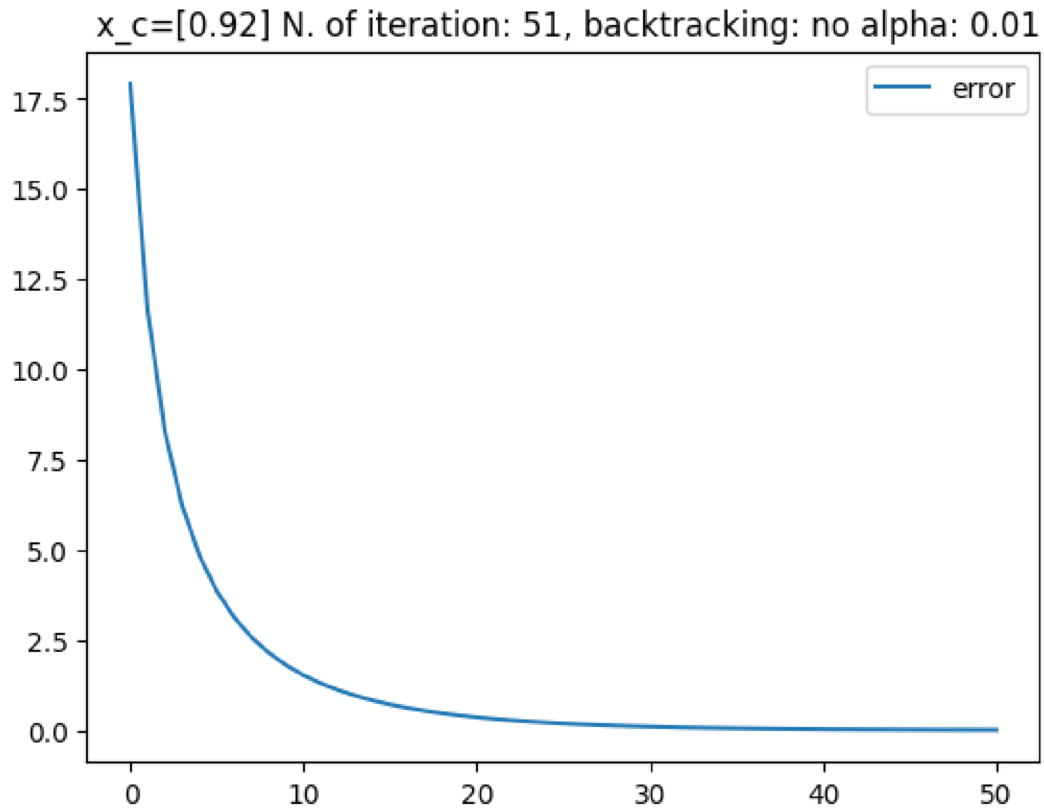
80
60
40
20
0
−3
−2
−1
0
1
2
3

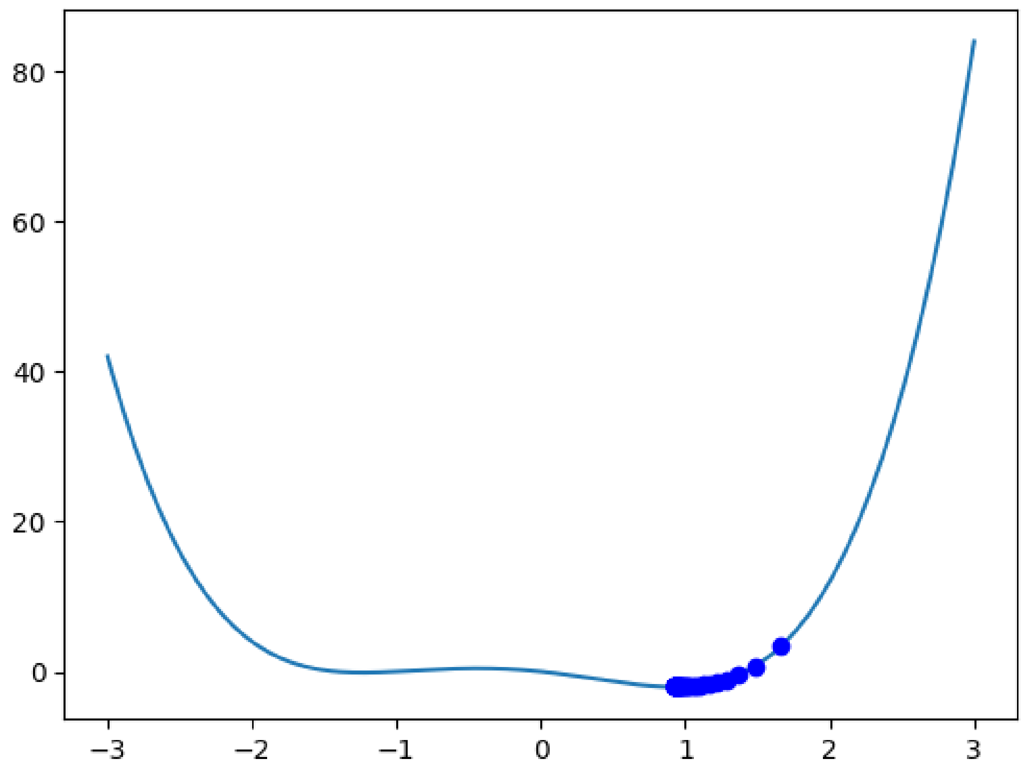

x_c=[0.92] N. of iteration: 36, backtracking: no alpha: 0.2
error
2.5
2.0
1.5
1.0
0.5
0.0
0
5
10
15
20
25
30
35

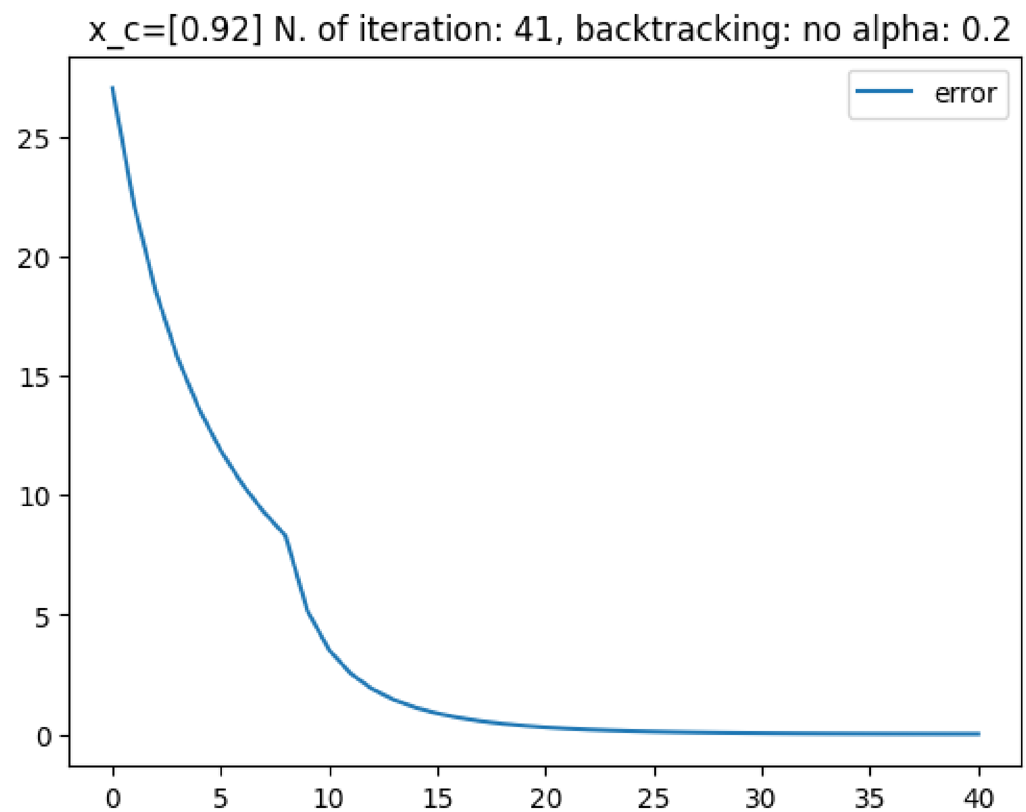

80
60
40
20
0
−3
−2
−1
0
1
2
3

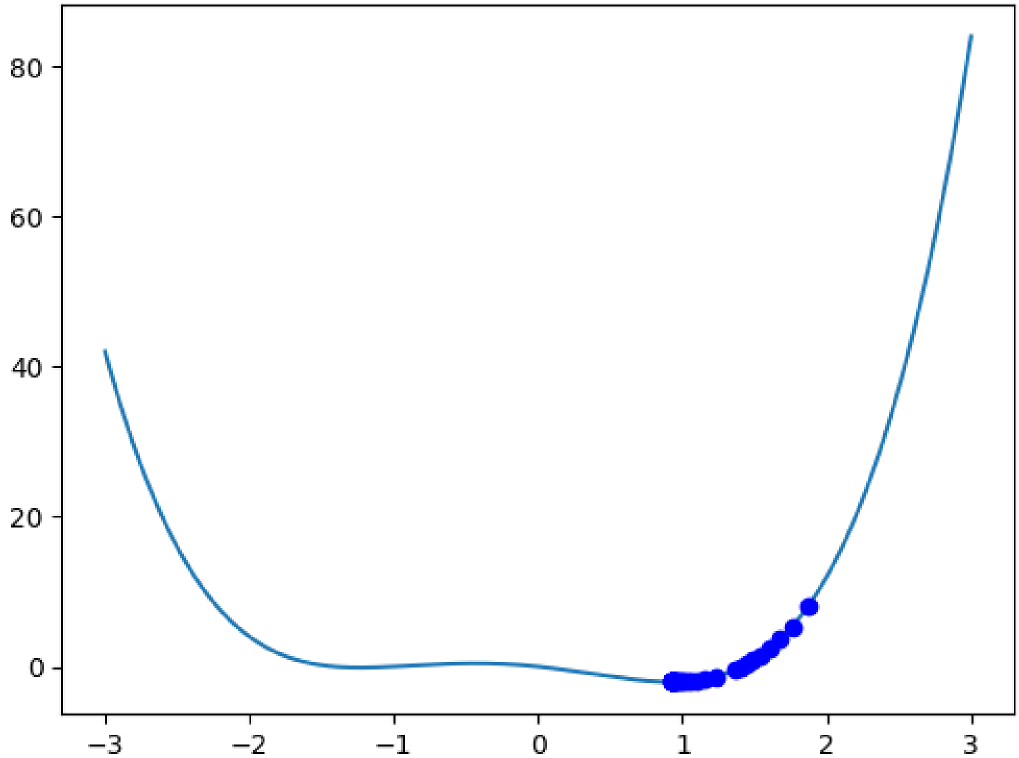

x_c=[-1.23] N. of iteration: 8, backtracking: no alpha: 0.1
error
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
1
2
3
4
5
6
7


80
60
40
20
0
−3
−2
−1
0
1
2
3


x_c=[0.92] N. of iteration: 51, backtracking: no alpha: 0.01
error
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
10
20
30
40
50


80
60
40
20
0
−3
−2
−1
0
1
2
3

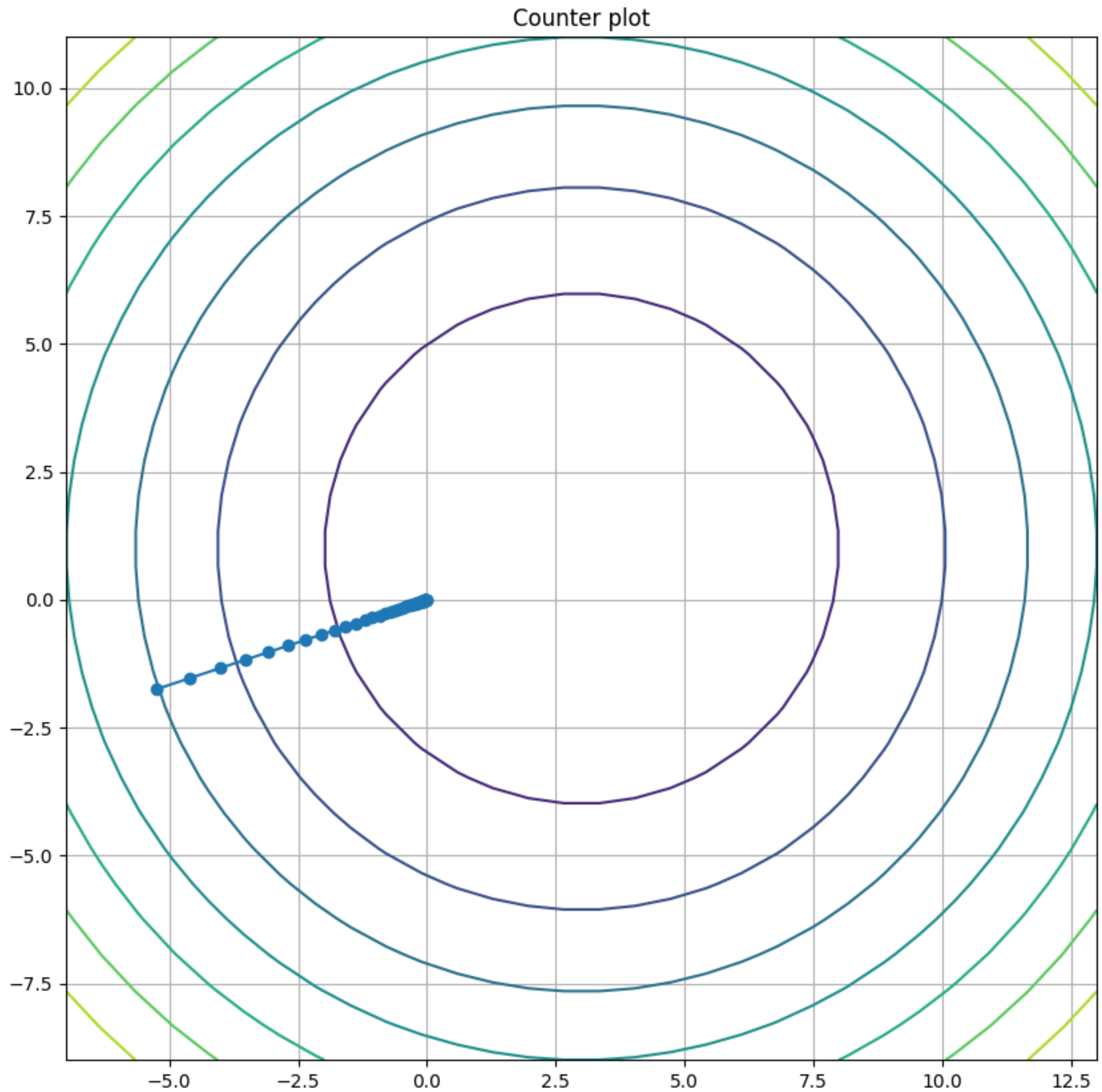

x_c=[0.92] N. of iteration: 41, backtracking: no alpha: 0.2
error
0
5
10
15
20
25
0
5
10
15
20
25
30
35
40

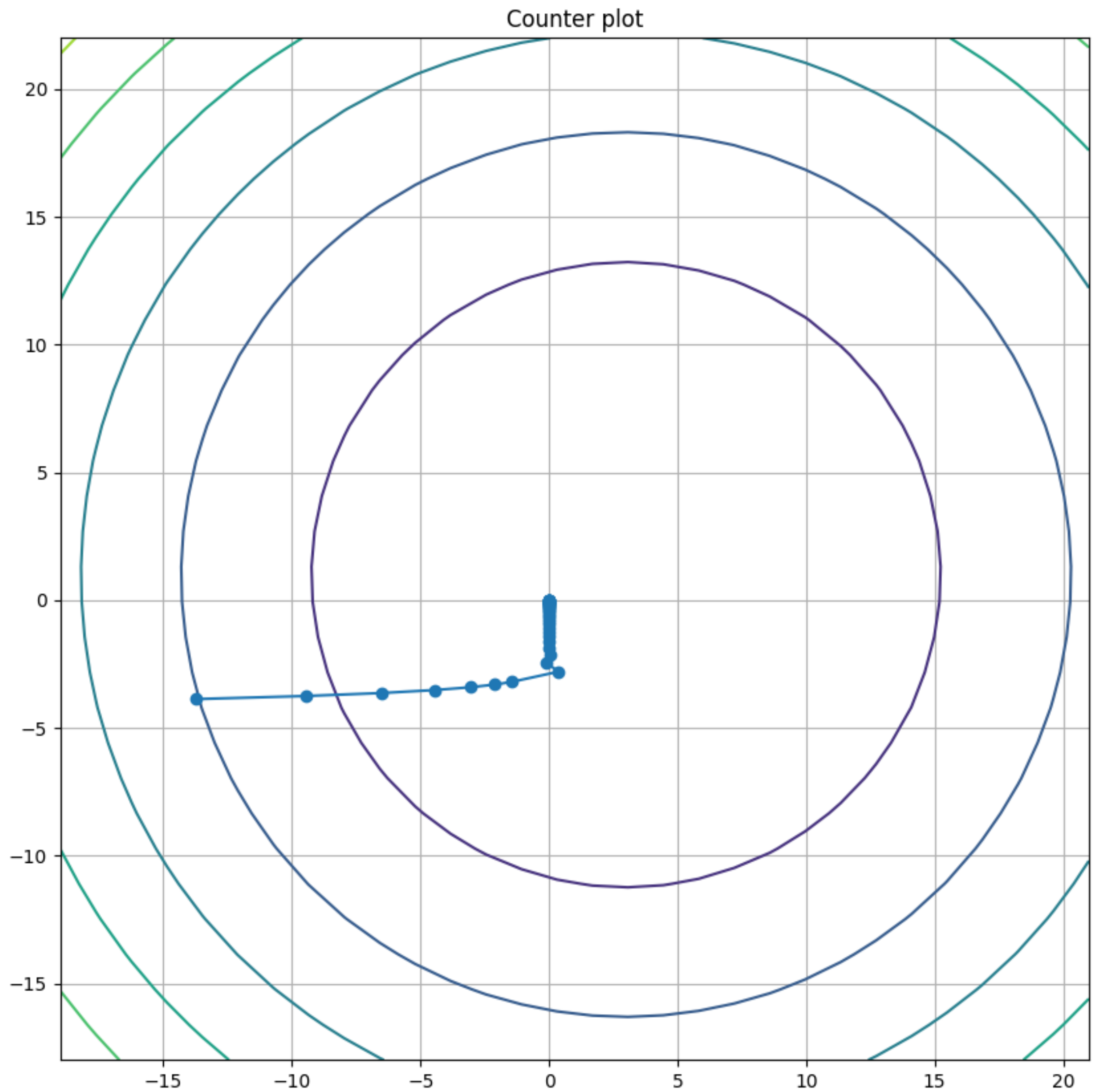


Hard (optional): For the functions 1 and 2, plot the contour around the minimum and the path defined by the iterations (following the example seen during the lesson). See plt.contour to do that.

[80]:
```
def contour(f, grad_f, x0, x_true, radius, tolx, tolf, kmax):
    x11, x12 = x_true

    xv = np.linspace(x11 - radius, x11 + radius, 30)
    yv = np.linspace(x12 - radius, x12 + radius, 30)
    xx, yy = np.meshgrid(xv, yv)

    x, f_val, grads, err = GD(f, grad_f, x0, tolf, tolx, kmax, back=True)
    zz = f1((xx, yy))

    plt.figure(figsize=(10,10))
    plt.contour(xx, yy, zz)
    plt.plot(grads[:, 0], grads[:, 1], 'o-')
    plt.title("Counter plot")
    plt.grid()
    plt.show()


contour(f1, grad_f1, (0, 0), x_true1, 10, tolx, tolf, kmax)
```

```
contour(f2, grad_f2, (0, 0), x_true2, 20, tolx, tolf, kmax)
```





2 Optimization via Stochastic Gradient DescentInput:

l: the function l(w; D) we want to optimize. It is supposed to be a Python function, not an array. grad_l: the gradient of l(w; D). It is supposed to be a Python function, not an array. w0: an n-dimensional array which represents the initial iterate. By default, it should be randomly sampled. data: a tuple (x, y) that contains the two arrays x and y, where x is the input data, y is the output data. batch_size: an integer. The dimension of each batch. Should be a divisor of the number of data. n_epochs: an integer. The number of epochs you want to reapeat the iterations. Output: w: an array that contains the value of w_k FOR EACH iterate w_k (not only the latter). f_val: an array that contains the value of l(w_k; D) FOR EACH iterate w_k ONLY after each epoch. grads: an array that contains the value of grad_l(w_k; D) FOR EACH iterate w_k ONLY after each epoch. err: an array the contains the value of ||grad_l(w_k; D)||_2 FOR EACH iterate w_k

ONLY after each epoch.

[81]:
```python
# Import the data MNIST

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy as sp

dataset = pd.read_csv("data.csv")
dataset = np.array(dataset)


def split(X, Y, Ntrain):
    d, N = X.shape

    idx = np.arange(N)
    np.random.shuffle(idx)

    train_idx = idx[:Ntrain]
    test_idx = idx[Ntrain:]

    Xtrain = X[:, train_idx]
    Ytrain = Y[train_idx]

    Xtest = X[:, test_idx]
    Ytest = Y[test_idx]

    return (Xtrain, Xtest, Ytrain, Ytest)
```

[82]:
```python
def create_dataset(dataset, digits=[3,6], Ntrain=4600):
    digits = [3, 6]

    y = dataset[:, 0]
    x = dataset[:, 1:].T

    Y = y#y.reshape((len(y), 1))
    X = np.concatenate((np.ones((1, len(y))), x), axis=0)

    I1 = (Y == digits[0])  # (Y[:, 0] == digits[0])
    I2 = (Y == digits[1])  # (Y[:, 0] == digits[1])
    X1 = X[:, I1]
    X2 = X[:, I2]
    Y1 = np.zeros((len(Y[I1]), ))
    Y2 = np.ones((len(Y[I2]), ))
    #Y1 = Y[I1]
    #Y2 = Y[I2]
```

```
    X = np.concatenate((X1, X2), axis=1)
    Y = np.concatenate((Y1, Y2))

    d, N = X.shape

    #Ntrain = 4600#int(N/3*2)

    x_train, x_test, y_train, y_test = split(X, Y, Ntrain)
    return x_train, x_test, y_train, y_test
```

[83]:
```
x_train, x_test, y_train, y_test = create_dataset(dataset)
print(x_train.shape, x_test.shape, y_train.shape)
```

```
(785, 4600) (785, 3888) (4600,)
```

[84]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def f(x, w):
    x_cup = x
    return sigmoid(x_cup.T @ w)


def grad_f(x):
    return np.exp(-x) / (np.exp(-x) + 1) ** 2


def l(w, x, y):
    return np.mean(np.linalg.norm(f(x, w)-y)**2)


def loss_grad(w, x, y):
    d, n = x.shape

    y = np.array(y)

    sum = 0
    for i in range(n):
        z = f(x[:, i], w)
        sum += z * (1 - z) * x[:, i].T * (z - y[i])

    return sum / n

def grad_l(w, x, y):
    x_cup = x
```

```
    return np.mean(sigmoid(x_cup.T @ w)*(1-sigmoid(x_cup.T @ w)) * x_cup.T *␣
↪(f(x, w) - y))
```

[85]:
```
n_epochs = 50
```

[86]:
```
batch_size = 15

def batch(x, y, batch_size):
    n = x.shape[1]
    idx = np.arange(n)
    np.random.shuffle(idx)
    n_batches = n // batch_size
    for i in range(n_batches):
        batch_index = idx[i*batch_size:(i+1)*batch_size]
        yield x[:, batch_index], y[batch_index]


def SGD(l, grad_l, w0, data, batch_size, n_epochs=50):

    shape = (n_epochs, *w0.shape)
    x_, y_ = data
    d, n = x_.shape
    w = np.zeros(shape)
    f_val = np.zeros((n_epochs, 1))
    grads = np.zeros((n_epochs, 1, w0.shape[0]))
    err = np.zeros((n_epochs, 1))

    #print("W0 shape: ", w0.shape)
    alpha = 1e-2
    w_old = w0
    w_k = w_old

    for epoch in range(n_epochs):
        batch_iterator = batch(x_, y_, batch_size)
        for x, y in batch_iterator:
            grad = grad_l(w_old, x, y)
            w_k = w_old - alpha * grad
            w_old = w_k
        w[epoch] = w_k
        f_val[epoch] = l(w_k, x_, y_)
        grads[epoch] = grad_l(w_k, x_, y_)
        err[epoch] = np.linalg.norm(grads[epoch], 2)
    return w, f_val, grads, err



data = (x_train, y_train)
```

```python
sigma = 1e-3
d, N = x_train.shape
w0 = np.random.normal(0, sigma, (d, ))
w_sgd, f_val__sgd, grads_sgd, err_sgd = SGD(
    l, loss_grad, w0, data, batch_size, n_epochs)
```

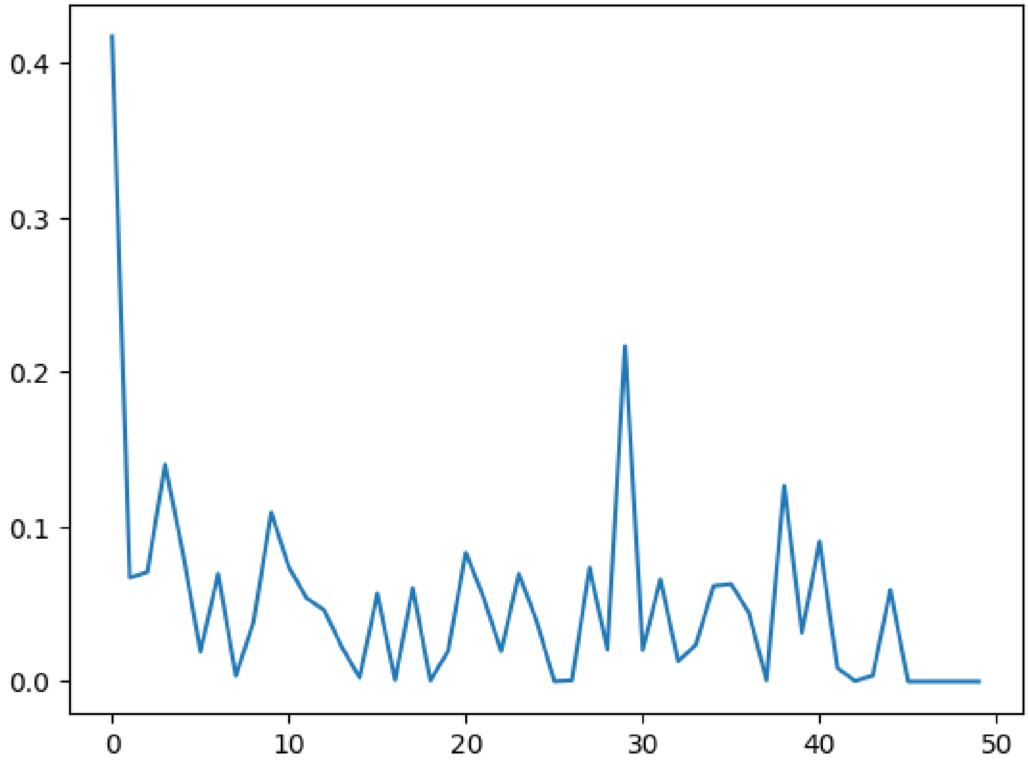
```
/tmp/ipykernel_11714/3753579620.py:2: RuntimeWarning: overflow encountered in
exp
  return 1 / (1 + np.exp(-x))
```

[87]:
```python
def predict(w, X, treshold=0.5):
    y_pred = f(X, w)
    y_copy = np.copy(y_pred)
    y_pred[y_copy < treshold] = 0
    y_pred[y_copy >= treshold] = 1
    return y_pred


def accuracy(y_hat, y):
    return round(np.mean(y_hat == y), 10)
```

[88]:
```python
w_star_sgd = w_sgd[-1]
y_hat = predict(w_star_sgd, x_train)
print("Accuracy: ", accuracy(y_hat, y_train))
x_plot = np.arange(n_epochs)
plt.plot(x_plot, err_sgd)
plt.show()
```

```
/tmp/ipykernel_11714/3753579620.py:2: RuntimeWarning: overflow encountered in
exp
  return 1 / (1 + np.exp(-x))
```

```
Accuracy:  0.9936956522
```



[89]:

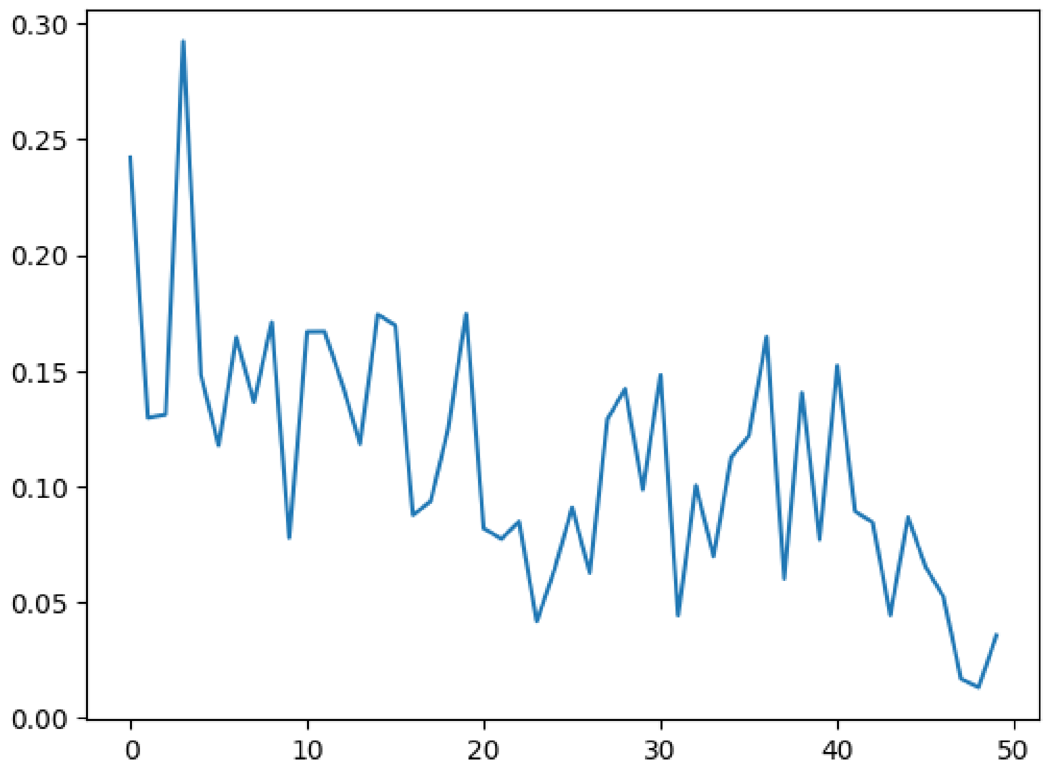
```
def GD2(f, grad_f, D, w0, tolf=1e-9, tolx=1e-9, kmax=50, alpha=0.1, back=False):

    x_cup, y = D
    shape = (kmax, *w0.shape)

    w = np.zeros(shape)
    f_val = np.zeros((kmax, 1))
    grads = np.zeros((kmax, 1, w0.shape[0]))
    err = np.zeros((kmax, 1))
    # output


    #x_cup = np.concatenate((np.ones((1, N)), X), axis=0)

    x_tol = tolx
    f_tol = tolf
    w_old = w0
    k = 0

    while k < kmax and x_tol >= tolx and f_tol >= tolf:
        if back:
            alpha = backtracking(f, grad_f, w_old)
```

```python
        w_k = w_old - alpha * grad_f(w_old, x_cup, y)
        x_tol = np.linalg.norm(w_k-w_old)
        f_tol = np.linalg.norm(f(w_k, x_cup, y))

        # Update arrays
        w[k] = w_k
        f_val[k] = f(w_k, x_cup, y)
        grads[k] = grad_f(w_k, x_cup, y)
        err[k] = np.linalg.norm(grads[k])
        w_old = w_k
        k = k+1

    return w[:k], f_val[:k], grads[:k], err[:k]
```

[97]:
```python
sigma = 1e-3
w0 = np.random.normal(0, sigma, (d, ))
w_gd, f_val_gd, grads_gd, err_gd = GD2(l, loss_grad, (x_train, y_train), w0)
```

```
/tmp/ipykernel_11714/3753579620.py:2: RuntimeWarning: overflow encountered in
exp
  return 1 / (1 + np.exp(-x))
```

[98]:
```python
w_star_gd = w_gd[-1]
x_plot = np.arange(len(w_gd))
plt.plot(x_plot, err_gd)
plt.show()
y_hat = predict(w_star_gd, x_train)
print("Accuracy: ", accuracy(y_hat, y_train))
```




```
Accuracy:  0.9531102733
```

```
/tmp/ipykernel_11714/3753579620.py:2: RuntimeWarning: overflow encountered in
exp
  return 1 / (1 + np.exp(-x))
```

[92]:
```
x_train, x_test, y_train, y_test = create_dataset(dataset, digits=[1, 7],␣
↪Ntrain=dataset.shape[0]//2)
```

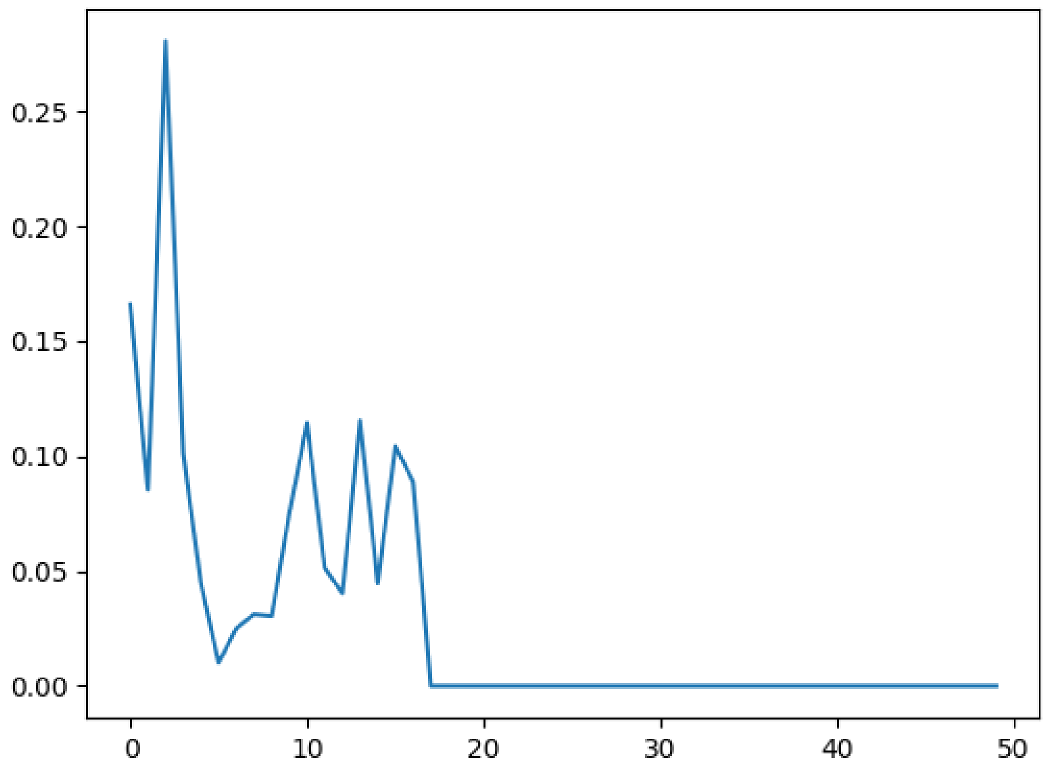
[103]:
```
data = (x_train, y_train)
sigma = 1e-3
d, N = x_train.shape
w0 = np.random.normal(0, sigma, (d, ))
w_sgd, f_val_sgd, grads_sgd, err_sgd = SGD(
    l, loss_grad, w0, data, batch_size, n_epochs)

sigma = 1e-3
w0 = np.random.normal(0, sigma, (d, ))
w_gd, f_val_gd, grads_gd, err_gd = GD2(l, loss_grad, (x_train, y_train), w0)
```

```
/tmp/ipykernel_11714/3753579620.py:2: RuntimeWarning: overflow encountered in
exp
  return 1 / (1 + np.exp(-x))
```

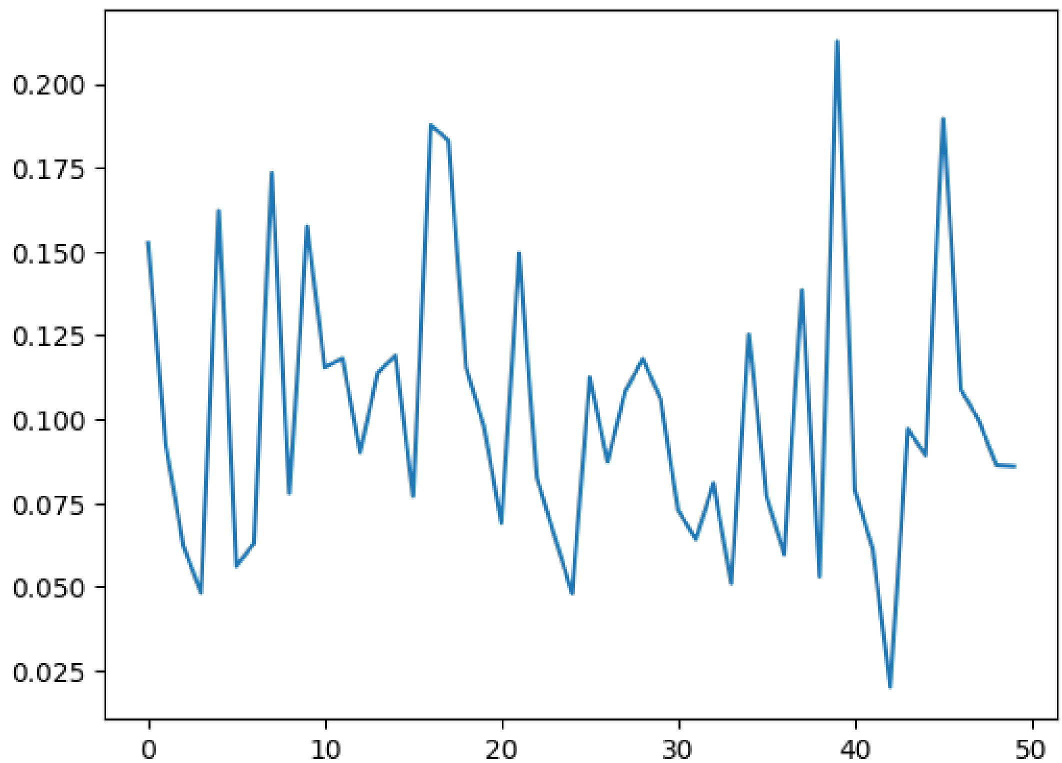
[104]:
```
w_star_sgd = w_sgd[-1]
y_hat = predict(w_star_sgd, x_train)
print("Accuracy: ", accuracy(y_hat, y_train))
x_plot = np.arange(n_epochs)
plt.plot(x_plot, err_sgd)
plt.show()
```

```
/tmp/ipykernel_11714/3753579620.py:2: RuntimeWarning: overflow encountered in
exp
  return 1 / (1 + np.exp(-x))
```

```
Accuracy:  0.9917530631
```



[105]:
```
w_star_gd = w_gd[-1]
x_plot = np.arange(len(w_gd))
plt.plot(x_plot, err_gd)
plt.show()
y_hat = predict(w_star_gd, x_train)
print("Accuracy: ", accuracy(y_hat, y_train))
```




```
Accuracy:  0.8225730443
```

```
/tmp/ipykernel_11714/3753579620.py:2: RuntimeWarning: overflow encountered in
exp
  return 1 / (1 + np.exp(-x))
```