Numerical Linear Regression

Steps

1. Gradient descent with one variable
2. Newton method with one variable
3. Gradient descent with two (and more) variables
4. Newton method with two (and more) variables

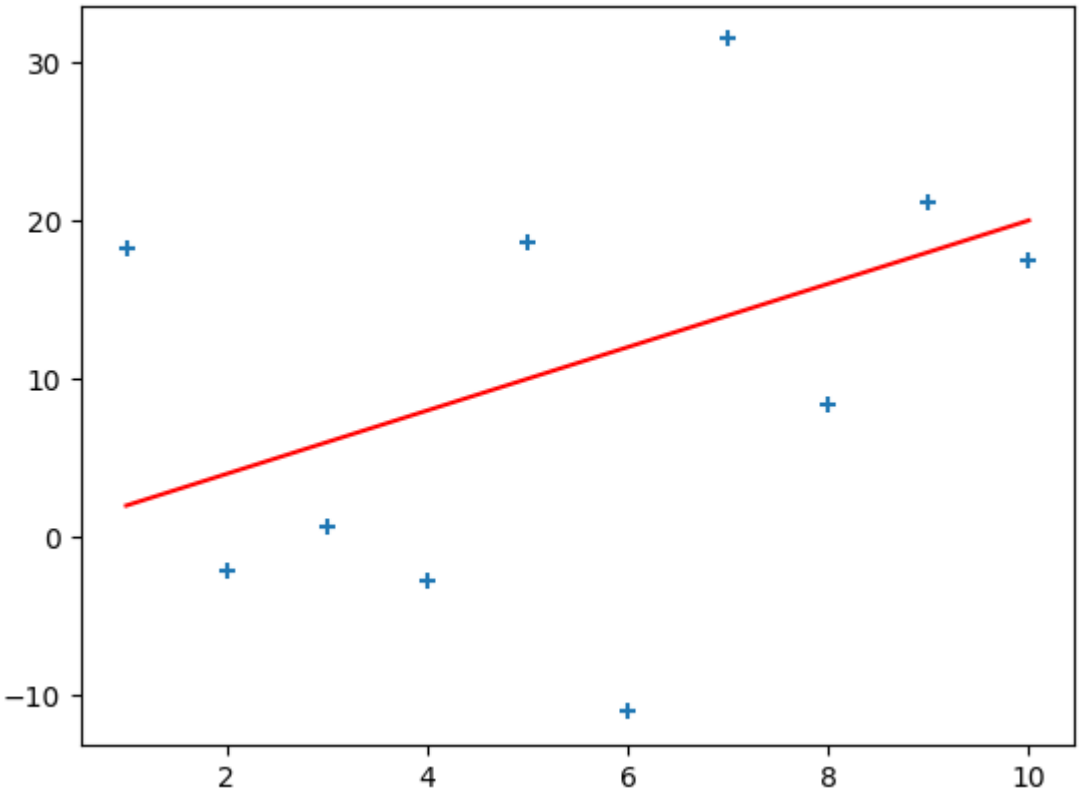
Gradient descent with one variable

Let us start a simple linear model $Y = aX$ with one feature X and one response Y, a slope of a that is to be learned assuming an intercept $b = 0$.

We generate data for $Y = 2X, X \in [1, 10]$ with a random error that is normally distributed proportional to $\mathcal{N}(0, 10)$.

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

np.random.seed(1) # for reproducibility
N = 10
a0 = 2
X = np.array(range(1,N+1))
Y = a0*X + np.random.normal(0.0,10.0, N)

plt.scatter(X,Y, marker='+')
plt.plot(X, a0*X, c='r')
plt.plot()
```

Out[ ]:
```
[]
```



We learn $\hat{a}$ (parameter, weight) from the data and expect it to be ≈ 2. Therefore, we minimize the residual sum of squares as our optimization goal (loss, error) function, i.e., $\hat{a} = \min \arg RSS(a)$ where $RSS(a) = \sum_{i=1}^{n}(y_i - ax_i)^2$.

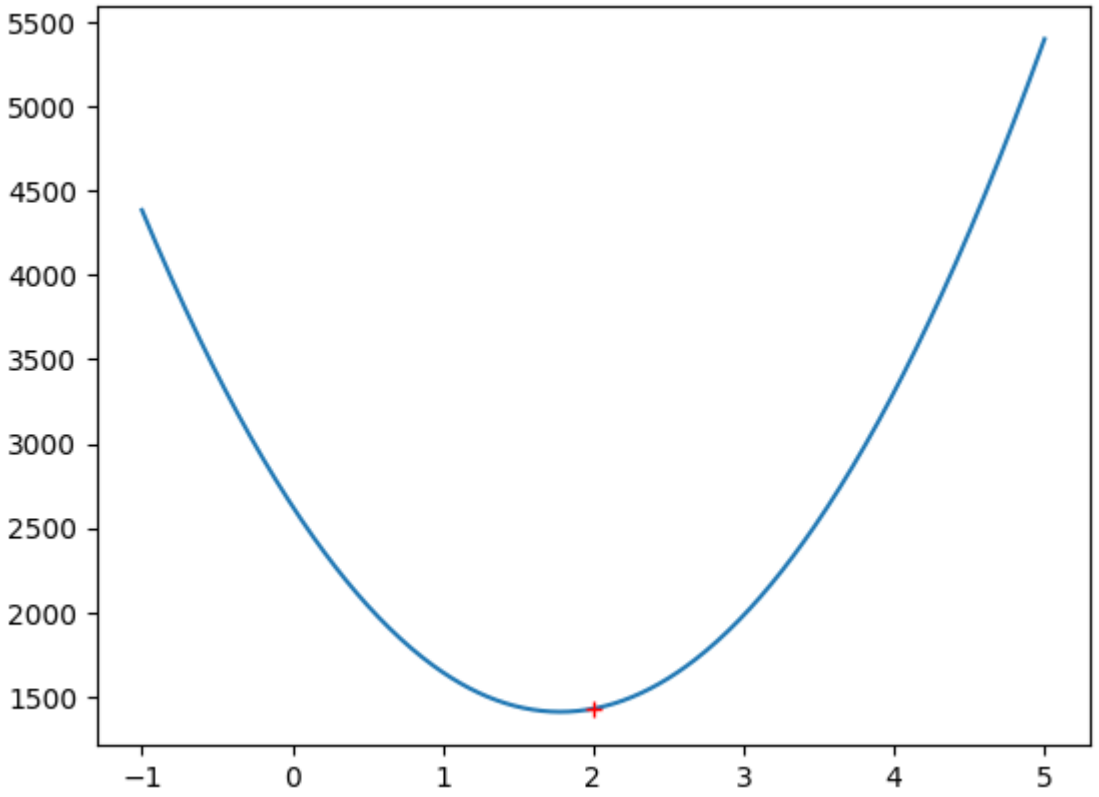
For the sake of demonstrating the numerical approach, we deliberately ignore that an analytic solution exists.

Let's define and plot RSS as a function of a. We chose the range of a around the (actually unknown) minimum of ≈ 2.

In [ ]:
```
def rss(a,X,Y):
    n = len(X)
    rss = 0
    for i in range(n):
        rss += (Y[i]-a*X[i])**2
    return rss
```

In [ ]:
```
def plot_rss(f,a_range):
    err_range = [f(e) for e in a_range]
    plt.plot(a_range, err_range)
```

In [ ]:
```
f = lambda a : rss(a,X,Y)
a_range = np.linspace(a0-3, a0+3, 500)
plot_rss(f, a_range)
rss_a0 = f(a0)
plt.plot(a0, rss_a0, marker='+', c='r')
plt.show()
print(f'rss = {rss_a0}')
```



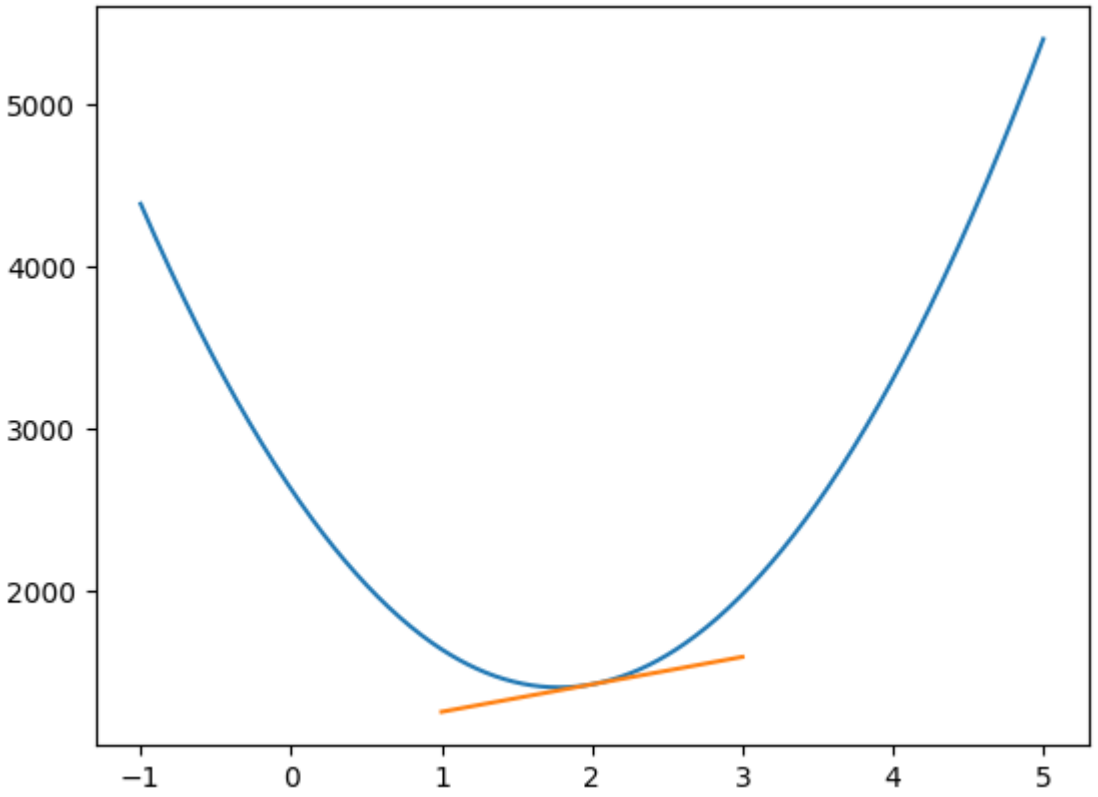

```
rss = 1427.6757139745007
```

The gradient of $RSS(a)$ for any a is defiend as:

$$\begin{aligned}\nabla RSS(a) &= \frac{\partial RSS(a)}{\partial a} \\ &= \frac{\partial \sum_{i=1}^{n}(y_i - ax_i)^2}{\partial a} \\ &= \sum_{i=1}^{n} 2(y_i - ax_i)(-x_i) \\ &= -2\sum_{i=1}^{n}(y_i - ax_i)x_i\end{aligned}$$

In [ ]:
```
def grad_rss(a,X,Y):
    n = len(X)
    grad_err = 0
    for i in range(n):
        grad_err += X[i]*(Y[i]-a*X[i])
    return -2 * grad_err
```

In [ ]:
```
ff = lambda a: grad_rss(a,X,Y)
fff = lambda a: (a-a0)*ff(a0)+f(a0)
a_range = np.linspace(a0-3, a0+3, 500)
plot_rss(f, a_range)
#plot_rss(ff, a_range)
a_range = np.linspace(a0-1, a0+1, 500)
plot_rss(fff, a_range)
```



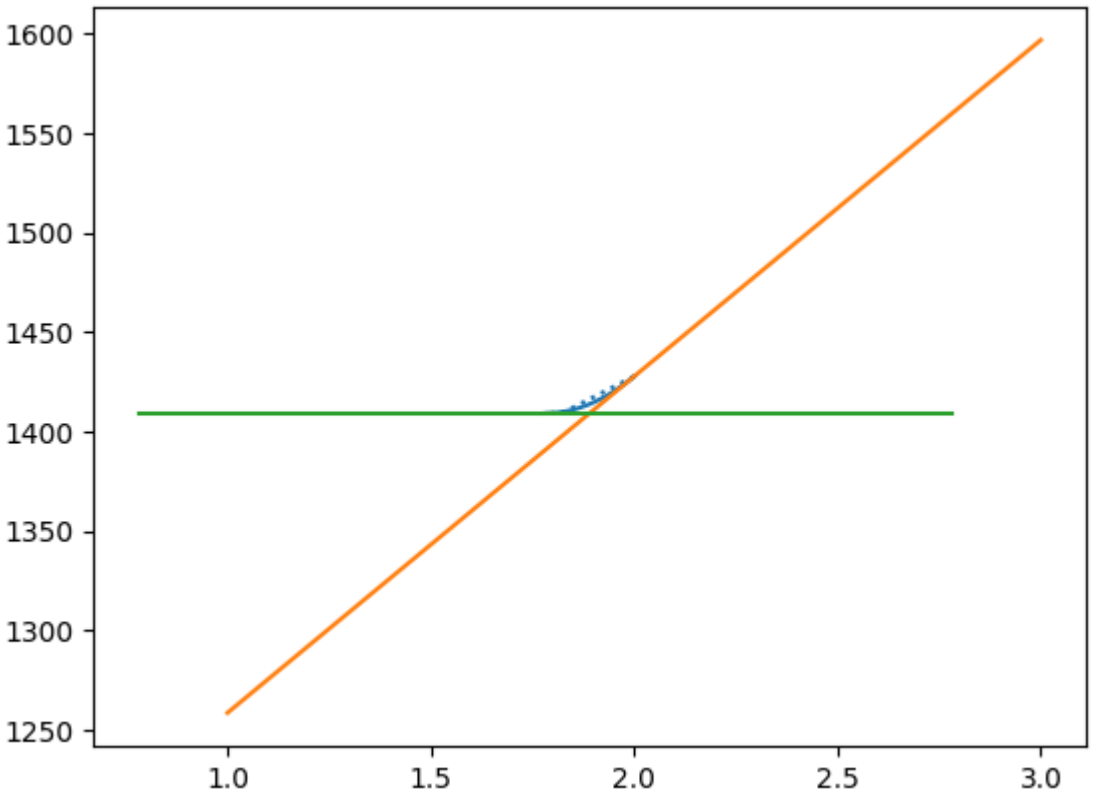
We iteratively approximate $\hat{a}$ with gradient descent, i.e., starting with an initial guess a_0 and a learning rate of ε we iterate over

$$\begin{aligned} a_{k+1} &= a_k - \varepsilon \nabla RSS(a_k) \\ &= a_k + 2\varepsilon \sum_{i=1}^{n} (y_i - a_k x_i) x_i \end{aligned}$$

In [ ]:
```
def grad_desc_rss(K, a0, learning_eps, f, ff, verbose):
    a_list = np.zeros((K+1))
    a_list[0] = a0
    for k in range(K):
        a_list[k+1] = a_list[k] - learning_eps * ff(a_list[k])
        if verbose:
            plt.plot(a_list[k:k+2], f(a_list[k:k+2]), linestyle=':')
            plt.plot(a_list[k+1], f(a_list[k+1]))

    if verbose:
        space = np.linspace(np.min(a_list), np.max(a_list), 1000)
        plot_rss(f,space)
        fff = lambda a: (a-a0)*ff(a0)+f(a0)
        space = np.linspace(a0-1, a0+1, 1000)
        plot_rss(fff,space)

        fff = lambda a: (a-a_list[k+1])*ff(a_list[k+1])+f(a_list[k+1])
        space = np.linspace(a_list[k+1]-1, a_list[k+1]+1, 1000)
        plot_rss(fff,space)
    return a_list
```

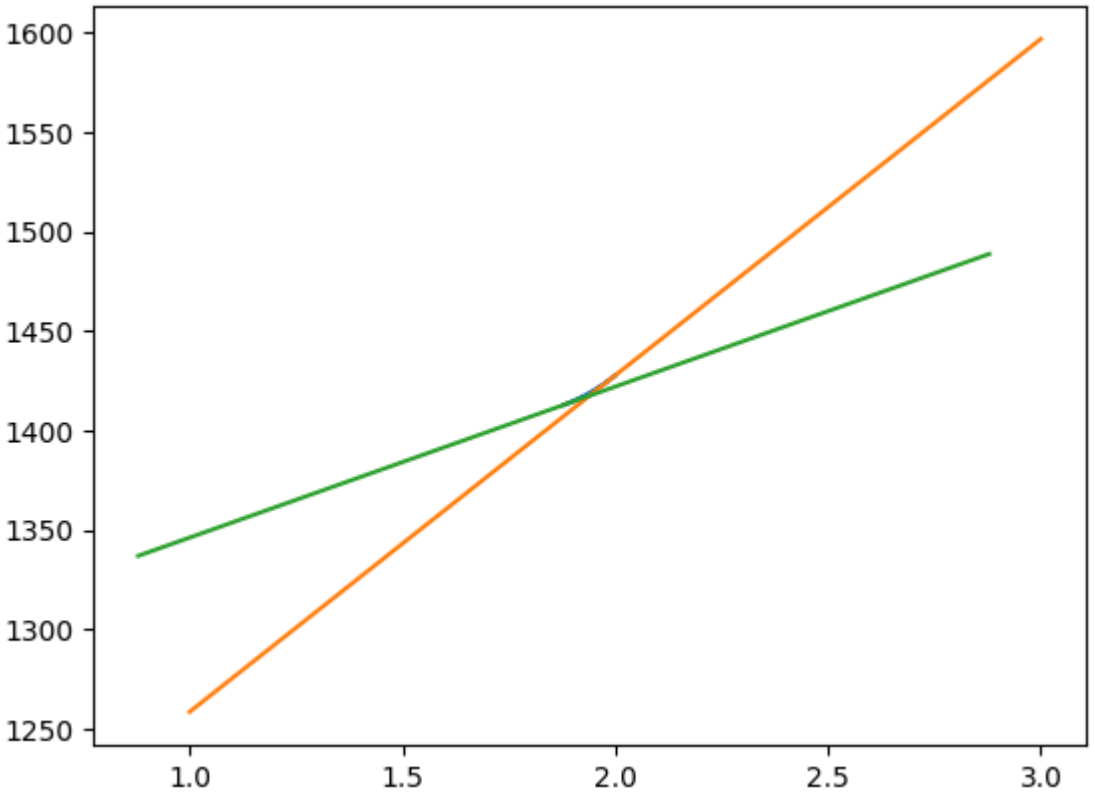
In [ ]:
```
K = 10
learning_eps = 0.001
a_list = grad_desc_rss(K, a0, learning_eps, f, ff, True)
print(f'as = {a_list}')
```

```
as = [2.         1.83087848 1.79198053 1.783034   1.7809763  1.78050303
 1.78039418 1.78036914 1.78036338 1.78036206 1.78036175]
```



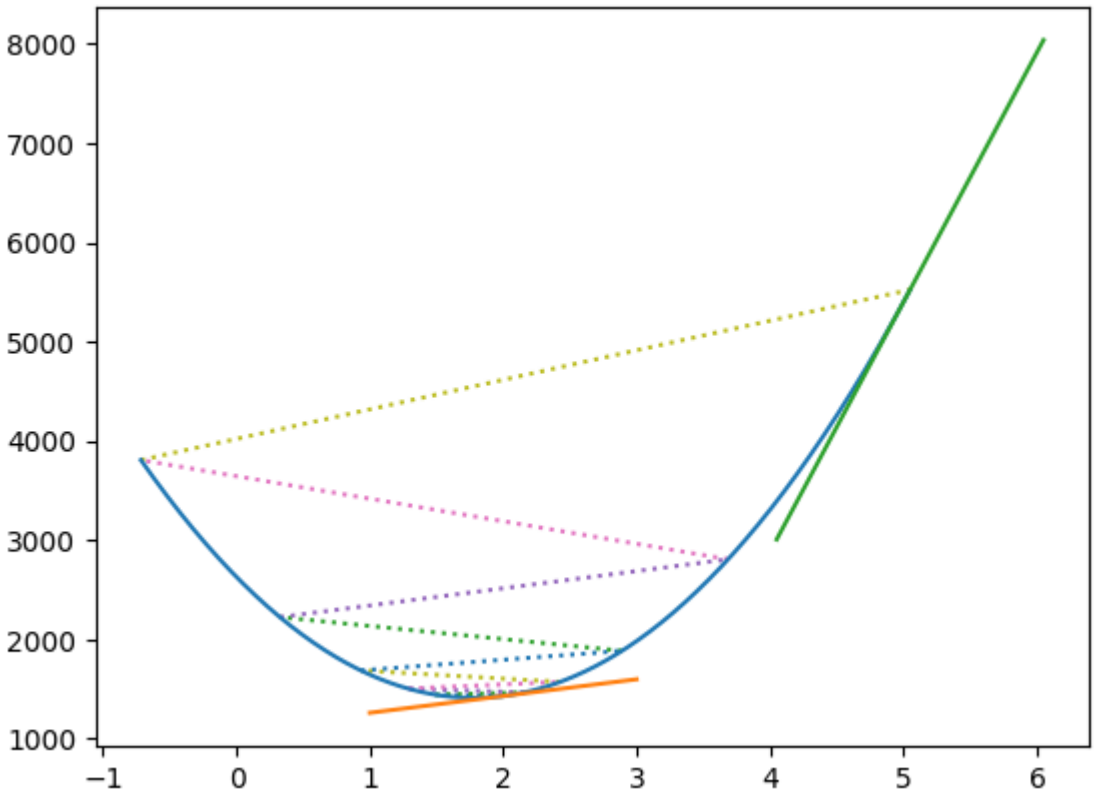
In [ ]:
```
learning_eps = 0.0001
print(f'as = {grad_desc_rss(K, a0, learning_eps, f, ff, True)}')
```

```
as = [2.         1.98308785 1.96747793 1.95306998 1.93977144 1.92749689
 1.91616747 1.90571043 1.89605857 1.88714991 1.87892721]
```



In [ ]:
```
learning_eps = 0.003
print(f'as = {grad_desc_rss(K, a0, learning_eps, f, ff, True)}')
```

```
as = [ 2.          1.49263544  2.15728301  1.28659469  2.42719639  0.93300
817
  2.89039474  0.32621833  3.68528943 -0.71509372  5.04940821]
```

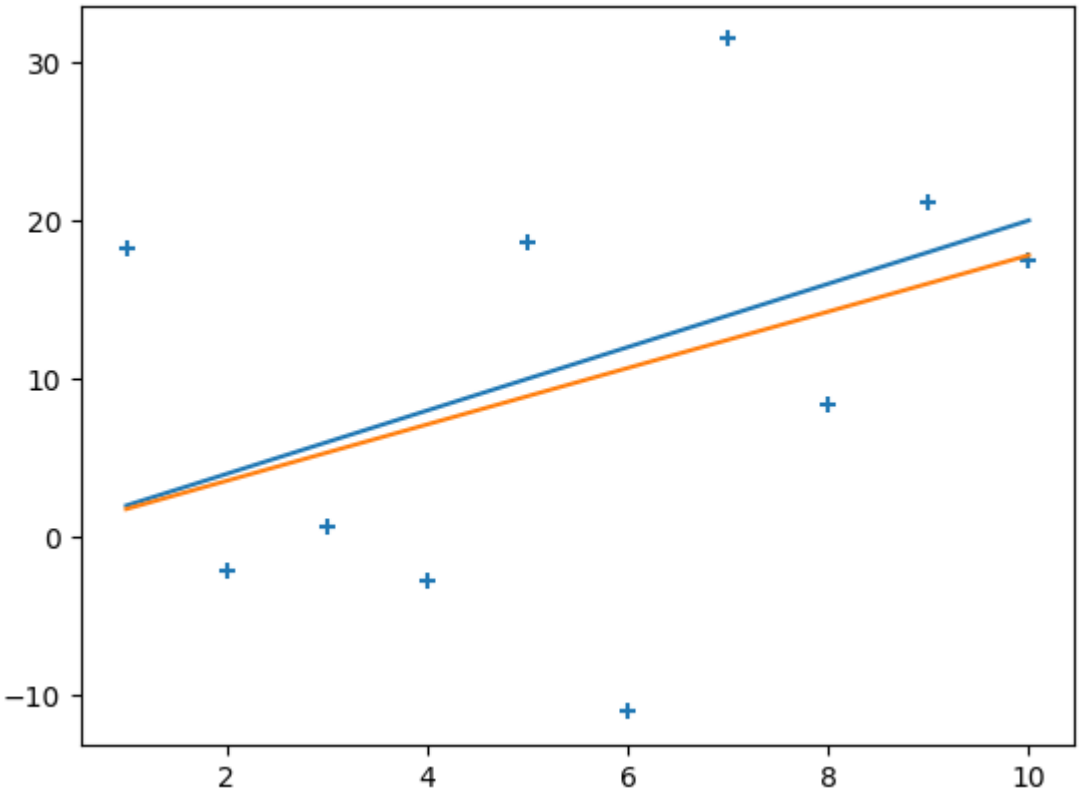


Try different learning rates and interpret the effect. Your answer goes here!

Answer: The learning rate of 0.001 produced the best result here and converges very fast within th considered 10 steps. The learning rate of 0.0001 converges to slow, i.e. the steps calulated by gradient decent are to small to reach the minimum within 10 steps. And the learining rate of 0.003 was to big and overshoots the minimum and, thus, considered bigger and bigger steps, since the gradient gots bigger and bigger. This will never converge.

Here the final result.

In [ ]:
```
plt.scatter(X,Y,marker='+')
plt.plot(X, a0*X)
plt.plot(X, a_list[-1]*X)
plt.show()
```



Newton method with one variable

It is obviously difficult to set the learning rate right. To help this, the Newton method adjusts the learning rate ε in each step to $1/h(a_i)$, i.e., the inverse of the second derivative $h(a)$ of the loss function at the current location a_i. In our case it is the inverse of the second derivative of RSS.

$$\begin{aligned} h(a) &= \frac{\partial^2 RSS(a)}{\partial a^2} \\ &= \frac{\partial(-2\sum_{i=1}^{n}(y_i - ax_i)x_i)}{\partial a} \\ &= 2\sum_{i=1}^{n} x_i^2 \end{aligned}$$

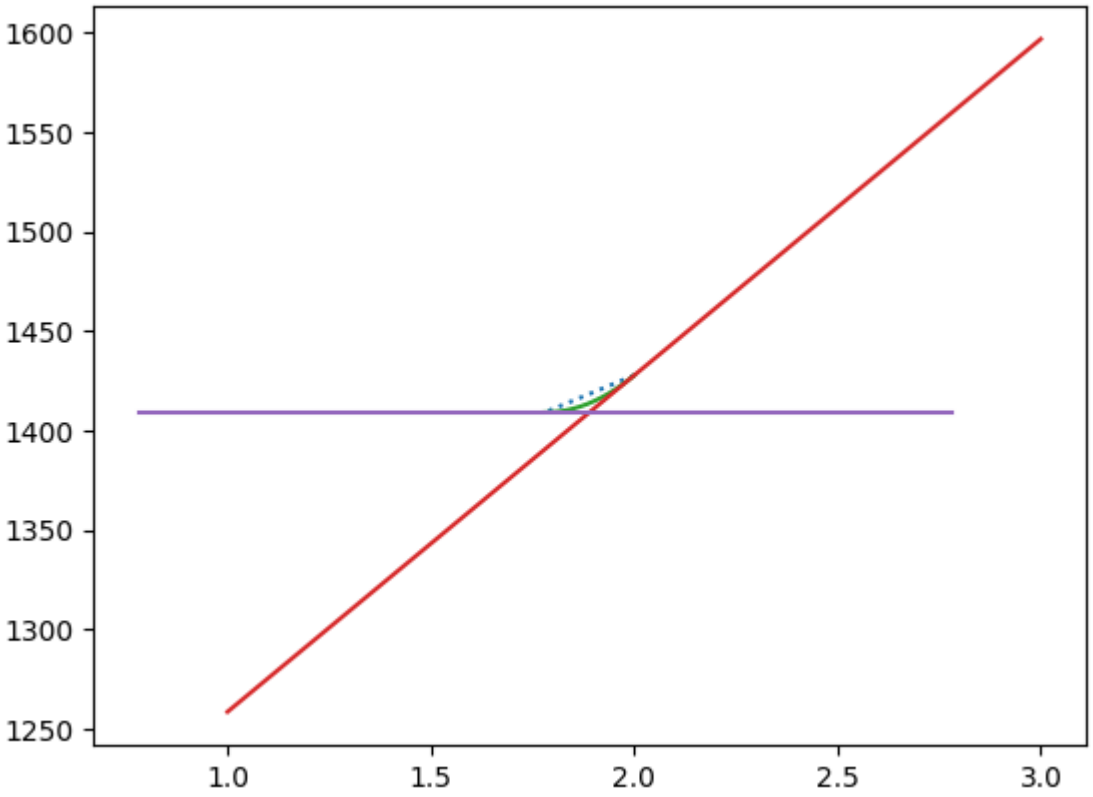
We observe, that h is a constant w.r.t. a and, hence, it does not change during iteration in our quadratic optimization case.

In [ ]:
```
h = 2 * sum(X**2)
learning_eps = 1/h
print(f'learning_eps = {learning_eps}' )
```

```
learning_eps = 0.0012987012987012987
```

In [ ]:
```
grad_desc_rss(1, a0, learning_eps, f, ff, True)
```

Out[ ]:
```
array([2.        , 1.78036166])
```

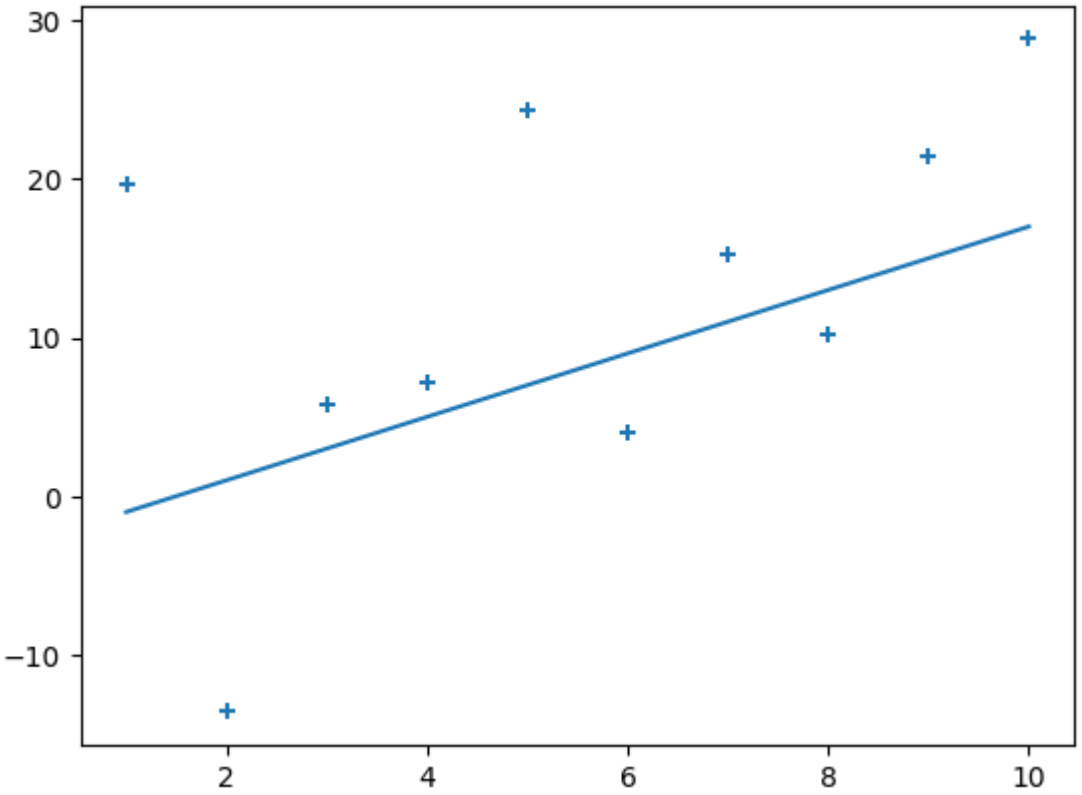


Gradient descent with two (and more) variables

Let us add an intercept to the linear model $Y = aX + b$ still with one feature X and one response Y, but with slope a and intercept b to be learned.

We generate data for $Y = 2X - 3$ with a random error that is normally distributed proportional to $\mathcal{N}(0, 10)$.

In [ ]:
```
b0 = -3
Y = a0*X-b0 + np.random.normal(0.0,10.0,N)
plt.scatter(X,Y, marker='+')
plt.plot(X, a0*X+b0)
plt.show()
```



We learn $\hat{a}$ and $\hat{b}$ from the data and expect them to be ≈ 2 and ≈ -3, respectively. Therefore, we minimize the residual sum of squares $RSS = \sum_{i=1}^{n}(y_i - \hat{a}x_i - \hat{b})^2$.

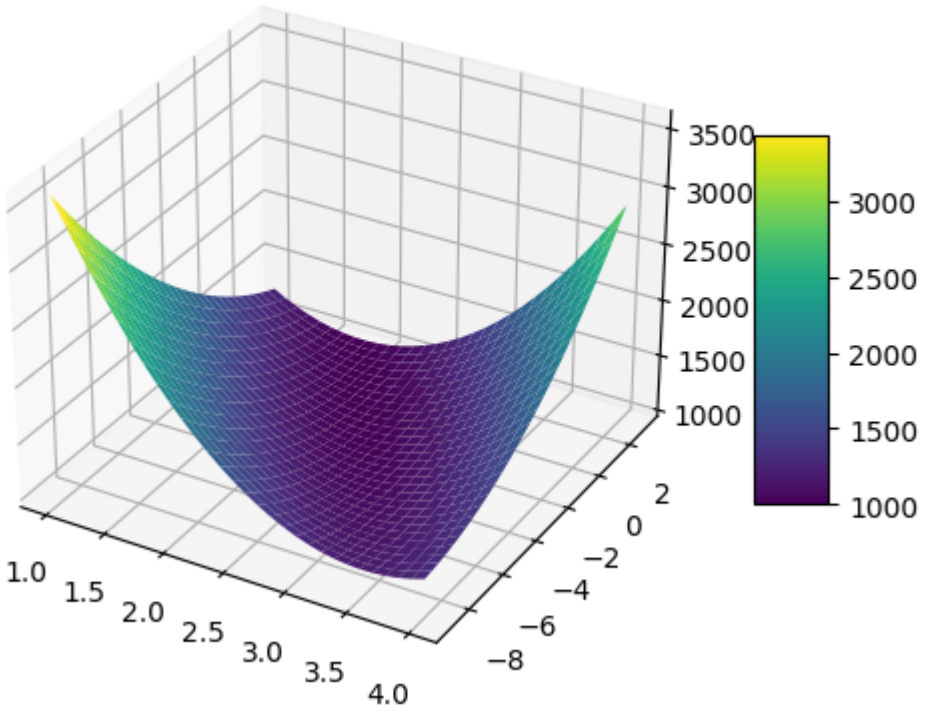
Let's define and 3D-plot the modified RSS as a function of a and b. We chose the ranges of a and b around the (actually unknown) expected minima of ≈ 2 and ≈ -3, respectively.

In [ ]:
```
def rss2(a,b,X,Y):
    n = len(X)
    rss = 0
    for i in range(n):
        rss += (Y[i]-a*X[i]-b)**2
    return rss
```

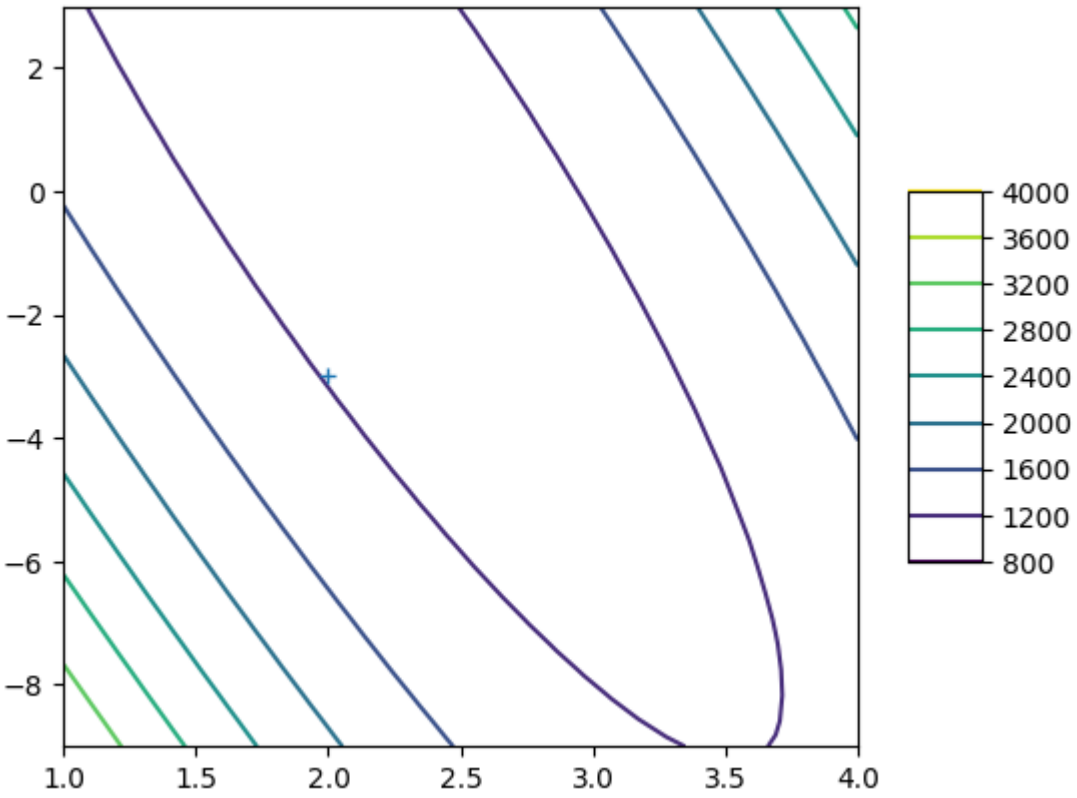
In [ ]:
```
def plot3d(f,A,B,real3d):
    Z = f(A,B)
    if real3d:
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        surf = ax.plot_surface(A, B, Z, cmap='viridis')
        fig.colorbar(surf, shrink=0.5, aspect=5)
    else:
        fig = plt.figure()
        ax = fig.add_subplot(111)
        surf = ax.contour(A, B, Z, cmap='viridis')
        fig.colorbar(surf, shrink=0.5, aspect=5)
    return fig, ax
```

In [ ]:
```
f = lambda a,b: rss2(a,b,X,Y)
```

```
a_sequence = np.linspace(a0-1, a0+2, 30)
b_sequence = np.linspace(b0-6, b0+6, 30)

A, B = np.meshgrid(a_sequence, b_sequence)

plot3d(f, A,B, True)
plt.show()
```



In [ ]:
```
plot3d(f,A,B,False)
plt.plot(a0, b0, marker='+')
plt.show()
```



What can you say about the loss function space? Your answer goes here.

Answer: The values of the loss function change very fast in diagonal direction (not parallel to x- and y-asix). Alonge the (approximated) diagonal axis, the function looks like a quardatic one, while it looks rather linear in the direction perdendicular to the digonal axis mentioned first. However, it is not linear but much flatter. It also seem to be a convex function. There are no saddle points at all.

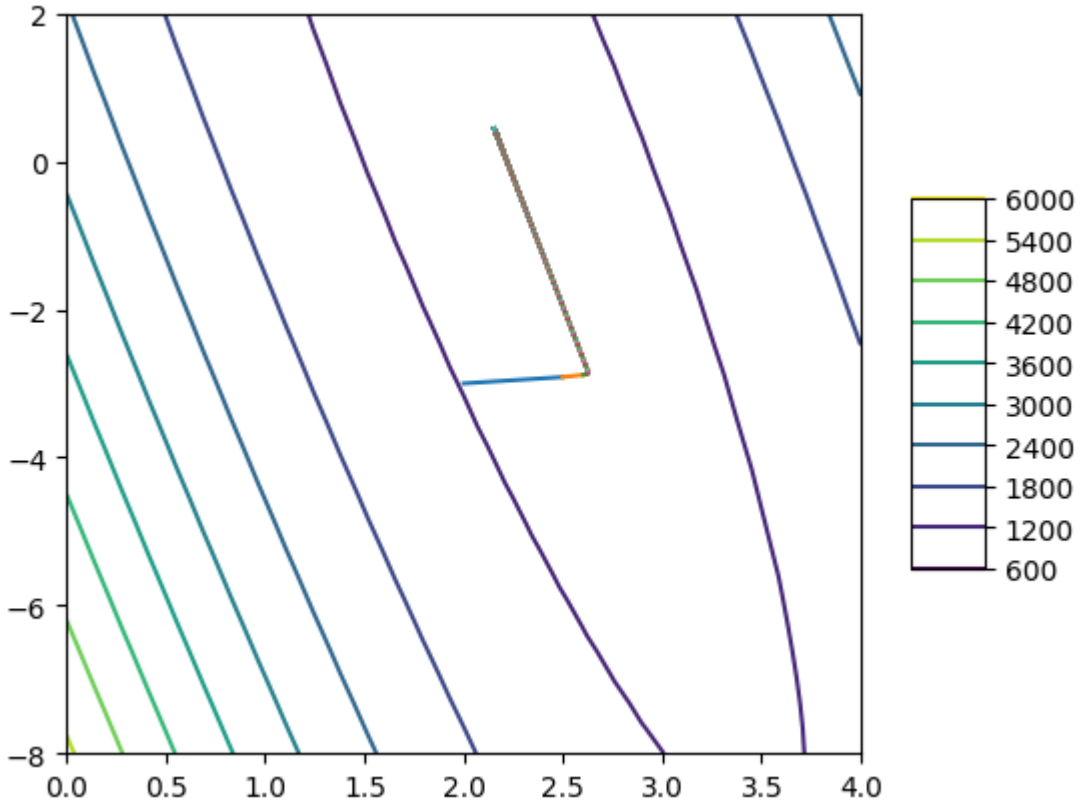
The gradient of $RSS(a, b)$ for any a, b is defiend as:

$$
\begin{aligned}
\nabla RSS(a,b) &= \left[\frac{\partial RSS(a,b)}{\partial a}, \frac{\partial RSS(a,b)}{\partial b}\right]^T \\
&= \left[\frac{\partial \sum_{i=1}^{n}(y_i - ax_i - b)^2}{\partial a}, \frac{\partial \sum_{i=1}^{n}(y_i - ax_i - b)^2}{\partial b}\right]^T \\
&= \left[\sum_{i=1}^{n} 2(y_i - ax_i - b)(-x_i), \sum_{i=1}^{n} 2(y_i - ax_i - b)(-1)\right]^T \\
&= \left[-2\sum_{i=1}^{n}(y_i - ax_i - b)x_i, -2\sum_{i=1}^{n}(y_i - ax_i - b)\right]^T \\
&= -2\left[\sum_{i=1}^{n}(y_i - ax_i - b)x_i, \sum_{i=1}^{n}(y_i - ax_i - b)\right]^T
\end{aligned}
$$

In [ ]:
```
def grad_rss2(a,b,X,Y):
    n = len(X)
    grad_a = 0
    grad_b = 0
    for i in range(n):
        temp = (Y[i]-a*X[i]-b)
        grad_a += temp*X[i]
        grad_b += temp

    return np.array([-2 * grad_a, -2 * grad_b])
```

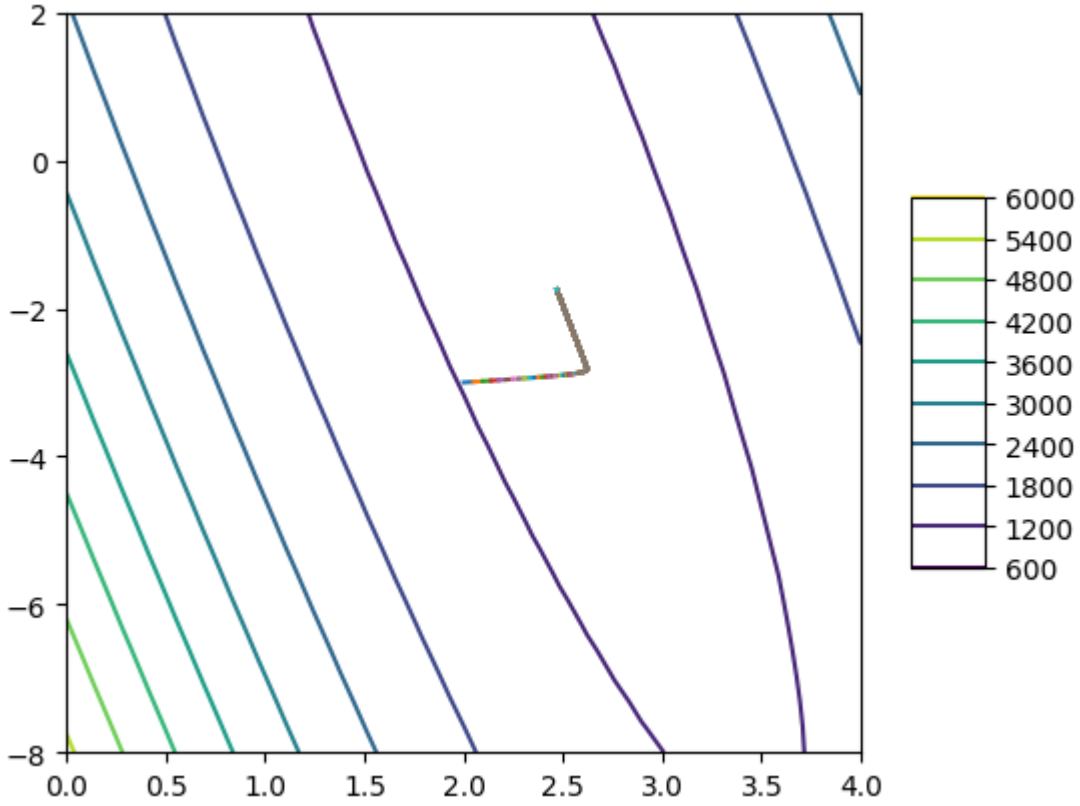
In [ ]:
```
def grad_desc_rss2(K, a0, b0, learning_eps, f, ff, verbose):
    a_list = np.zeros((K+1))
    b_list = np.zeros((K+1))
    a_list[0] = a0
    b_list[0] = b0
    for k in range(K):
        grad_w = ff(a_list[k], b_list[k])
        grad_a = grad_w[0]
        grad_b = grad_w[1]

        a_list[k+1] = a_list[k] - learning_eps * grad_a
        b_list[k+1] = b_list[k] - learning_eps * grad_b

    if verbose:
        alow = min(a0-2, np.min(a_list))
        ahigh = max(a0+2, np.max(a_list))
        blow = min(b0-5, np.min(b_list))
        bhigh = max(b0+5, np.max(b_list))
        a_sequence = np.linspace(alow, ahigh, 30)
        b_sequence = np.linspace(blow, bhigh, 30)

        A, B = np.meshgrid(a_sequence, b_sequence)
        fig, ax = plot3d(f, A, B, False)
        for k in range(K):
            ax.plot([a_list[k], a_list[k+1]], [b_list[k], b_list[k+1]])

    return a_list, b_list
```

In [ ]:
```
ff = lambda a,b: grad_rss2(a,b,X,Y)
K = 1000
learning_eps = 0.001
a_fitted, b_fitted = grad_desc_rss2(K, a0, b0, learning_eps, f, ff, True)
```



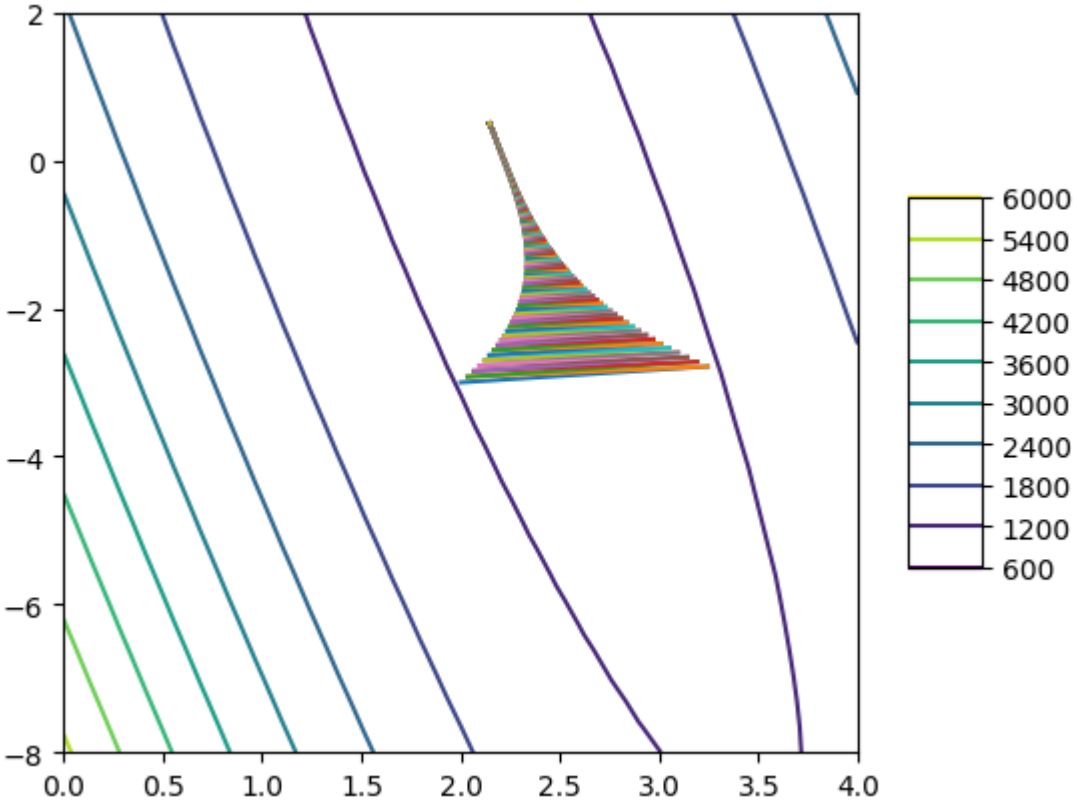
In [ ]:
```
learning_eps = 0.0001
grad_desc_rss2(K, a0, b0, learning_eps, f, ff, True)
```

Out[ ]:
```
(array([2.        , 2.04977214, 2.09561713, ..., 2.46858519, 2.46845017,
        2.46831521]),
 array([-3.        , -2.99139177, -2.98334825, ..., -1.74308243,
        -1.74214247, -1.74120291]))
```



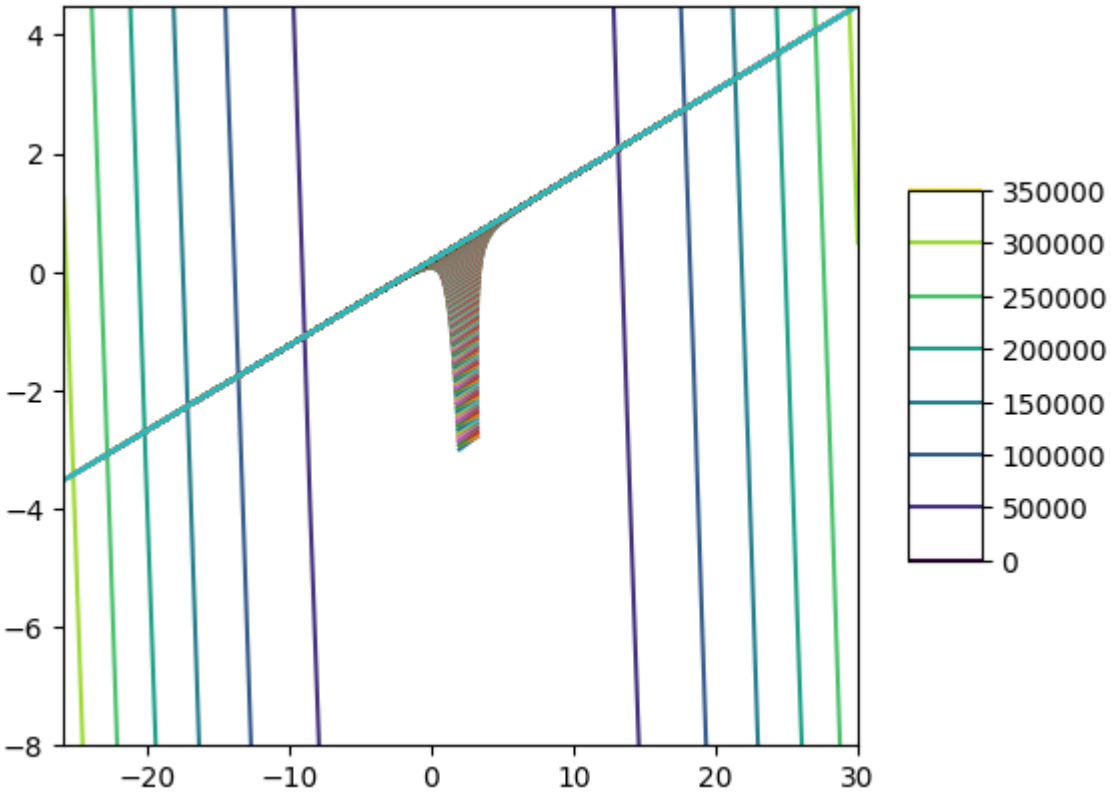
In [ ]:
```
learning_eps = 0.0025
grad_desc_rss2(K, a0, b0, learning_eps, f, ff, True)
```

Out[ ]:
```
(array([2.        , 3.24430341, 2.03414116, ..., 2.14709535, 2.14709521,
        2.14709508]),
 array([-3.        , -2.78479419, -2.92253211, ...,  0.49507273,
         0.49507368,  0.49507463]))
```



In [ ]:
```
learning_eps = 0.00255
grad_desc_rss2(K, a0, b0, learning_eps, f, ff, True)
```

Out[ ]:
```
(array([  2.        ,   3.26918948,   1.98475288, ..., -25.62113426,
         30.02059407, -25.83208199]),
 array([-3.        , -2.78049007, -2.9281828 , ..., -3.4935534 ,
         4.49885591, -3.52385245]))
```

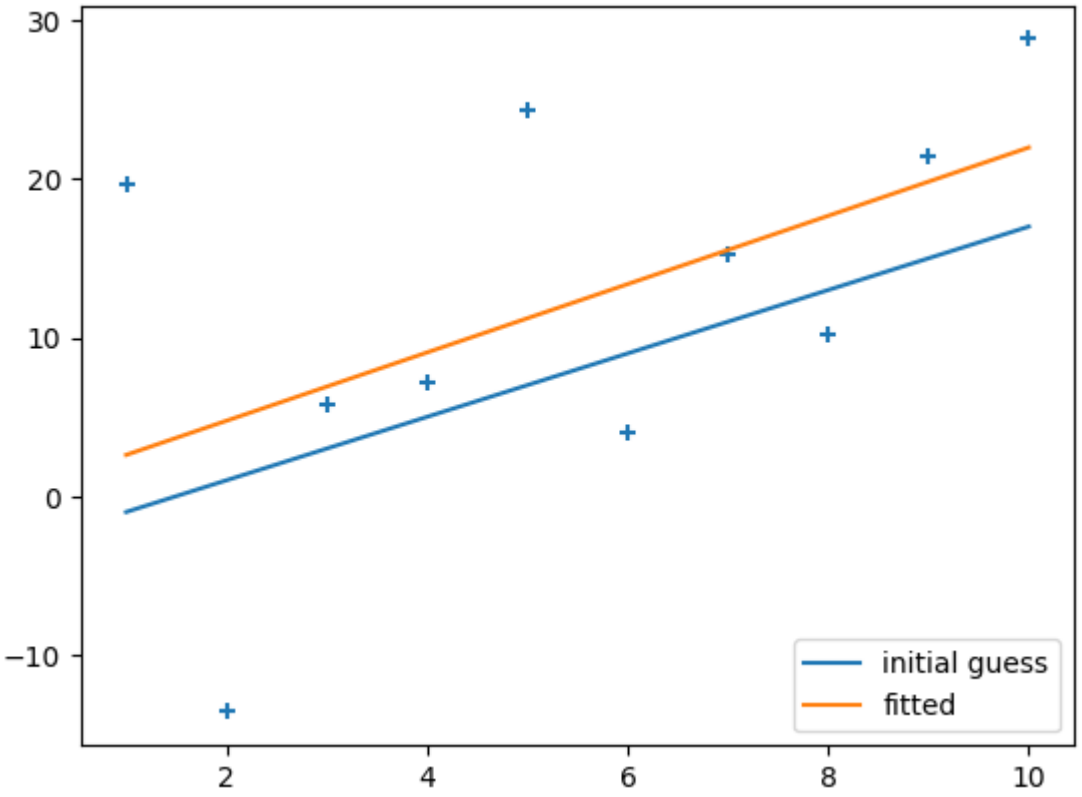


Try different learning rates and interpret the effect. Your answer goes here!

Answer: The learning rate of 0.001 produced the best result here and converges with quite big steps at the beginning. The learning rate of 0.0001 converges much slower, since the fist steps are much smaller, due to the smaller learning rate. The learning rate of 0.0025 is likely to converge to the minimum. However, it goes less directly, since it overshoots the point in one of the directions in each step. So it converges slower that with the former learning rates. And the learining rate of 0.0055 show a quite similar picture like with 0.0025, but it overshoots the minimum in both directions and, thus, considered bigger and bigger steps, since the gradient gots bigger and bigger. This will never converge.

Here the final result.

In [ ]:
```
plt.scatter(X,Y, marker='+')
plt.plot(X, a0*X+b0, label='initial guess')
plt.plot(X, a_fitted[-1]*X+b_fitted[-1], label='fitted')
plt.legend(loc='lower right')
plt.show()
```



Newton method with two (and more) variables

It is still difficult to set the learning rate right. To help this, the Newton method for one variable generalizes:

$$[a_{k+1}, b_{k+1}] = [a_k, b_k] - H_{RSS}(a_k, b_k)^{-1} \nabla RSS(a_k, b_k)$$

where H_{RSS} is the Hessian matrix of RSS.

The Hessian matrix is defined as:

$$\begin{aligned} H_{RSS}(a,b) &= \begin{bmatrix} \frac{\partial^2 RSS(a,b)}{\partial a^2} & \frac{\partial^2 RSS(a,b)}{\partial a \partial b} \\ \frac{\partial^2 RSS(a,b)}{\partial b \partial a} & \frac{\partial^2 RSS(a,b)}{\partial b^2} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial(-2\sum_{i=1}^n (y_i - a x_i - b) x_i)}{\partial a} & \frac{\partial(-2\sum_{i=1}^n (y_i - a x_i - b) x_i)}{\partial b} \\ \frac{\partial(-2\sum_{i=1}^n (y_i - a x_i - b))}{\partial a} & \frac{\partial(-2\sum_{i=1}^n (y_i - a x_i - b))}{\partial b} \end{bmatrix} \\ &= \begin{bmatrix} 2\sum_{i=1}^n x_i^2 & 2\sum_{i=1}^n x_i \\ 2\sum_{i=1}^n x_i & 2n \end{bmatrix} \\ &= 2 \begin{bmatrix} \sum_{i=1}^n x_i^2 & \sum_{i=1}^n x_i \\ \sum_{i=1}^n x_i & n \end{bmatrix} \end{aligned}$$

As in the case for one variable, we observe that H_{RSS} is a constant matrix, i.e., it does not change during the iterations in our quadratic optimization case.

In [ ]:
```
H = np.zeros((2,2))
H[0,0] = 2*sum(X**2)
```

```
H[0,1] = 2*sum(X)
H[1,0] = H[0,1]
H[1,1] = 2*len(X)
H_inv = np.linalg.inv(H)
print(f'H = {H}')
print(f'H_inv = {H_inv}')
```

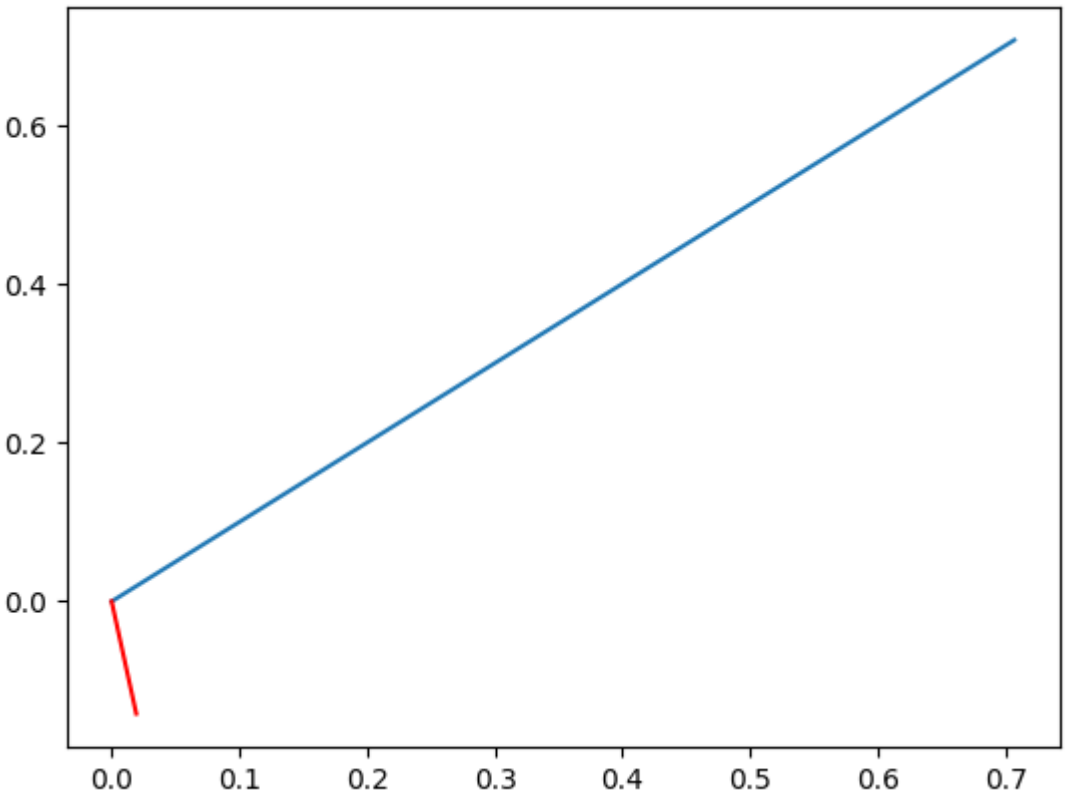
```
H = [[770. 110.]
 [110.  20.]]
H_inv = [[ 0.00606061 -0.03333333]
 [-0.03333333  0.23333333]]
```

Recall the inverse of a 2×2 matrix H:

$$H^{-1} = \frac{1}{h_{11}h_{22} - h_{12}h_{21}} \begin{bmatrix} h_{22} & -h_{12} \\ -h_{21} & h_{11} \end{bmatrix}$$

A learning rate ε only scales the gradient vector; a Hessian matrix scales and rotates the gradient vector. Check how this concrete Hessian matrix transforms (scales and rotates) a unit vector:

In [ ]:
```
import math

v = np.array([1/math.sqrt(2), 1/math.sqrt(2)])
plt.plot([0,v[0]], [0,v[1]])
v2 = np.dot(-H_inv, np.transpose(v))
plt.plot([0,v2[0]], [0,v2[1]], c='r')
plt.show()
```

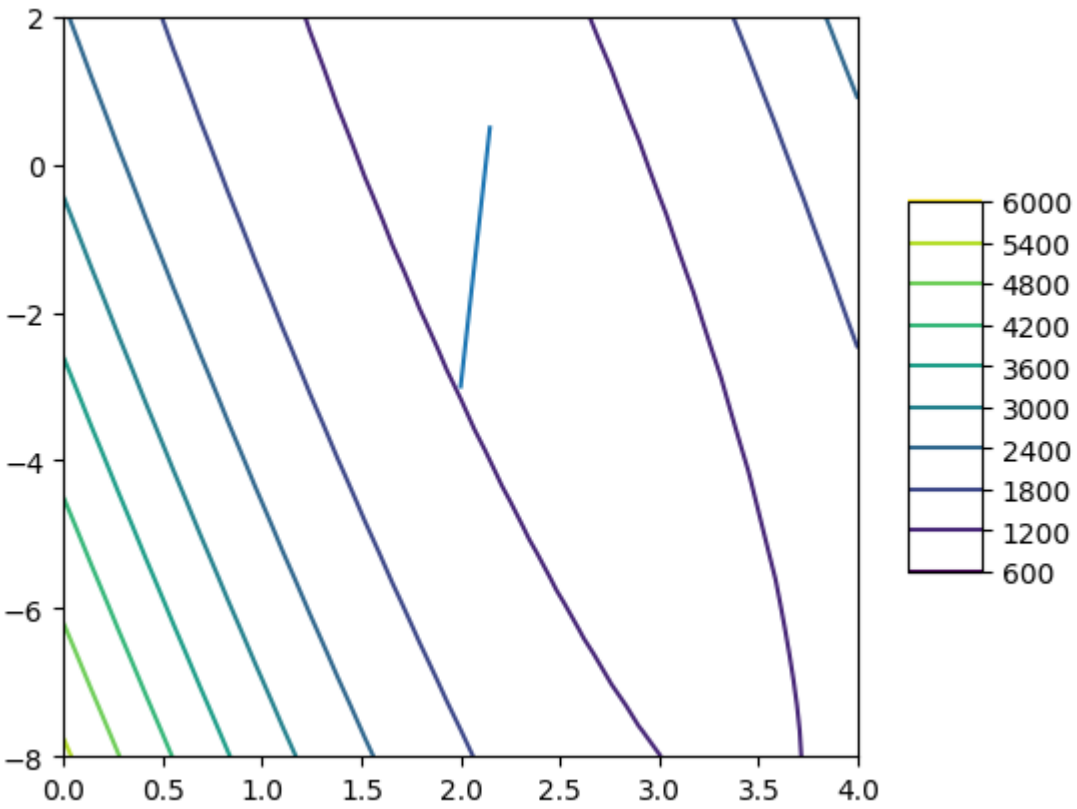


In [ ]:
```
def newton_rss2(K, a0, b0, H, f, ff, verbose):
    H_inv = np.linalg.inv(H)
    ab = np.zeros((2,K+1))
    ab[0,0] = a0
```

```
    ab[1,0] = b0
    for k in range(K):
        grad_w = ff(ab[0,k], ab[1,k])
        grad_a = grad_w[0]
        grad_b = grad_w[1]
        ab[:,k+1] = ab[:,k] - np.dot(H_inv, np.transpose(np.array([grad_a

    if verbose:
        alow = min(a0-2, np.min(ab[0,:]))
        ahigh = max(a0+2, np.max(ab[0,:]))
        blow = min(b0-5, np.min(ab[1,:]))
        bhigh = max(b0+5, np.max(ab[1,:]))
        a_sequence = np.linspace(alow, ahigh, 30)
        b_sequence = np.linspace(blow, bhigh, 30)

        A, B = np.meshgrid(a_sequence, b_sequence)
        fig, ax = plot3d(f, A, B, False)
        for k in range(K):
            ax.plot([ab[0,k], ab[0,k+1]], [ab[1,k], ab[1,k+1]])

    return ab
```

In [ ]:
```
K = 2
ab_newton = newton_rss2(K, a0, b0, H, f, ff, True)
print(f'ab_newton = {ab_newton}')
```

```
ab_newton = [[ 2.          2.14708232  2.14708232]
 [-3.          0.4951634   0.4951634 ]]
```



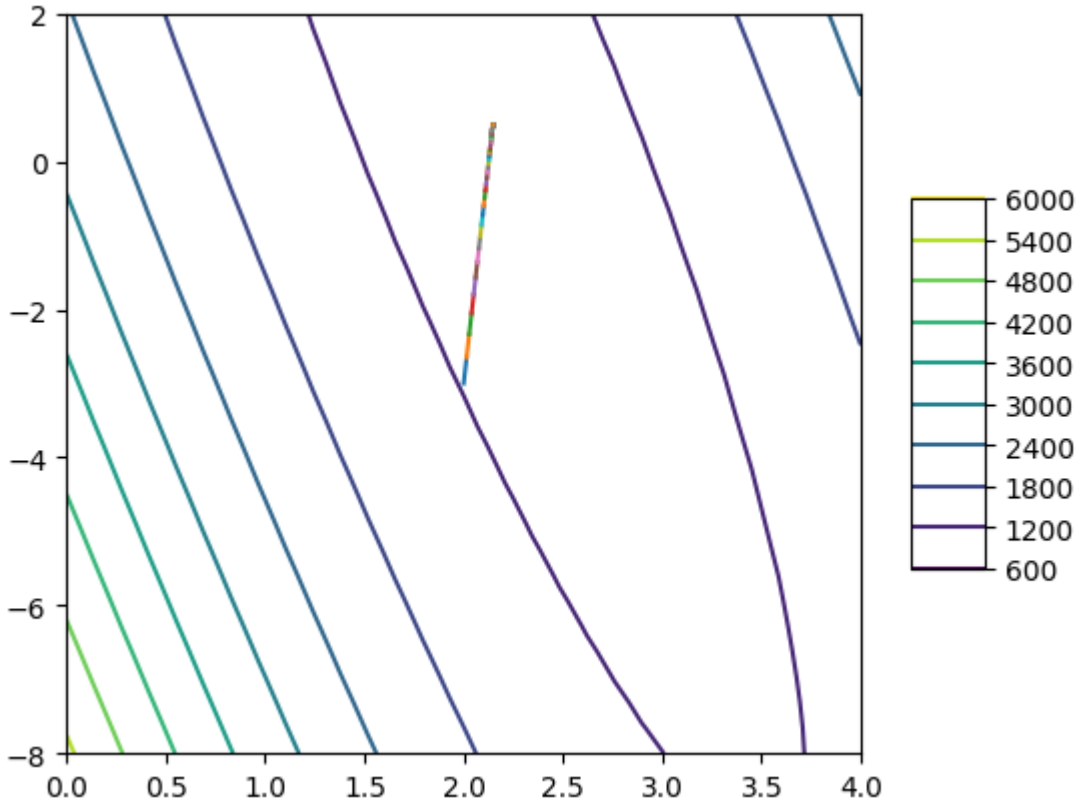
In [ ]:
```
rss_0 = f(a0, b0)
rss_opt = f(ab_newton[0,-1], ab_newton[1,-1])

print(f'rss_0 = {rss_0}')
print(f'rss_opt = {rss_opt}')
```

```
rss_0 = 1183.5345788610894
rss_opt = 996.4956782237203
```

The Newton method could overshoot the minimum (if the loss function is not convex or due to numerical errors). Therefore, the relaxed or damped Newton's method introduces a damping factor $0 < \gamma \leq 1$. Also, we exit when we overshoot the optimum.

In [ ]:
```
def damped_newton_rss2(K, a0, b0, H, gamma, f, ff, verbose):
    H_inv = gamma*np.linalg.inv(H)
    ab = np.zeros((2, K+1))
    ab[0,0] = a0
    ab[1,0] = b0
    for k in range(K):
        grad_w = ff(ab[0,k], ab[1,k])
        grad_a = grad_w[0]
        grad_b = grad_w[1]
        ab[:,k+1] = ab[:,k] - np.dot(H_inv, np.transpose(np.array([grad_a

    if verbose:
        alow = min(a0-2, np.min(ab[0,:]))
        ahigh = max(a0+2, np.max(ab[0,:]))
        blow = min(b0-5, np.min(ab[1,:]))
        bhigh = max(b0+5, np.max(ab[1,:]))
        a_sequence = np.linspace(alow, ahigh, 30)
        b_sequence = np.linspace(blow, bhigh, 30)

        A, B = np.meshgrid(a_sequence, b_sequence)
        fig, ax = plot3d(f, A, B, False)
        for k in range(K):
            ax.plot([ab[0,k], ab[0,k+1]], [ab[1,k], ab[1,k+1]])
    return ab
```

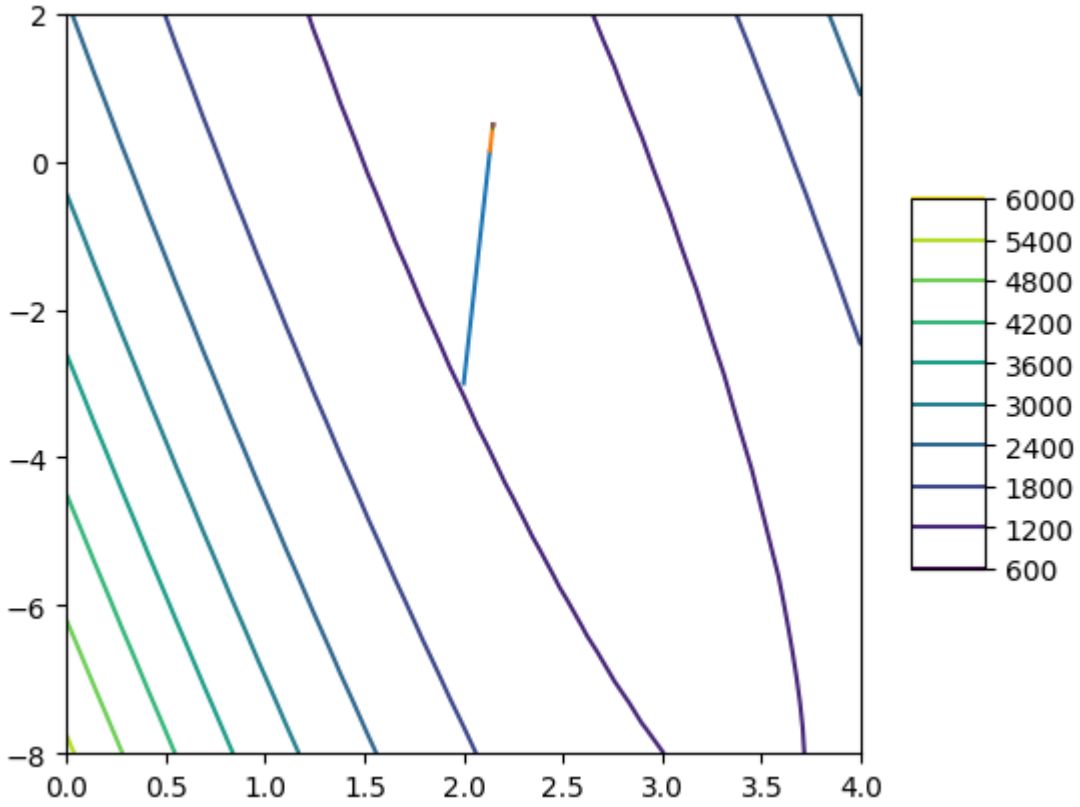
In [ ]:
```
K = 1000
gamma = 0.1
ab_newton = damped_newton_rss2(K, a0, b0, H, gamma, f, ff, True)
```



In [ ]:
```
rss_0 = f(a0,b0)
rss_opt = f(ab_newton[0,-1], ab_newton[1,-1])

print(f'rss_0 = {rss_0}')
print(f'rss_opt = {rss_opt}')
```

```
rss_0 = 1183.5345788610894
rss_opt = 996.4956782237206
```

In [ ]:
```
gamma = 0.9
ab_newton = damped_newton_rss2(K, a0, b0, H, gamma, f, ff, True)
```



In [ ]:
```
rss_opt = f(ab_newton[0,-1], ab_newton[1,-1])

print(f'rss_0 = {rss_0}')
print(f'rss_opt = {rss_opt}')
```

```
rss_0 = 1183.5345788610894
rss_opt = 996.4956782237202
```

Try different damping factors and interpret the effect. Your answer goes here!

Answer: The dumping factor gamma helped to prevent overshooting the minimum using newton method. The differences between gamma = 0.1 and gamma = 0.9 is that with 0.9 the method converges much faster than with 0.1, since the considered steps are larger. However, we have to be aware of the fact that, the larger the steps are, the higher the risk for overshooting, e.g., we had in the original case where technically gamma = 1.0. In the result, the optimized losses are quite similar.

Here the final result.

In [ ]:
```
print(f'{ab_newton[0,1], ab_newton[1,-1]}')
```

```
(2.132374091583392, 0.49516339694790856)
```