# Variants of stochastic gradient-based optimization

May 4, 2021

## 1 Variants of stochastic gradient-based optimization

### 1.1 Steps

1. Setting the scene
2. Stochastic Gradient Descent (SGD)
3. SGD with adaptive learning rate
4. SGD with momentum
5. SGD with accumulated squared gradient: AdaGrad
6. SGD with accumulated squared gradient: RMSProp
7. SGD with accumulated squared gradient: ADAM
8. SGD with Newton's method
9. SGD with the conjugate gradient method

### 1.2 Setting the scene

During learning, we optimize the mean squared error $MSE$ of the models $m$ for the model parameters $\mathbf{w}$:

$$MSE(\mathbf{w}, m, X, Y) = \frac{1}{N}\sum_{i=1}^{N}(y_i - m(\mathbf{w}, \mathbf{x}_i))^2$$

In other words, we find $\arg\min_{\mathbf{w}} MSE(\mathbf{w}, m, X, Y)$.

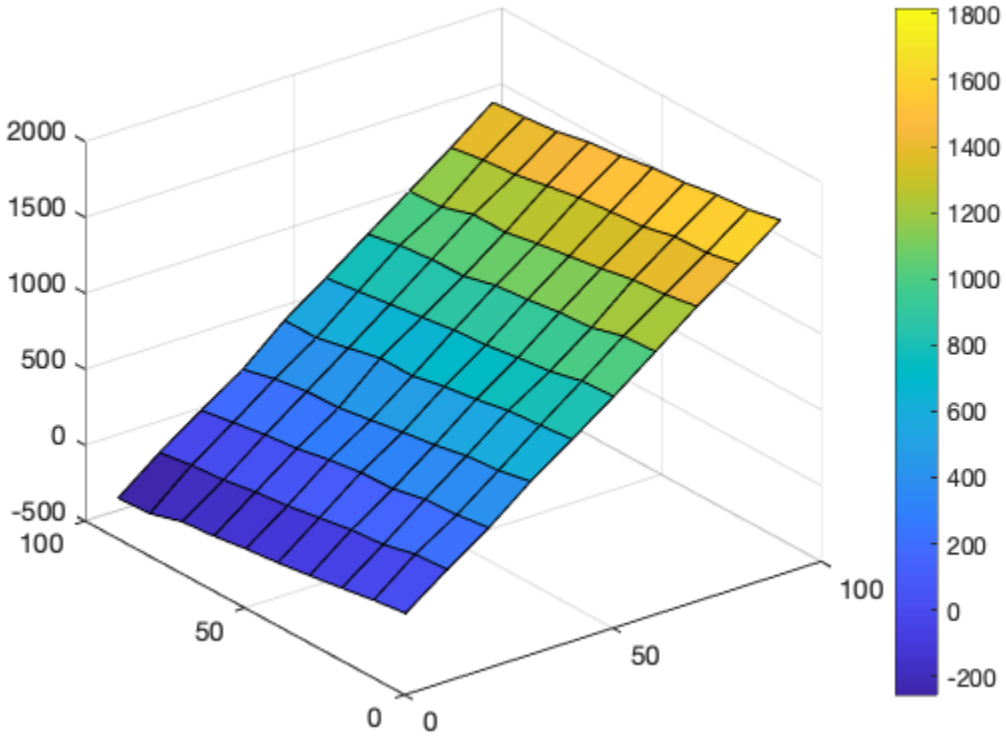
We will reuse some functions defined in the notebook "Learn an XOR Neural Network using gradient-based optimization" including: 1. The mean squared error function `mse` (loss function). 2. The gradient of the mean squared error function `grad_mse` (gradient of the loss function). 3. The gradient descent function for the loss function `grad_desc_mse`. 4. The 3D surcace and contour plot function `plot3D`.

First we generate sample data points. The generator function is $y = 20x_1 - 3x_2$. We sample data at integer points $x_{i,1} \times x_{i,2} \in [1 \dots N] \times [1 \dots N], N = 100$ and add a random error to $y_i$ that is normally distributed proportional to $\mathcal{N}(0, 10)$.

Below we plot the function.

[1]:
```
rng(1);
N=100; %N=1000;
XX=fliplr(fullfact([N N]));
a10 = 20;
```

```
a20 = -3;
X1 = XX(:,1);
X2 = XX(:,2);

f0 = @(x1,x2)(a10*x1 + a20*x2);
f = @(x1,x2,r)(a10*x1 + a20*x2 + r);
R = normrnd(0,10,1,N*N).';
Y = arrayfun(f,X1,X2,R);
[A,B] = meshgrid(1:N/10:N,1:N/10:N);
ff =@(x1,x2)(f(x1,x2,normrnd(0,10)));
plot3d(ff,A,B, true);
```




The model that we use throughout this notebook is a simple linear Neural Network model. It consists of one neuron connected to the input $\mathbf{x}$ and an identity, i.e., no effective, activation function. The neuron and the whole model $m_1$ implements $m_1(\mathbf{w}, \mathbf{x}) = \mathbf{w}^T\mathbf{x}$.

[2]:
```
m1 = @(ws,x)(ws(1)*x(1) + ws(2)*x(2));
mse1 = @(ws)(mse(ws,m1,XX,Y));
```

As the Tensorflow default, we implement the Glorot uniform initializer for setting the initial weights $\mathbf{w}_0$. It draws samples from a uniform random distribution within $[-limit, limit]$, where $limit = \sqrt{\frac{6}{in+out}}$, and where $in$ and $out$ is the number of input and output units, resp.

[3]:
```
in = 2;
out = 1;
limit = sqrt(6 / (in + out));
ws0 = [rand*2*limit-limit;rand*2*limit-limit]
```

```
ws0 =

    1.3988
    0.6175
```

We assess the loss $MSE$ of $m_1$ for this initial weights setting.

[4]:
```
mse1(ws0)
```

```
ans =

   8.7171e+05
```

The gradient of $MSE(\mathbf{w})$ for any $\mathbf{w}$ is defiend as:

$$
\begin{aligned}
\nabla_{MSE}(\mathbf{w}) &= \left[\frac{\partial MSE(\mathbf{w})}{\partial w_1}, \frac{\partial MSE(\mathbf{w})}{\partial w_2}\right]^T \\
&= \frac{1}{N}\left[\frac{\partial \sum_{i=1}^{N}(y_i - m_1(\mathbf{w}, \mathbf{x}_i)^2}{\partial w_1}, \frac{\partial \sum_{i=1}^{N}(y_i - m_1(\mathbf{w}, \mathbf{x}_i))^2}{\partial w_2}\right]^T \\
&= \frac{1}{N}\left[\sum_{i=1}^{N} 2(y_i - m_1(\mathbf{w}, \mathbf{x}_i))\frac{-\partial m_1(\mathbf{w}, \mathbf{x}_i)}{\partial w_1}, \sum_{i=1}^{N} 2(y_i - m_1(\mathbf{w}, \mathbf{x}_i))\frac{-\partial m_1(\mathbf{w}, \mathbf{x}_i)}{\partial w_2}\right]^T \\
&= -\frac{2}{N}\left[\sum_{i=1}^{N}(y_i - m_1(\mathbf{w}, \mathbf{x}_i))\frac{\partial m_1(\mathbf{w}, \mathbf{x}_i)}{\partial w_1}, \sum_{i=1}^{N}(y_i - m_1(\mathbf{w}, \mathbf{x}_i))\frac{\partial m_1(\mathbf{w}, \mathbf{x}_i)}{\partial w_2}\right]^T
\end{aligned}
$$

We can plug in the function $m_1$ and the first derivative of $m_1$ wrt. $w_1$ and $w_2$ resp.

$$\frac{\partial m_1(\mathbf{w}, \mathbf{x}_i)}{\partial w_1} = \frac{\partial \mathbf{w}^T \mathbf{x}_i}{\partial w_1} = \frac{\partial w_1 x_{i,1} + w_2 x_{i,2}}{\partial w_1} = x_{i,1}$$
$$\frac{\partial m_1(\mathbf{w}, \mathbf{x}_i)}{\partial w_2} = \frac{\partial \mathbf{w}^T \mathbf{x}_i}{\partial w_2} = \frac{\partial w_1 x_{i,1} + w_2 x_{i,2}}{\partial w_2} = x_{i,2}$$

We have already defined the function $m_1$. Let's also define the functions of the first derivative of $m_1$ wrt. $w_1$ and $w_2$ resp. For the sake of generality, we keep $\mathbf{w}$ as a formal parameter `ws` even though it is actually not needed for derivatives of this concrete model $m_1$.

[5]:
```
gradients1{1} = @(ws,x)(x(1));
gradients1{2} = @(ws,x)(x(2));
```

The gradient $\nabla_{MSE}(\mathbf{w})$ has been defined using $m_1$ and the first derivative of $m_1$ wrt. $w_1$ and $w_2$, resp., as parameters.

The gradient descent function optimizes $\mathbf{w}$ by iterating over:

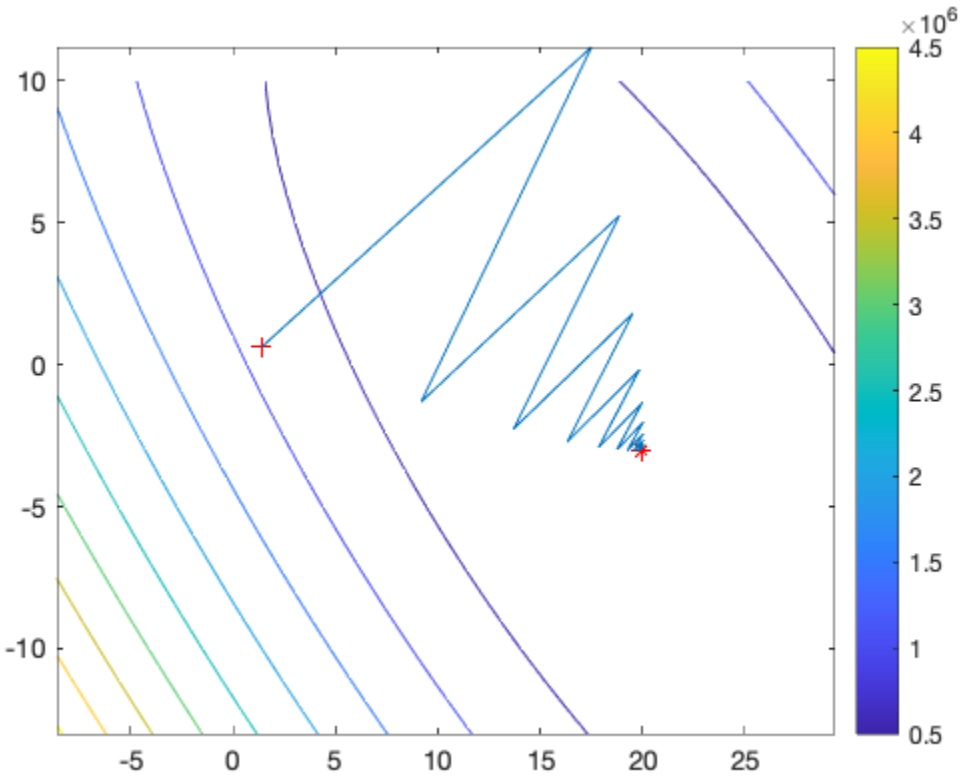
$$\mathbf{w}_{k+1} = \mathbf{w}_k - \varepsilon \nabla_{MSE}(\mathbf{w}_k)$$

staring with $\mathbf{w}_0$.

Below we apply the gradient descent function minimizing $MSE$ for the model $m_1$. We plot the $MSE$ for each step $k$ in the iteration as a function of the current parameter setting $\mathbf{w}_k$. We also marked the starting point $\mathbf{w}_0$ with a `+` and the (ideal) minimum $(20, -3)$ with a `*`.

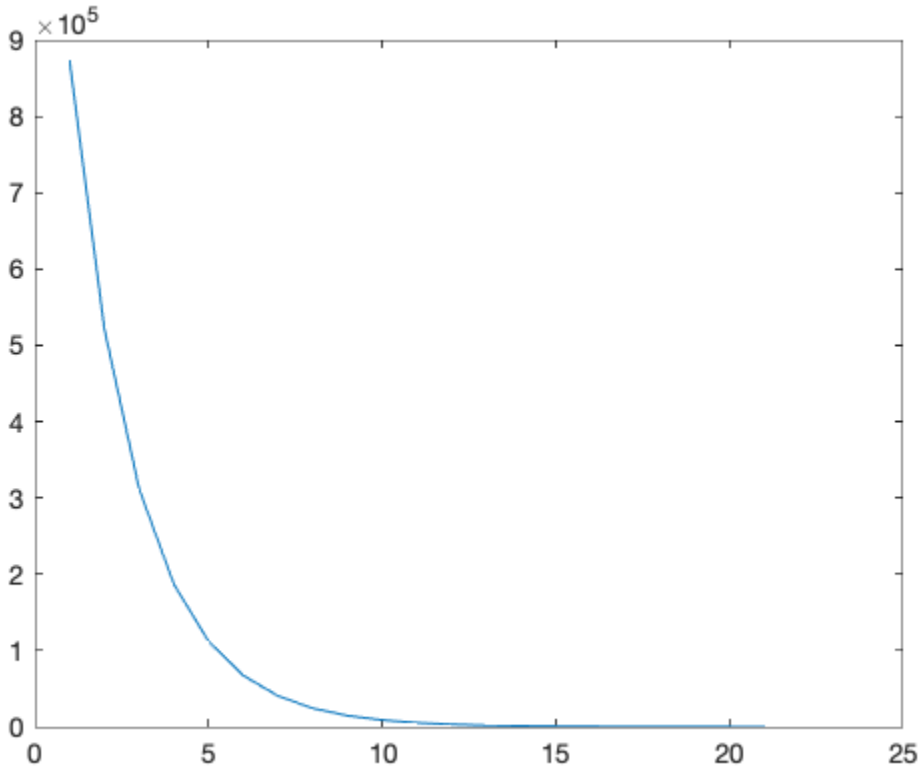
[6]:
```
%% plot search space
f = @(a, b)(mse1([a,b]));
[A,B] = meshgrid(min(a10,ws0(1))-10:1:max(a10,ws0(1))+10,min(a20,ws0(2))-10:1:
 ↪max(a20,ws0(2))+10);
plot3d(f, A, B, false) %3D contour
hold on
plot(a10,a20,'*r')
plot(ws0(1),ws0(2),'+r')

%% gradient descent
grad_loss = @(ws)(grad_mse(ws, m1, gradients1, XX, Y));
K = 20;
learning_eps = 0.00015;

tic;
[ws, history] = grad_desc_mse(K, ws0, learning_eps, mse1, grad_loss, true);
toc
hold off
```

```
Elapsed time is 0.258120 seconds.
```




The history plot shows the convergence of the algorithm. After ca. $k = 15$ iterations, the loss $MSE$ is almost zero.

[7]:
```
plot(1:length(history),history)
```




### 1.3 Stochastic Gradient Descent (SGD)

SGD works like the gradient descent but computes the gradient based on a mini batch of size $m \ll N$. Therefore, our SGD implementation `stochastic_grad_desc_mse` generates in each iteration an array `randices`, i.e., random indices making up the current mini batch sample of the whole data set. We resample this array in each iteration of the SGD algorithm.

We also define a corresponding `grad_mse2` function that computes the gradient only based on the mini batch data points.

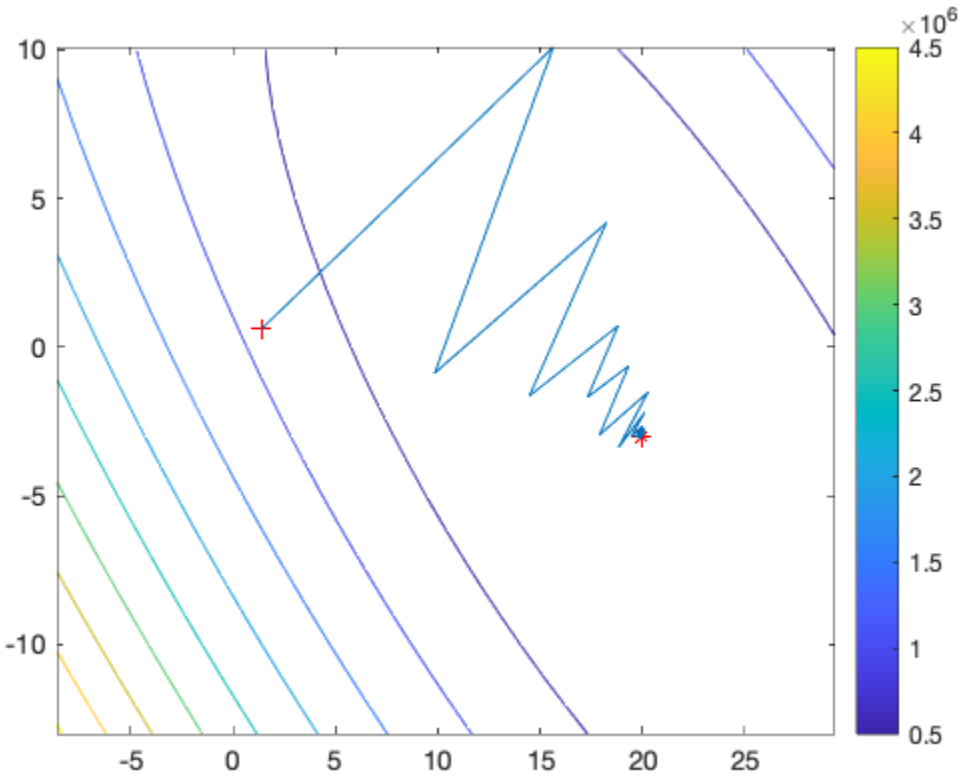
[8]:
```
%%file stochastic_grad_desc_mse.m
function [ws, history] = stochastic_grad_desc_mse(K, ws, learning_eps, loss,␣
 ↪grad_loss, N, verbose)
    batch_size = N*0.01;
    history(1) = loss(ws);
    for k = 1:K
        randices = randsample(1:N,batch_size,false);
        grad_ws = grad_loss(ws, randices);
        old_ws = ws;
        ws= old_ws - learning_eps * grad_ws;
        if verbose
```

```
            line([old_ws(1),ws(1)],[old_ws(2),ws(2)]);
        end
        history(k+1) = loss(ws);
    end
end
```

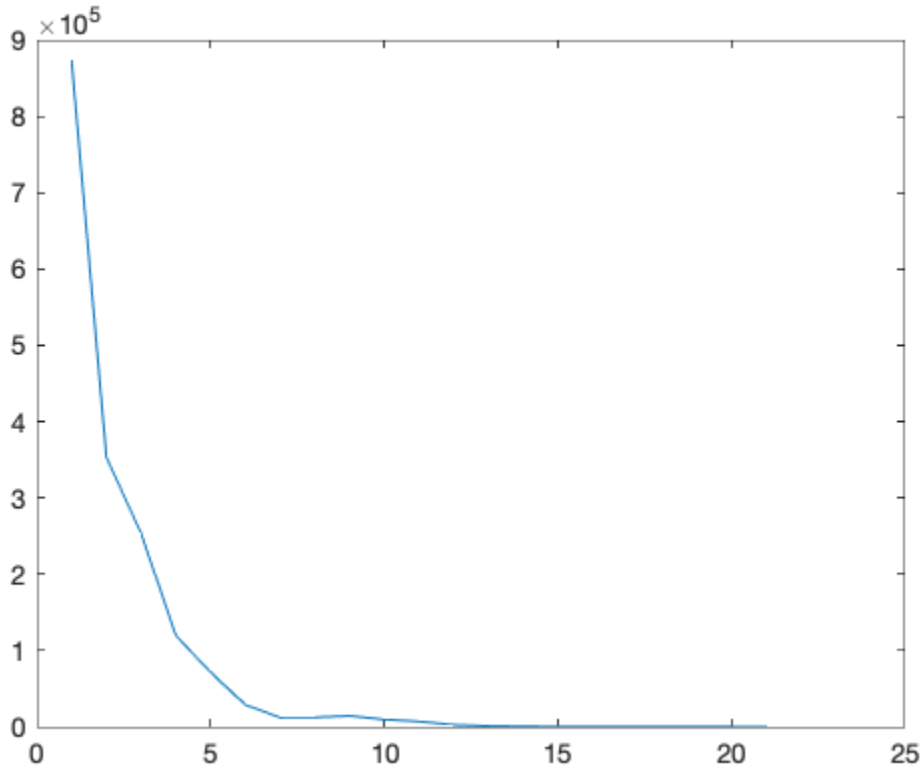
```
Created file '/Users/wlomsi/Documents/ProjekteUni/Vorlesungen/ML
4DV660+4DV661+4DV652/Public ML Notebooks/stochastic_grad_desc_mse.m'.
```

[9]:
```
%% plot search space
plot3d(f, A, B, false) %3D contour
hold on
plot(a10,a20,'*r')
plot(ws0(1),ws0(2),'+r')

%% stochastic gradient descent
grad_loss2 = @(ws, randices)(grad_mse(ws, m1, gradients1, XX(randices,:
 ↪),Y(randices)));
tic;
[ws, history] = stochastic_grad_desc_mse(K, ws0, learning_eps, mse1,␣
 ↪grad_loss2, N*N, true);
toc
```

```
Elapsed time is 0.064421 seconds.
```




[10]:
```
plot(1:length(history),history)
```




### 1.4 SGD with adaptive learning rate

The learning rate $\varepsilon$ is adapted in each iteration, e.g., exponentially by $\varepsilon_{k+1} = (1-\alpha)\varepsilon_k$ with $\alpha$ a new hyper-parameter.

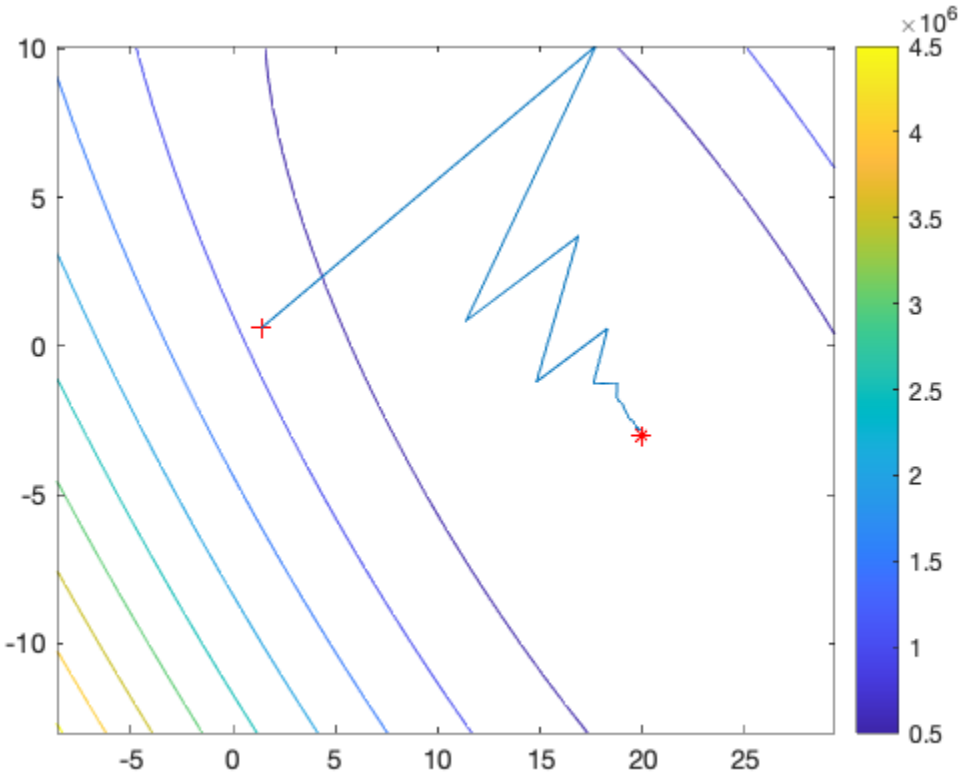
[11]:
```
%%file stochastic_adaptive_grad_desc_mse.m
function [ws, history] = stochastic_adaptive_grad_desc_mse(K, ws, learning_eps,␣
↪loss, grad_loss, N, alpha, verbose)
    batch_size = N*0.01;
    history(1) = loss(ws);
    for k = 1:K
        randices = randsample(1:N,batch_size,false);
        grad_ws = grad_loss(ws, randices);
        old_ws = ws;
        ws= old_ws - learning_eps * grad_ws;
        learning_eps = (1-alpha) * learning_eps;
        if verbose
            line([old_ws(1),ws(1)],[old_ws(2),ws(2)]);
        end
        history(k+1) = loss(ws);
    end
```

```
end
```

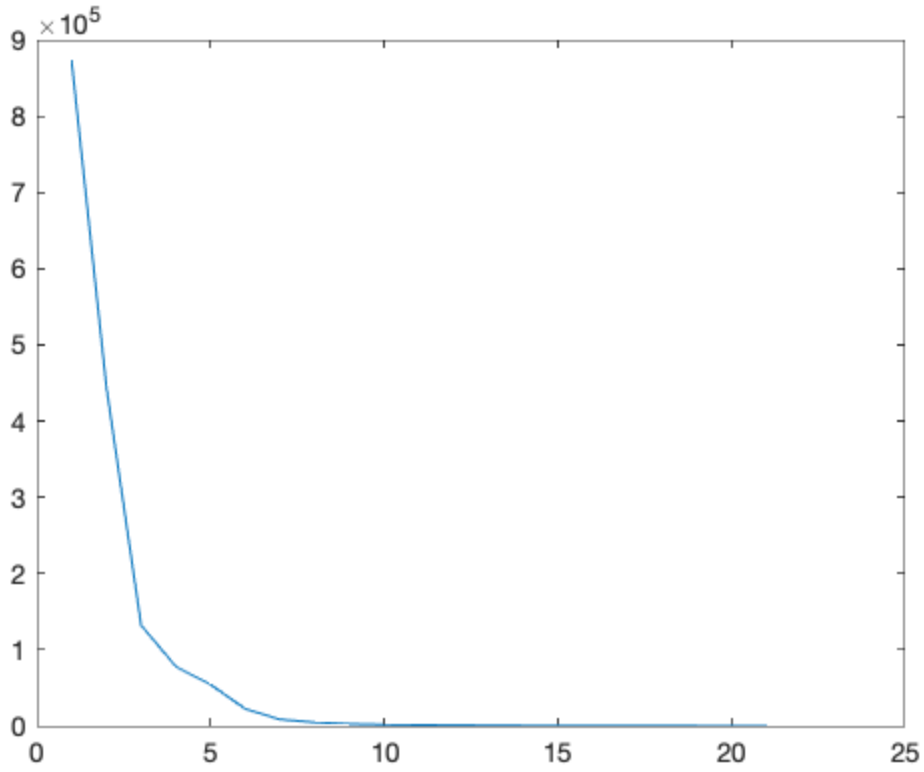
```
Created file '/Users/wlomsi/Documents/ProjekteUni/Vorlesungen/ML
4DV660+4DV661+4DV652/Public ML Notebooks/stochastic_adaptive_grad_desc_mse.m'.
```

[12]:
```
%% plot search space
plot3d(f, A, B, false) %3D contour
hold on
plot(a10,a20,'*r')
plot(ws0(1),ws0(2),'+r')

%% stochastic adaptive gradient descent
alpha = 0.03;
tic;
[ws, history] = stochastic_adaptive_grad_desc_mse(K, ws0, learning_eps, mse1,␣
 ↪grad_loss2, N*N, alpha, true);
toc
```

```
Elapsed time is 0.041576 seconds.
```



[13]:
```
plot(1:length(history),history)
```




### 1.5 SGD with momentum

The gradient can be understood as a velocity at which we move towards the optimum. In this analogy, momentum $p$ is introduced as the velocity $v$ (gradient) times mass $m$, a new hyper-parameter of the algorithm. The velocity is set to the average of past gradients (with importance of past velocities decaying exponentially). The initial velocity $v_0$ is the initial gradient.

**OBS!** The velocity (earlier gradient vectors) and the current gradient vector point "uphill", i.e, into the inverse direction of the expected minimum. They should, hence, be **both** subtracted from the current parameters.

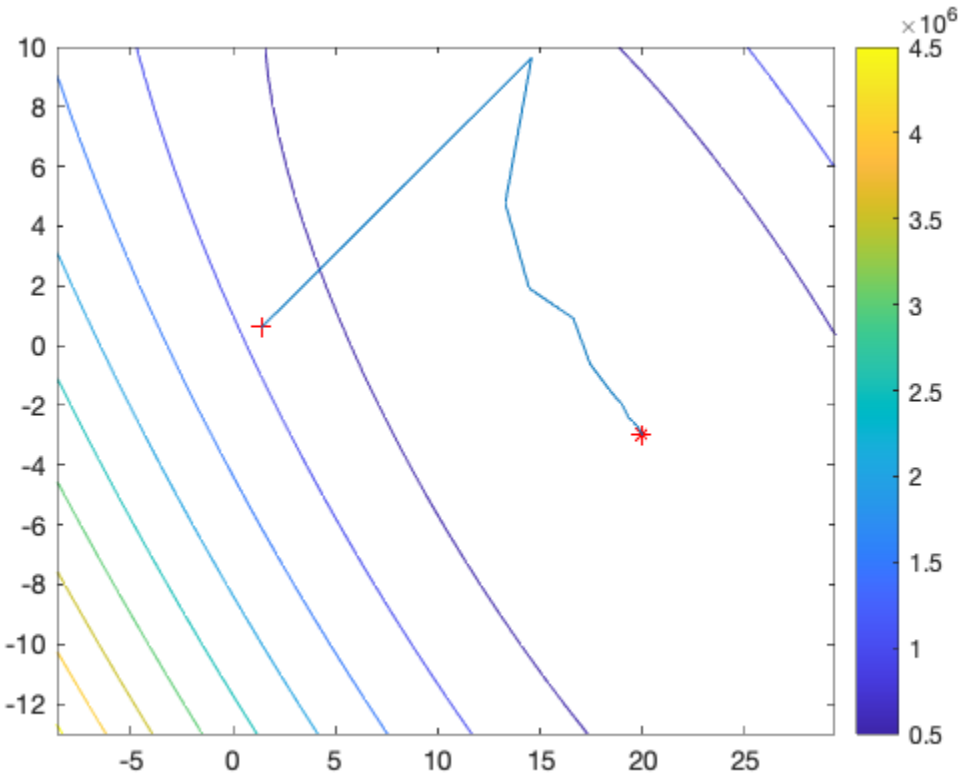
[14]:
```
%%file stochastic_momentum_grad_desc_mse.m
function [ws, history] = stochastic_momentum_grad_desc_mse(K, ws, learning_eps,␣
↪loss, grad_loss, N, mass, verbose)
    batch_size = N*0.01;
    history(1) = loss(ws);
    v = 0;
```

```
    for k = 1:K
        randices = randsample(1:N,batch_size,false);
        grad_ws = grad_loss(ws, randices);
        v = (v+grad_ws)/2;
        old_ws = ws;
        ws= old_ws - mass * v - learning_eps * grad_ws;
        if verbose
            line([old_ws(1),ws(1)],[old_ws(2),ws(2)]);
        end
        history(k+1) = loss(ws);
    end
end
```

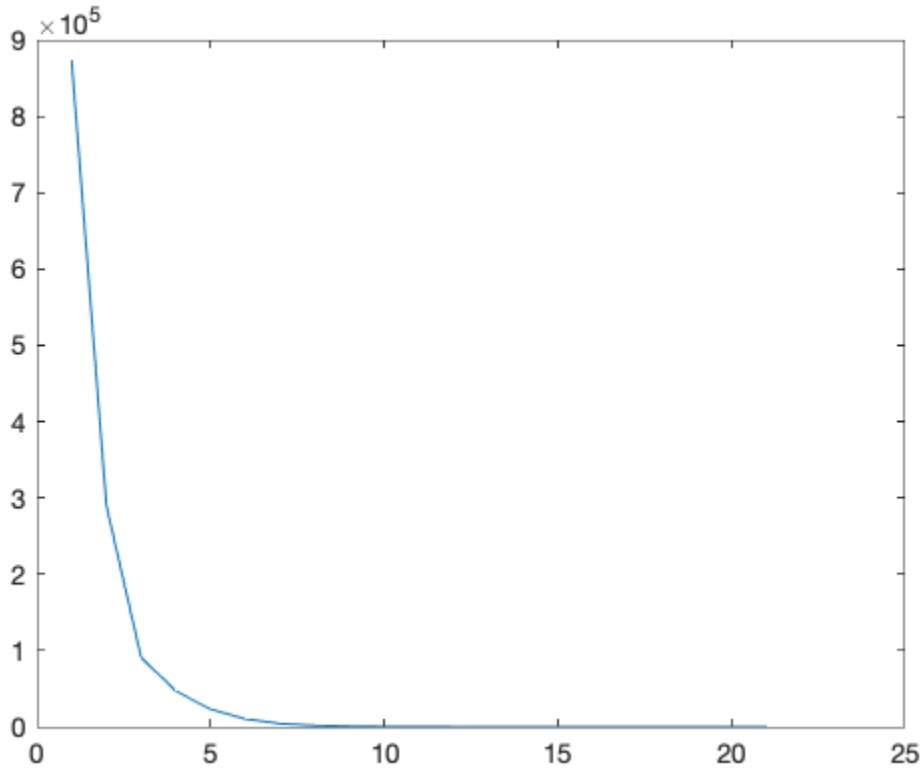
```
Created file '/Users/wlomsi/Documents/ProjekteUni/Vorlesungen/ML
4DV660+4DV661+4DV652/Public ML Notebooks/stochastic_momentum_grad_desc_mse.m'.
```

[15]:
```
%% plot search space
plot3d(f, A, B, false) %3D contour
hold on
plot(a10,a20,'*r')
plot(ws0(1),ws0(2),'+r')

%% stochastic momentum gradient descent
mass = 1/2 * 0.00015;
learning_eps = 1/2 * 0.00015;
tic;
[ws, history] = stochastic_momentum_grad_desc_mse(K, ws0, learning_eps, mse1,␣
 ↪grad_loss2, N*N, mass, true);
toc
```

```
Elapsed time is 0.046022 seconds.
```




[16]:
```
plot(1:length(history),history)
```




### 1.6 SGD with accumulated squared gradient: AdaGrad

With AdaGrad, each parameter has its own learning rate. This learning rate is always decreasing, but faster for the parameters with the larger gradients. The derease is quite fast, so we have to adapt the constant component $\varepsilon$ of the learning rate.

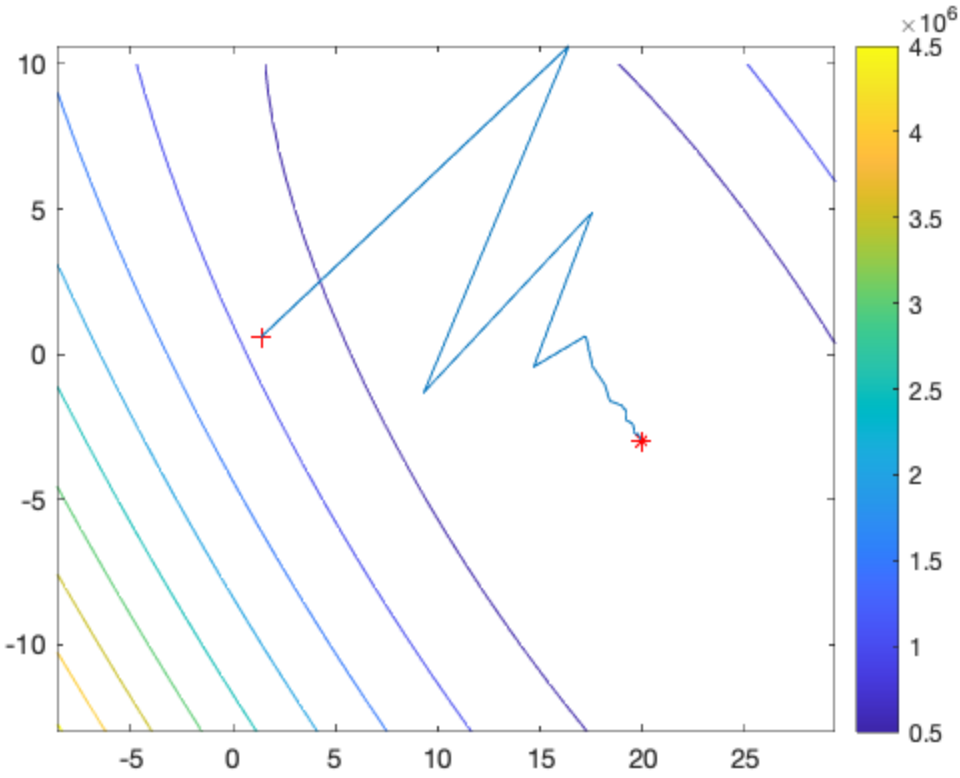
[45]:
```
%%file ada_grad_mse.m
function [ws, history] = ada_grad_mse(K, ws, learning_eps, loss, grad_loss, N,␣
 ↪verbose)
    batch_size = N*0.01;
    history(1) = loss(ws);
    r = zeros(length(ws),1);
    delta = 1e-10*ones(length(ws),1);
    for k = 1:K
        randices = randsample(1:N,batch_size,false);
        grad_ws = grad_loss(ws, randices);
        old_ws = ws;
        r = r + grad_ws .* grad_ws;
        ws= old_ws - learning_eps/(delta+sqrt(r)) .* grad_ws;
        if verbose
            line([old_ws(1),ws(1)],[old_ws(2),ws(2)]);
```

```
        end
        history(k+1) = loss(ws);
    end
end
```

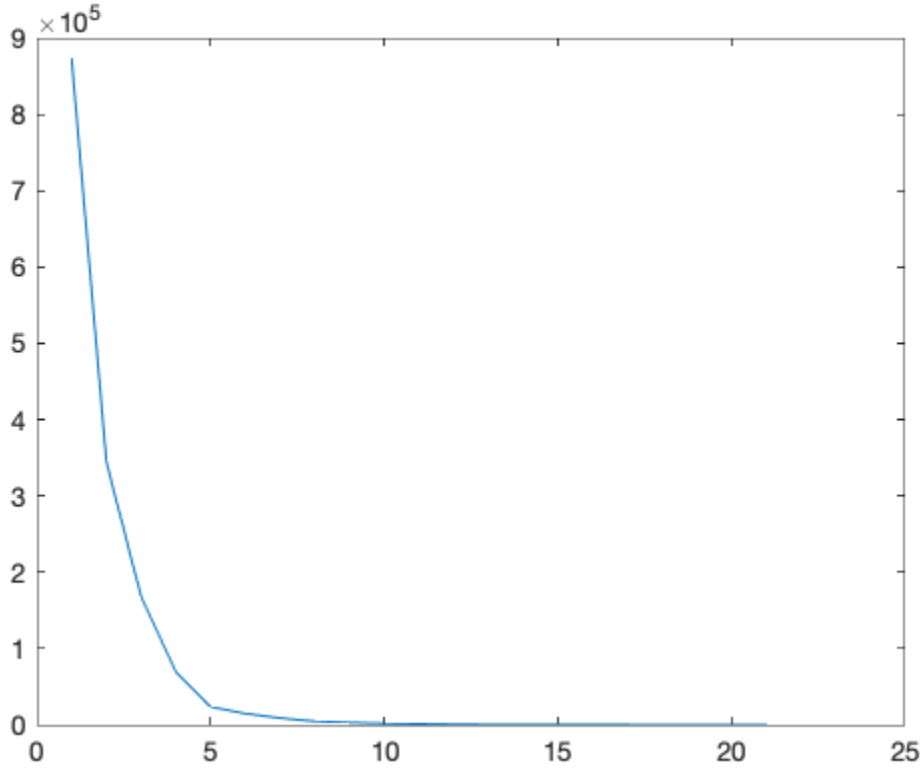
```
Created file '/Users/wlomsi/Documents/ProjekteUni/Vorlesungen/ML
4DV660+4DV661+4DV652/Public ML Notebooks/ada_grad_mse.m'.
```

[47]:
```
%% plot search space
plot3d(f, A, B, false) %3D contour
hold on
plot(a10,a20,'*r')
plot(ws0(1),ws0(2),'+r')

%% AdaGrad
learning_eps = 15;
tic;
[ws, history] = ada_grad_mse(K, ws0, learning_eps, mse1, grad_loss2, N*N, true);
toc
```

```
Elapsed time is 0.037942 seconds.
```



[19]:
```
plot(1:length(history),history)
```




### 1.7 SGD with accumulated squared gradient: RMSProp

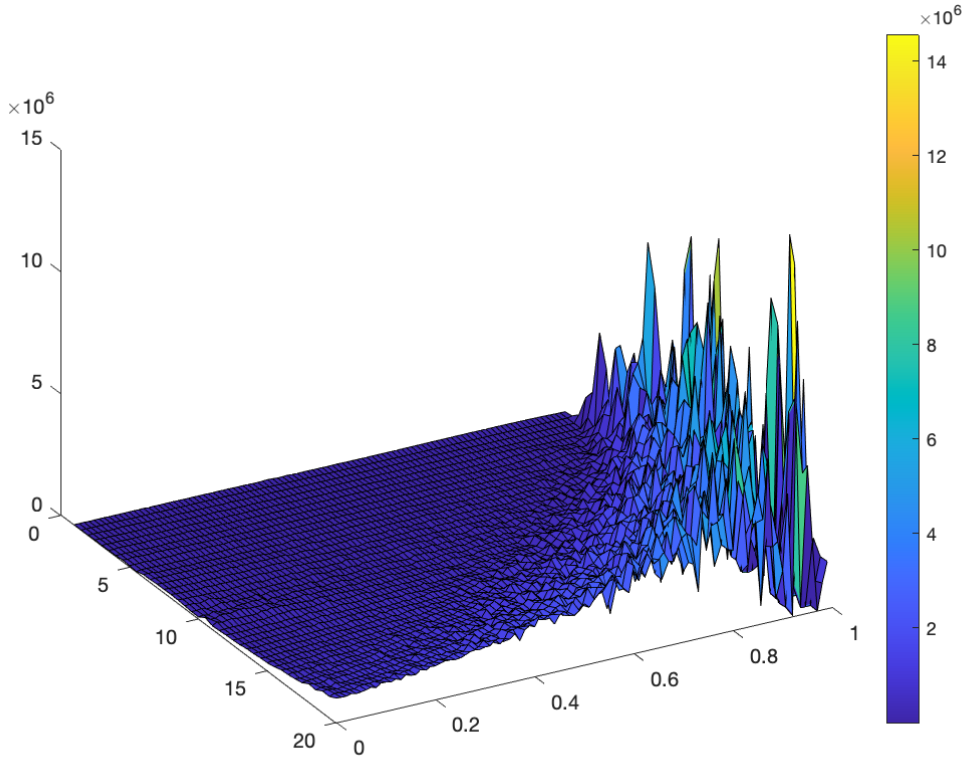
RMSProp adds a hyper-parameter $\rho \in [0 \dots 1]$ to AdaGrad. It biases between the accumulated squared gradients $r$, i.e., the historic (squared) velocity, and the current squared gradient before deviding the learning rate by $\sqrt{r}$.

[61]:
```
%%file rms_prob_mse.m
function [ws, history] = rms_prob_mse(K, ws, learning_eps, loss, grad_loss, N,␣
 ↪rho, verbose)
    batch_size = N*0.01;
    history(1) = loss(ws);
    r = zeros(length(ws),1);
    delta = 1e-10*ones(length(ws),1);
    for k = 1:K
        randices = randsample(1:N,batch_size,false);
        grad_ws = grad_loss(ws, randices);
```

```
        old_ws = ws;
        r = rho*r + (1-rho) * grad_ws .* grad_ws;
        ws= old_ws - learning_eps/(sqrt(delta+r)) .* grad_ws;
        if verbose
            line([old_ws(1),ws(1)],[old_ws(2),ws(2)]);
        end
        history(k+1) = loss(ws);
    end
end
```

```
Created file '/Users/wlomsi/Documents/ProjekteUni/Vorlesungen/ML
4DV660+4DV661+4DV652/Public ML Notebooks/rms_prob_mse.m'.
```

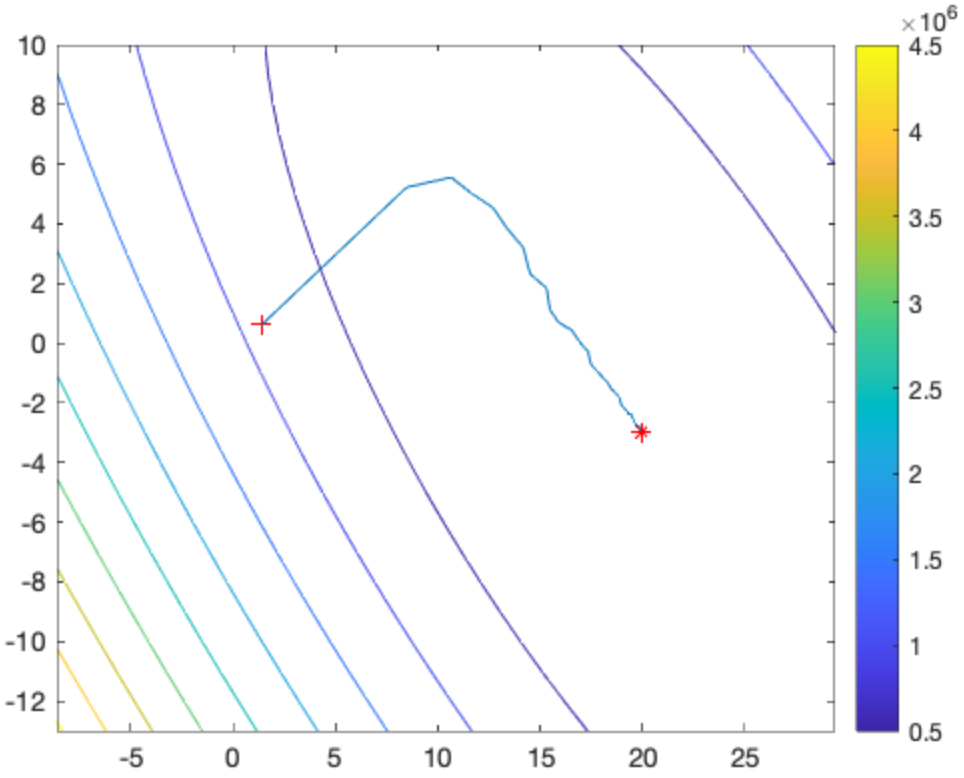
For RMSProb, it was hard to find good hyper-parameters just by trying out a few. Here the loss for different hyper-parameter settings of $\rho \in [0 \dots 1]$ and $\varepsilon \in [1 \dots 20]$. We finally chose the setting with the minimum loss after 20 iterations: $\varepsilon = 1, \rho = 0.98$. Note, that we needed to increase the number of iterations to arrive at a the ideal parameters. So, for this simplistic setup, RMSProb did not speed up convergence.



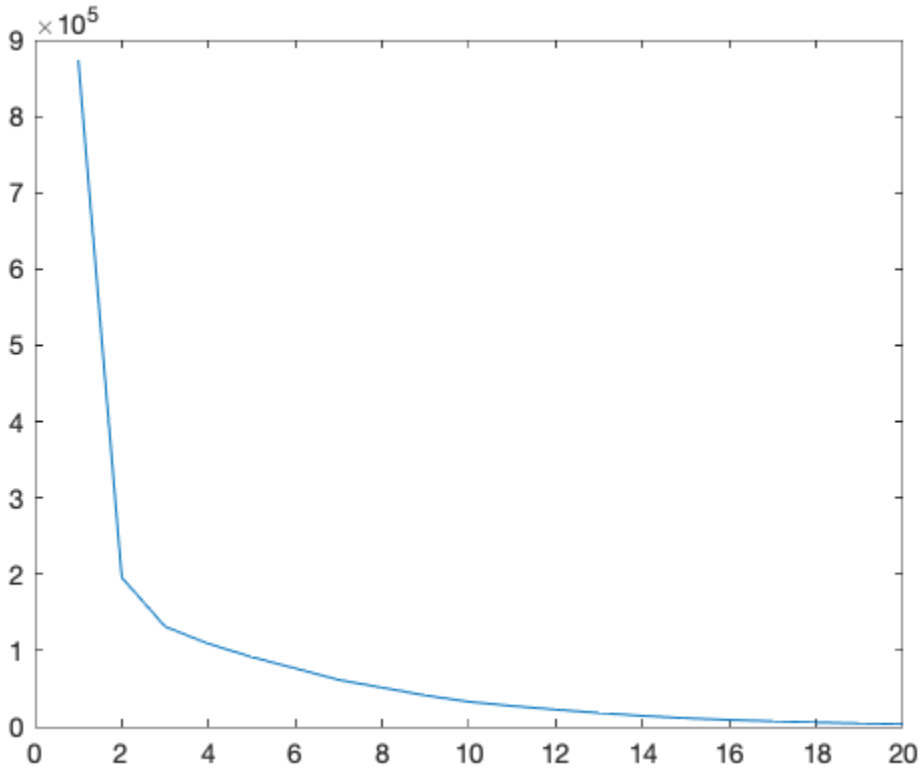

[67]:
```
%% plot search space
plot3d(f, A, B, false) %3D contour
hold on
plot(a10,a20,'*r')
```

```
plot(ws0(1),ws0(2),'+r')

%% RMSProb;
learning_eps = 1;
rho = 0.98;
K = 40; %!
tic;
[ws, history] = rms_prob_mse(K, ws0, learning_eps, mse1, grad_loss2, N*N, rho,␣
 ↪true);
toc
```

```
Elapsed time is 0.052475 seconds.
```




[22]:
```
plot(1:20,history(1:20))
```




### 1.8 SGD with accumulated squared gradient: ADAM

ADAM seems to be the current choice of many. However, it hasn't been trivial to choose the learning rate and the two hyper-parameters $\rho_1, \rho_2$. We even needed to increase the number of iterations to arrive at a the ideal parameters. So, for this simplistic setup, ADAM did not speed up convergence.

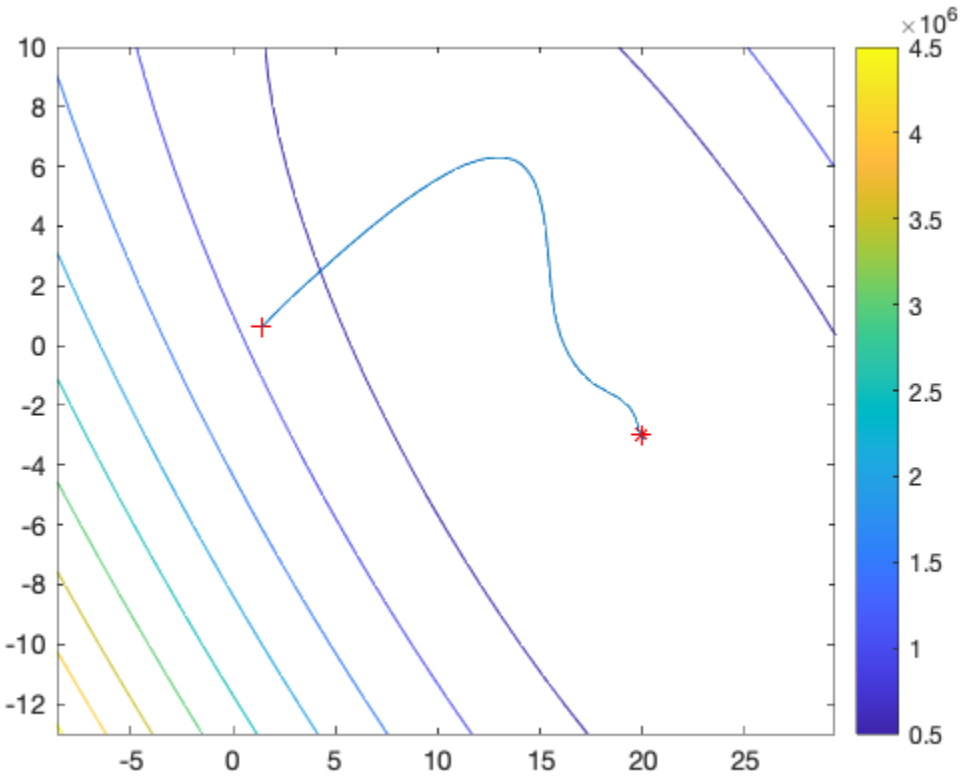
[48]:
```
%%file adam_mse.m
function [ws, history] = adam_mse(K, ws, learning_eps, loss, grad_loss, N,␣
 ↪rho1, rho2, verbose)
    batch_size = N*0.01;
    history(1) = loss(ws);
    s = zeros(length(ws),1);
    r = zeros(length(ws),1);
    t = 0;
    delta = 1e-10*ones(length(ws),1);
    for k = 1:K
        randices = randsample(1:N,batch_size,false);
        grad_ws = grad_loss(ws, randices);
        old_ws = ws;
        t = t +1;
```

```
        s = rho1*s + (1-rho1) * grad_ws;
        r = rho2*r + (1-rho2) * grad_ws .* grad_ws;
        s_hat = s/(1-rho1^t);
        r_hat = r/(1-rho2^t);
        ws= old_ws - (learning_eps*s_hat)/(delta+sqrt(r_hat));
        if verbose
            line([old_ws(1),ws(1)],[old_ws(2),ws(2)]);
        end
        history(k+1) = loss(ws);
    end
end
```

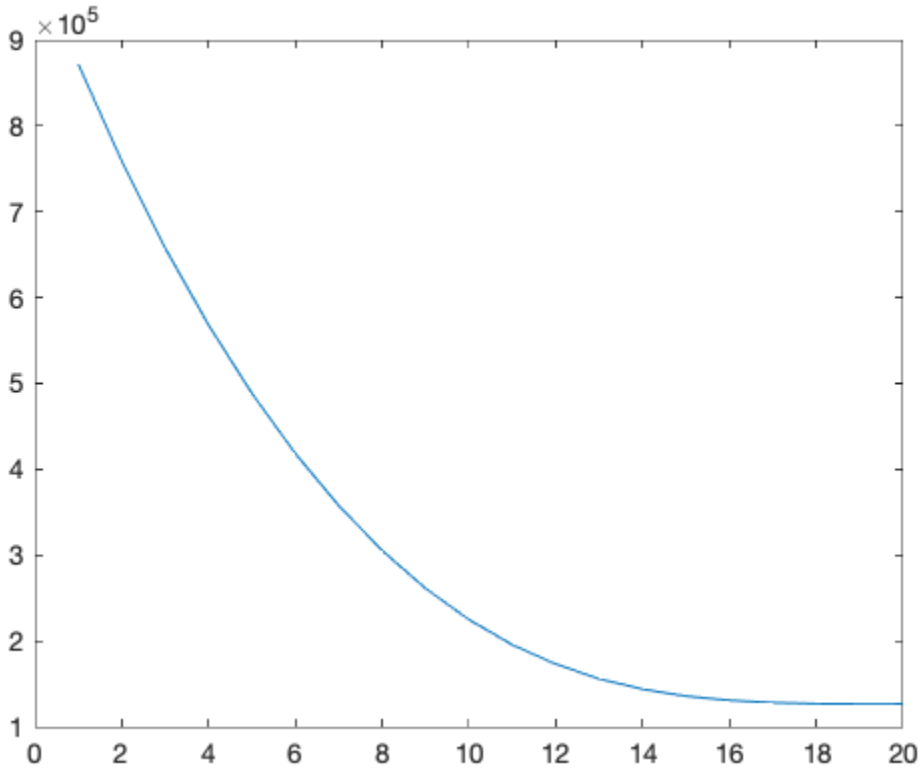
```
Created file '/Users/wlomsi/Documents/ProjekteUni/Vorlesungen/ML
4DV660+4DV661+4DV652/Public ML Notebooks/adam_mse.m'.
```

[59]:
```
%% plot search space
plot3d(f, A, B, false) %3D contour
hold on
plot(a10,a20,'*r')
plot(ws0(1),ws0(2),'+r')

%% ADAM;
learning_eps = 0.75;
rho1 = 0.9;
rho2 = 0.999;
K=100; %!!
tic;
[ws, history] = adam_mse(K, ws0, learning_eps, mse1, grad_loss2, N*N, rho1,␣
 ↪rho2, true);
toc
```

```
Elapsed time is 0.123435 seconds.
```




[60]:
```
plot(1:20,history(1:20))
```




### 1.9 SGD with Newton's method

It is obviously difficult to set the learning rate (and the other parameters) right. To help this, Newton's method (and other second order methods) involve the second order derivatives, e.g., the Hessian matrix $H_{MSE}$. Then they iterate:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - H_{MSE}(\mathbf{w}_k)^{-1}\nabla_{MSE}(\mathbf{w}_k)$$

where $H_{MSE}$ is the Hessian matrix of $MSE$.

$H_{MSE}$ is defined as:

$$\begin{aligned}
H_{MSE}(\mathbf{w}) &= \begin{bmatrix} \frac{\partial^2 MSE(\mathbf{w})}{\partial w_1^2} & \frac{\partial^2 MSE(\mathbf{w})}{\partial w_1 \partial w_2} \\ \frac{\partial^2 MSE(\mathbf{w})}{\partial w_2 \partial w_1} & \frac{\partial^2 MSE(\mathbf{w})}{\partial w_2^2} \end{bmatrix} \\
&= \frac{-2}{N} \begin{bmatrix} \frac{\partial(\sum_{i=1}^N (y_i - m_1(\mathbf{w},\mathbf{x}_i)x_{i,1})}{\partial w_1} & \frac{\partial(\sum_{i=1}^N (y_i - m_1(\mathbf{w},\mathbf{x}_i)x_{i,1})}{\partial w_2} \\ \frac{\partial(\sum_{i=1}^N (y_i - m_1(\mathbf{w},\mathbf{x}_i)x_{i,2})}{\partial w_1} & \frac{\partial(\sum_{i=1}^N (y_i - m_1(\mathbf{w},\mathbf{x}_i)x_{i,2})}{\partial w_2} \end{bmatrix} \\
&= \frac{2}{N} \begin{bmatrix} \sum_{i=1}^N x_{i,1}^2 & \sum_{i=1}^N x_{i,1}x_{i,2} \\ \sum_{i=1}^N x_{i,1}x_{i,2} & \sum_{i=1}^N x_{i,2}^2 \end{bmatrix}
\end{aligned}$$

This approach generalizes easily to a stochastic version of Newtons method using only a sample of the training data of size $m \ll N$.

[26]:
```
%%file hessian_mse.m
function H = hessian_mse(ws, X)
    N = length(X);
    M = length(ws);
    H = zeros(M);
    for r=1:M
        for c=1:M
            for i=1:N
                H(r,c) = H(r,c) + X(i,r)*X(i,c);
            end
        end
    end
    H = 2/N*H;
end
```

```
Created file '/Users/wlomsi/Documents/ProjekteUni/Vorlesungen/ML
4DV660+4DV661+4DV652/Public ML Notebooks/hessian_mse.m'.
```

[27]:
```
grad2_loss = @(ws, randices)(hessian_mse(ws, XX(randices,:)));
```

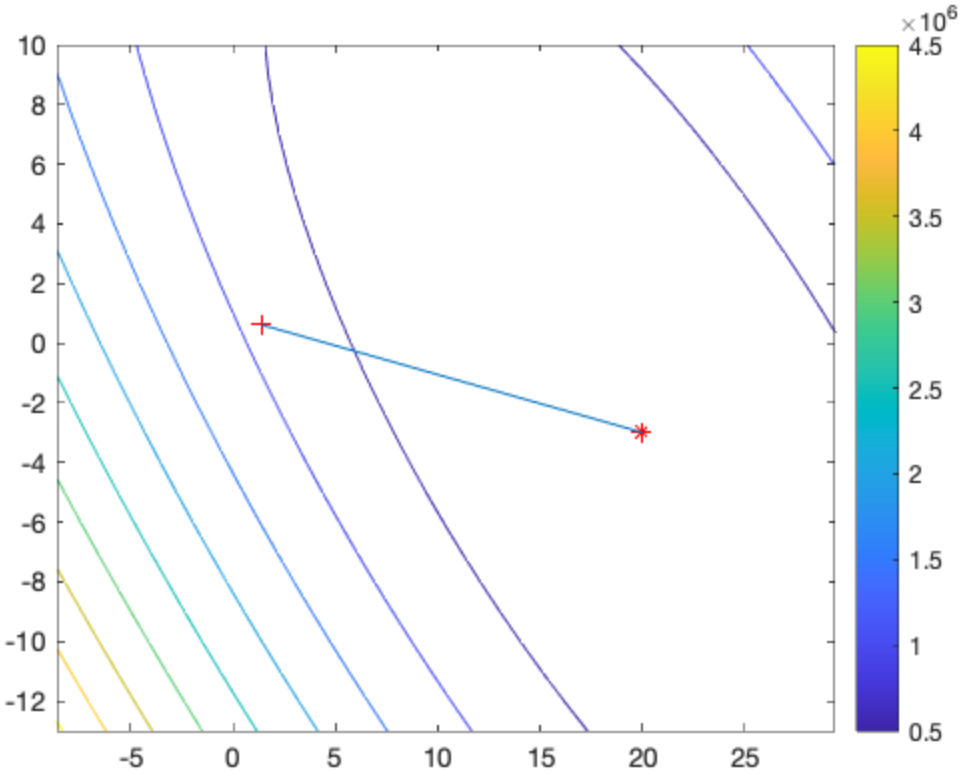
[28]:
```
%%file stochastic_newton_mse.m
function [ws, history] = stochastic_newton_mse(K, ws, loss, grad_loss,␣
 ↪grad2_loss, N, verbose)
    batch_size = N*0.01;
    history(1) = loss(ws);
    for k = 1:K
        randices = randsample(1:N,batch_size,false);
        grad_ws = grad_loss(ws, randices);
        old_ws = ws;
        H = grad2_loss(ws, randices);
        ws= old_ws - inv(H) * grad_ws;
        if verbose
            line([old_ws(1),ws(1)],[old_ws(2),ws(2)]);
        end
        history(k+1) = loss(ws);
    end
end
```

```
Created file '/Users/wlomsi/Documents/ProjekteUni/Vorlesungen/ML
4DV660+4DV661+4DV652/Public ML Notebooks/stochastic_newton_mse.m'.
```

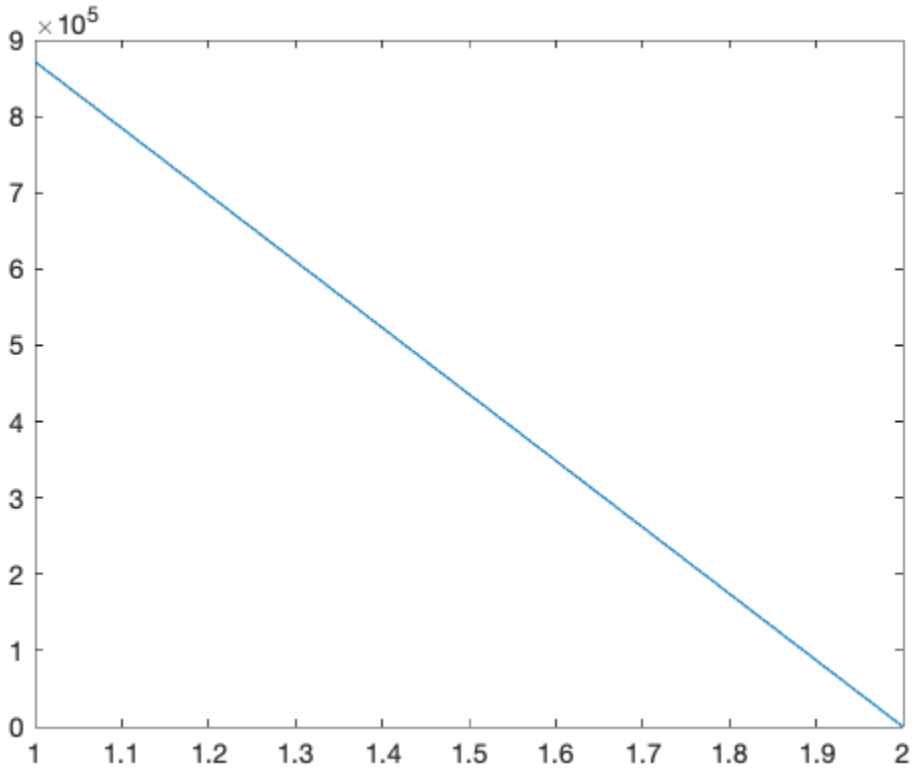
[29]:
```
%% plot search space
plot3d(f, A, B, false) %3D contour
```

```
hold on
plot(a10,a20,'*r')
plot(ws0(1),ws0(2),'+r')

%% Newton's method;
K=1; %!!!
tic;
[ws, history] = stochastic_newton_mse(K, ws0, mse1, grad_loss2, grad2_loss,␣
 ↪N*N, true);
toc
```

```
Elapsed time is 0.019463 seconds.
```




[30]:
```
plot(1:length(history),history)
```




### 1.10 SGD with the conjugate gradient method

The Hessian is quite large, with a size $O(n^2)$ for $n$ parameters, and needs to be updated in each iteration. The conjugate gradient method tries to avoid the explicite calculation of the Hessian.

Part of the algorithm is a search along the (adjusted) gradient line for the optimum (lowest loss) point on this line. Therefore, we would like to use the mini batch instead of the whole training data set. We define a stochastic loss function accordingly.

[31]:
```
mse2 = @(ws, randices)(mse(ws,m1,XX(randices,:),Y(randices)));
```

In *linear* conjugate gradient descent, we search along the gradient line for the point to adjust the parameters to. For *non-linear* conjugate gradient descent, we adjust the gradient line before this linear search based on the gradients' history.

**OBS!** For non-linear conjugate gradient descent, the gradient line adjustment (actually computing its scaling factor $\beta$) in iteration $k$ requires the gradient of iteration $k-1$. This gradient is not available in the first iteration. Therefore, we need to do a linear conjugate gradient descent in the first iteration.

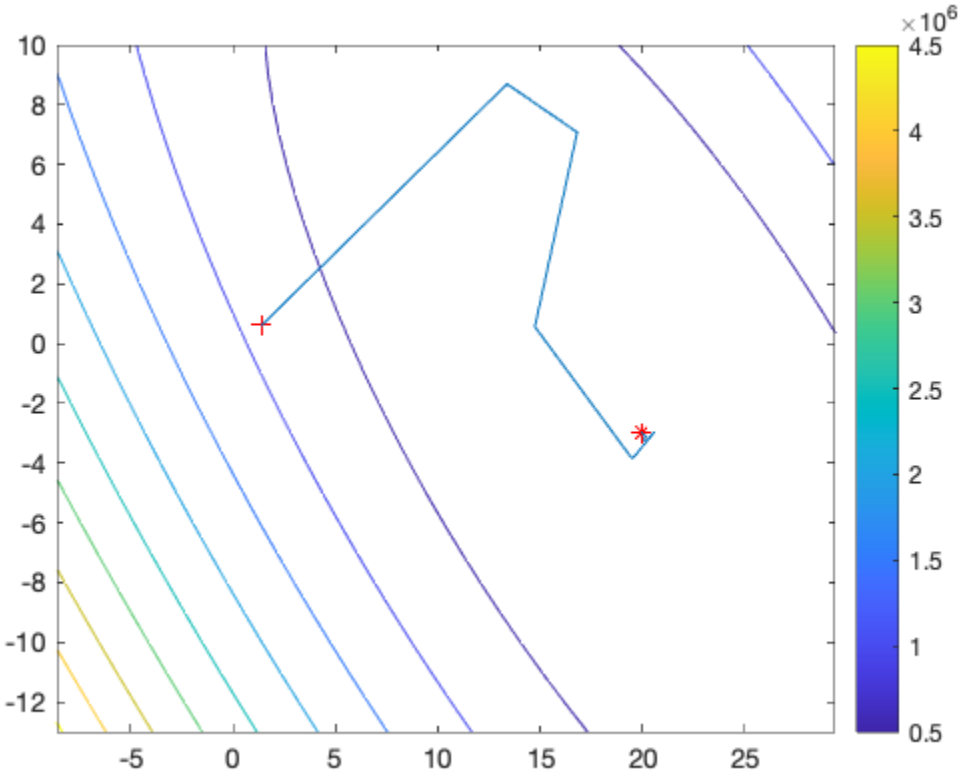
[68]:
```
%%file cg_mse.m
function [ws, history] = cg_mse(K, ws, loss, stochastic_loss, grad_loss, N,␣
↪verbose)
    %initialization
    batch_size = N*0.01;
    history(1) = loss(ws);
    %we only need variables at t (var) and t-1 (var_old)
    rho_old = zeros(length(ws),1);
    grad_ws_old = zeros(length(ws),1);

    for k = 1:K
        randices = randsample(1:N,batch_size,false);
        %Compute gradient
        grad_ws = grad_loss(ws, randices);
        if k==1
            %Linear cg, i.e., no search direction adjustment in the first round
            rho = - grad_ws;
        else
            %Compute Polak-Ribière
            beta = ((grad_ws - grad_ws_old)' * grad_ws) /(grad_ws_old' *␣
↪grad_ws_old);
            %Compute search direction
            rho = - grad_ws + beta*rho_old;
        end
        %Naive line search for epsilon* = argmin stochastic_loss(ws+epsilon *␣
↪rho)
        epsilon_star = 0.00001;
        minimum_star = stochastic_loss(ws+epsilon_star*rho,randices);
        for epsilon = 0.00001:0.0001:1
            minimum = stochastic_loss(ws+epsilon*rho,randices);
            if minimum < minimum_star
                epsilon_star = epsilon;
                minimum_star = minimum;
            end
        end
        %Remember parameters (for drawing the line of this step if verbose)
        old_ws = ws;
        %Update the parameters
        ws= old_ws + epsilon_star * rho;
        %Remember the variables at t as the old variables at t-1 in the next␣
↪iteration
        rho_old = rho;
        grad_ws_old = grad_ws;
        %Draw the line
        if verbose
            line([old_ws(1),ws(1)],[old_ws(2),ws(2)]);
        end
```

```
        history(k+1) = loss(ws);
    end
end
```

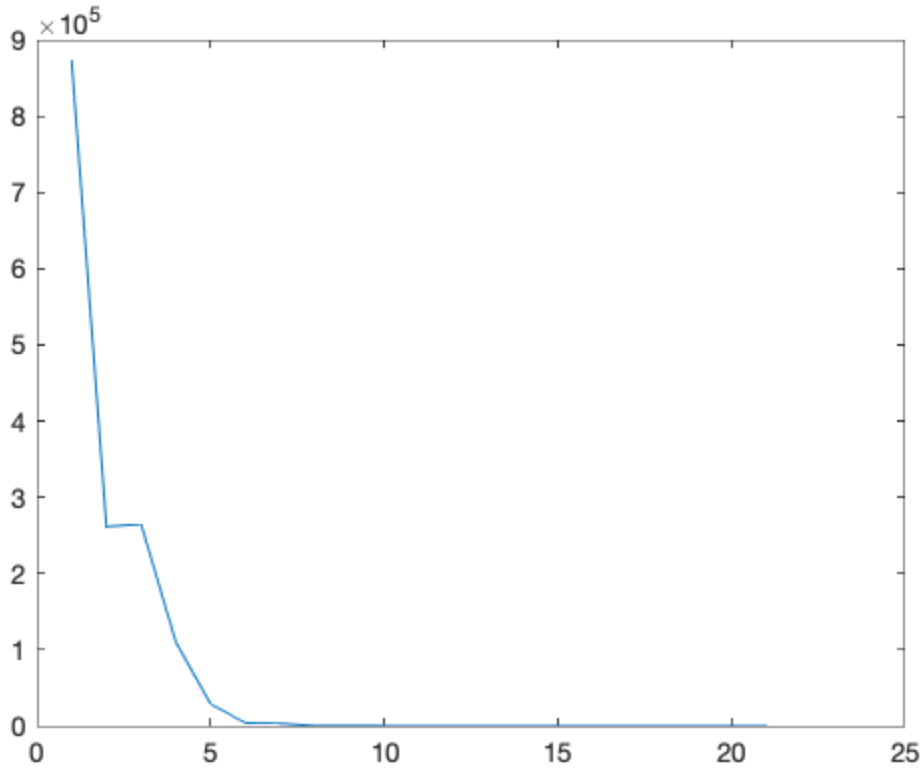
```
Created file '/Users/wlomsi/Documents/ProjekteUni/Vorlesungen/ML
4DV660+4DV661+4DV652/Public ML Notebooks/cg_mse.m'.
```

[70]:
```
%% plot search space
plot3d(f, A, B, false) %3D contour
hold on
plot(a10,a20,'*r')
plot(ws0(1),ws0(2),'+r')

%%CG;
K=20;
tic;
[ws, history] = cg_mse(K, ws0, mse1, mse2, grad_loss2, N*N, true);
toc
```

```
Elapsed time is 1.801698 seconds.
```



[34]:
```
plot(1:length(history),history)
```




The line search that we implemented naïvely in the above algorithm is actually an optimization problem. Since, we only have one variable $\varepsilon$ to optimize for, we could use the quickly converging Newton method to implement this optimization.

Substituting $\mathbf{w} \rightarrow \mathbf{w} + \varepsilon\boldsymbol{\rho}$, we seek to find $\varepsilon^* = \arg\min_\varepsilon MSE(\mathbf{w} + \varepsilon\boldsymbol{\rho}, m, X, Y)$. This leads to the following function to optimize:

$$MSE^*(\varepsilon, \mathbf{w}, \boldsymbol{\rho}, m, X, Y) = \frac{1}{N}\sum_{i=1}^{N}(y_i - m(\mathbf{w} + \varepsilon\boldsymbol{\rho}, \mathbf{x}_i))^2$$

Recall, that only $\varepsilon$ is a variable in this function; all others are constant.

To apply Newton's method, we find the first and second derivations of $MSE^*$:

The gradient of $MSE^*(\varepsilon)$ is:

$$
\begin{aligned}
\nabla_{MSE^*}(\varepsilon) &= \frac{\partial MSE^*(\varepsilon)}{\partial \varepsilon} \\
&= \frac{1}{N} \frac{\partial \sum_{i=1}^{N} (y_i - m_1(\mathbf{w} + \varepsilon \boldsymbol{\rho}, \mathbf{x}_i))^2}{\partial \varepsilon} \\
&= \frac{1}{N} \sum_{i=1}^{N} 2(y_i - m_1(\mathbf{w} + \varepsilon \boldsymbol{\rho}, \mathbf{x}_i)) \frac{-\partial m_1(\mathbf{w} + \varepsilon \boldsymbol{\rho}, \mathbf{x}_i)}{\partial \varepsilon} \\
&= -\frac{2}{N} \sum_{i=1}^{N} (y_i - m_1(\mathbf{w} + \varepsilon \boldsymbol{\rho}, \mathbf{x}_i)) \frac{\partial m_1(\mathbf{w} + \varepsilon \boldsymbol{\rho}, \mathbf{x}_i)}{\partial \varepsilon}
\end{aligned}
$$

We can plug in the function $m_1$ and the first derivation of $m_1$ to $\varepsilon$.

$$
\frac{\partial m_1(\mathbf{w} + \varepsilon \boldsymbol{\rho}, \mathbf{x}_i)}{\partial \varepsilon} = \frac{\partial (\mathbf{w} + \varepsilon \boldsymbol{\rho})^T \mathbf{x}_i}{\partial \varepsilon} = \frac{\partial (\mathbf{w}^T \mathbf{x}_i + \varepsilon \boldsymbol{\rho}^T \mathbf{x}_i)}{\partial \varepsilon} = \boldsymbol{\rho}^T \mathbf{x}_i
$$

The second derivation of $MSE^*(\varepsilon)$ is:

$$
\begin{aligned}
h_{MSE^*}(\varepsilon) &= \frac{\partial^2 MSE^*(\varepsilon)}{\partial \varepsilon^2} \\
&= -\frac{2}{N} \frac{\partial \sum_{i=1}^{N} (y_i - m_1(\mathbf{w} + \varepsilon \boldsymbol{\rho}, \mathbf{x}_i)) \boldsymbol{\rho}^T \mathbf{x}_i}{\partial \varepsilon} \\
&= -\frac{2}{N} \frac{\partial \sum_{i=1}^{N} (y_i - (\mathbf{w}^T \mathbf{x}_i + \varepsilon \boldsymbol{\rho}^T \mathbf{x}_i)) \boldsymbol{\rho}^T \mathbf{x}_i}{\partial \varepsilon} \\
&= \frac{2}{N} \sum_{i=1}^{N} (\boldsymbol{\rho}^T \mathbf{x}_i)^2
\end{aligned}
$$

[35]:
```
%%file grad_eps.m
function grad_eps = grad_eps(eps, ws, rho, m, X, Y)
    N = length(X);
    grad_eps = 0;
    for i=1:N
        xi = X(i,:);
        yi = Y(i);
        tmp = yi - m(ws+eps*rho, xi);
        grad_eps = grad_eps + tmp*(rho'*xi');
    end
    grad_eps = -2/N*grad_eps;
end
```

```
Created file '/Users/wlomsi/Documents/ProjekteUni/Vorlesungen/ML
4DV660+4DV661+4DV652/Public ML Notebooks/grad_eps.m'.
```

[36]:
```
stochastic_grad_eps = @(eps, ws, rho, randices)(grad_eps(eps, ws, rho, m1,␣
 ↪XX(randices,:), Y(randices)));
```

[37]:
```
%%file grad2_eps.m
function grad2_eps = grad2_eps(rho, X)
    N = length(X);
    grad2_eps = 0;
    for i=1:N
        xi = X(i,:);
        tmp = rho'*xi';
        grad2_eps = grad2_eps + tmp^2;
    end
    grad2_eps = 2/N*grad2_eps;
end
```

```
Created file '/Users/wlomsi/Documents/ProjekteUni/Vorlesungen/ML
4DV660+4DV661+4DV652/Public ML Notebooks/grad2_eps.m'.
```

[38]:
```
stochastic_grad2_eps = @(rho, randices)(grad2_eps(rho, XX(randices,:)));
```

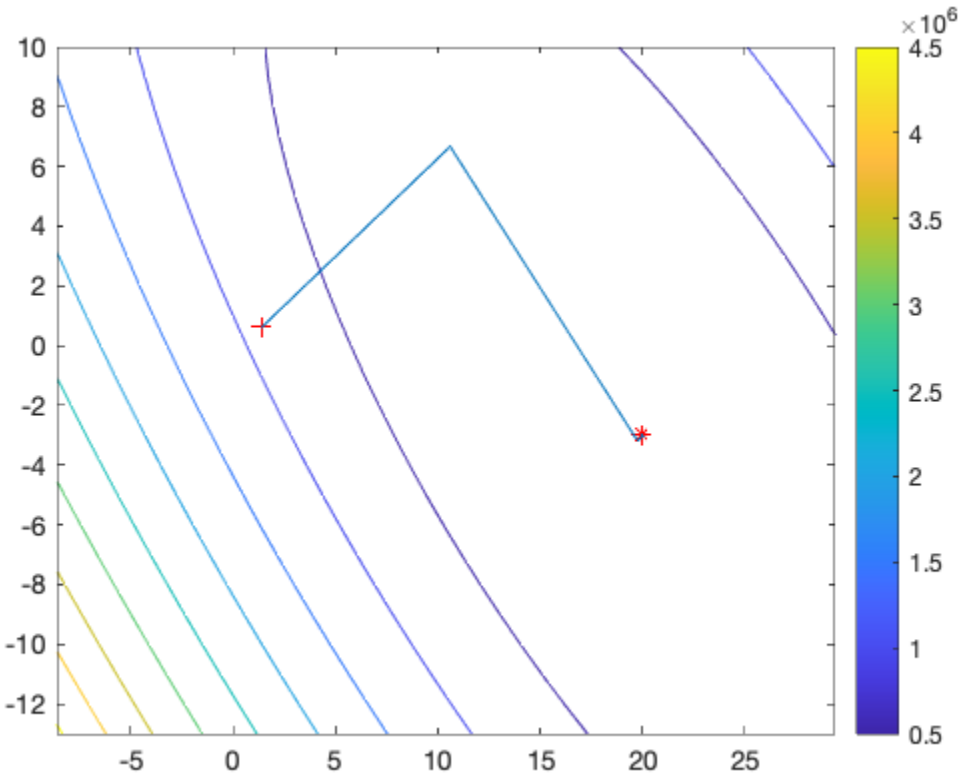
[39]:
```
%%file cg_mse_newton.m
function [ws, history] = cg_mse_newton(K, ws, loss, grad_loss,␣
↪stochastic_grad_eps, stochastic_grad2_eps, N, verbose)
    %initialization
    batch_size = N*0.01;
    history(1) = loss(ws);
    %we only need variables at t (var) and t-1 (var_old)
    rho_old = zeros(length(ws),1);
    grad_ws_old = zeros(length(ws),1);

    for k = 1:K
        randices = randsample(1:N,batch_size,false);
        %Compute gradient
        grad_ws = grad_loss(ws, randices);
        if k==1
            %Linear cg, i.e., no search direction adjustment in the first round
            rho = - grad_ws;
        else
            %Compute Polak-Ribière
            beta = ((grad_ws - grad_ws_old)' * grad_ws) /(grad_ws_old' *␣
↪grad_ws_old);
            %Compute search direction
            rho = - grad_ws + beta*rho_old;
        end
        %Newton line search for epsilon* = argmin stochastic_loss(ws+epsilon *␣
↪rho)
```

```
        %Just one iteration as the (stochastic) loss function is quadratic in␣
 ↪epsilon
        epsilon_star = 0.5;
        grad_eps = stochastic_grad_eps(epsilon_star, ws, rho, randices);
        h = stochastic_grad2_eps(rho, randices);
        epsilon_star = epsilon_star - grad_eps/h;

        %Remember parameters (for drawing the line of this step if verbose)
        old_ws = ws;
        %Update the parameters
        ws= old_ws + epsilon_star * rho;
        %Remember the variables at t as the old variables at t-1 in the next␣
 ↪iteration
        rho_old = rho;
        grad_ws_old = grad_ws;
        %Draw the line
        if verbose
            line([old_ws(1),ws(1)],[old_ws(2),ws(2)]);
        end
        history(k+1) = loss(ws);
    end
end
```

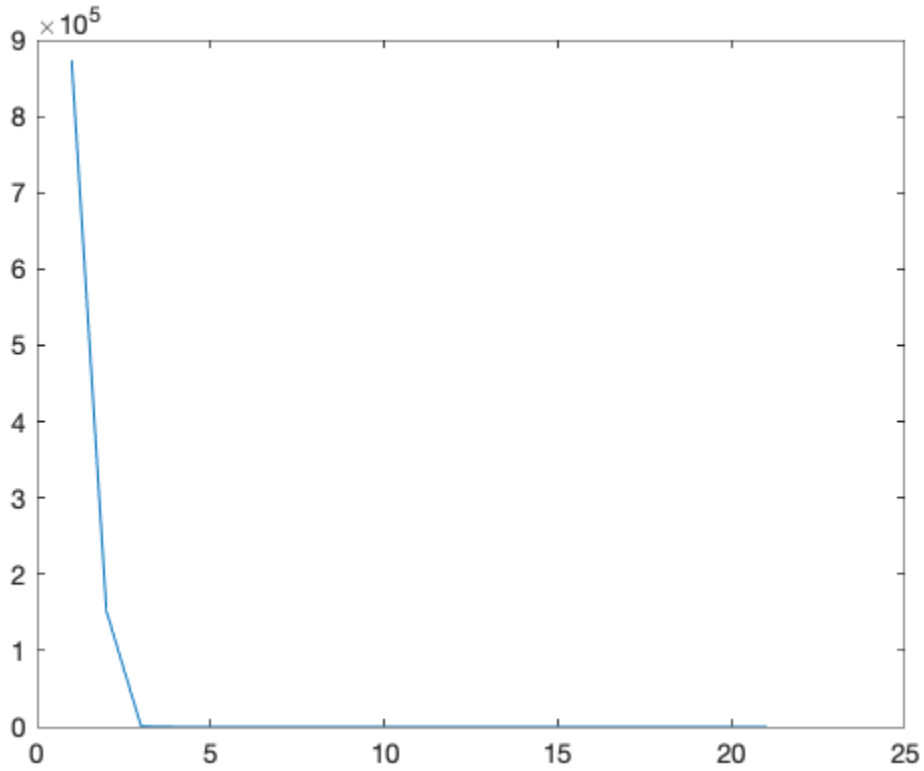
```
Created file '/Users/wlomsi/Documents/ProjekteUni/Vorlesungen/ML
4DV660+4DV661+4DV652/Public ML Notebooks/cg_mse_newton.m'.
```

[40]:
```
%% plot search space
plot3d(f, A, B, false) %3D contour
hold on
plot(a10,a20,'*r')
plot(ws0(1),ws0(2),'+r')

%% CG w Newton's method
K=20;
tic;
[ws, history] = cg_mse_newton(K, ws0, mse1, grad_loss2, stochastic_grad_eps,␣
 ↪stochastic_grad2_eps, N*N, true);
toc
```

```
Elapsed time is 0.061431 seconds.
```




[41]:
```
plot(1:length(history),history)
```


×10^5
9
8
7
6
5
4
3
2
1
0
0
5
10
15
20
25