In [ ]:

```
%matplotlib inline
```

In [1]:

```
import matplotlib.pyplot as plt
import autograd.numpy as np
from sympy import *

from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LogNorm
import matplotlib.colors as mcolors
from matplotlib import animation
from IPython.display import HTML

from autograd import elementwise_grad, value_and_grad
from scipy.optimize import minimize
from collections import defaultdict
from itertools import zip_longest
from functools import partial

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

Test funkcija

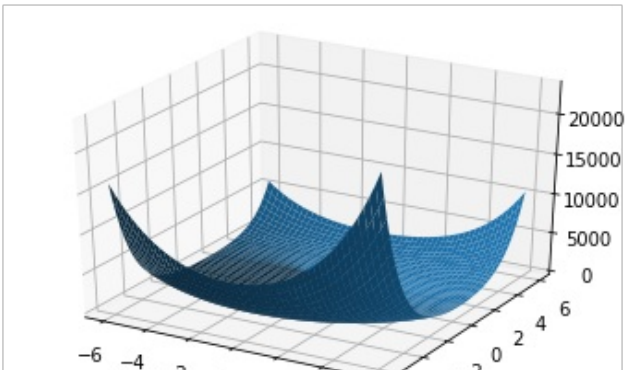
$$f = 0.01((x-1)^2+2(y-1)^2)\ ((x+1)^2+2(y+1)^2+0.5)\ ((x+2)^2+2(y-2)^2+0.7)$$

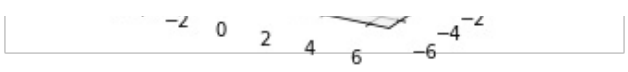
In [138]:

```
def testFunction(x):
  # x - vektor promjenljivih
  # x[0] = x, x[1] = y
  val = 0.01*((x[0]-1)**2+2 *(x[1]-1)**2)* ((x[0]+1)**2+2 *(x[1]+1)**2+0.5)* ((x[0]+2)*
*2+2* (x[1]-2)**2+0.7)

  return val
```

In [139]:

```
x1v = np.arange(-6, 6, 0.01)
x2v = np.arange(-6, 6, 0.01)
x1, x2 = np.meshgrid(x1v, x2v)
z = np.zeros((len(x1),len(x2)))

for i in range(0,len(x1),1):
    for j in range(0,len(x1),1):
        z[i,j]=testFunction((x1[i,j], x2[i,j]))

fig = plt.figure()
ax = fig.gca(projection='3d')
p1 = ax.plot_surface(x1, x2, z)
plt.show()
```





Izvod funkcije

In [140]:

```
x = Symbol('x')
y = Symbol('y')
f =  0.01*((x-1)**2 + 2*(y-1)**2) * ((x+1)**2 + 2*(y+1)**2 + 0.5)* ((x+2)**2 + 2*(y-2)*
*2 + 0.7)
xprime = f.diff(x)
yprime = f.diff(y)
print("Izvod po x", xprime)
print("Izvod po y", yprime)
```

```
Izvod po x (0.02*x - 0.02)*((x + 1)**2 + 2*(y + 1)**2 + 0.5)*((x + 2)**2 + 2*(y - 2)**2 +
0.7) + (2*x + 2)*(0.01*(x - 1)**2 + 0.02*(y - 1)**2)*((x + 2)**2 + 2*(y - 2)**2 + 0.7) +
(2*x + 4)*(0.01*(x - 1)**2 + 0.02*(y - 1)**2)*((x + 1)**2 + 2*(y + 1)**2 + 0.5)
Izvod po y (0.04*y - 0.04)*((x + 1)**2 + 2*(y + 1)**2 + 0.5)*((x + 2)**2 + 2*(y - 2)**2 +
0.7) + (4*y - 8)*(0.01*(x - 1)**2 + 0.02*(y - 1)**2)*((x + 1)**2 + 2*(y + 1)**2 + 0.5) +
(4*y + 4)*(0.01*(x - 1)**2 + 0.02*(y - 1)**2)*((x + 2)**2 + 2*(y - 2)**2 + 0.7)
```

In [141]:

```
def gradFunc(x):
    diffX = (0.02*x[0] - 0.02)*((x[0] + 1)**2 + 2*(x[1] + 1)**2 + 0.5)*((x[0] + 2)**2 +
2*(x[1] - 2)**2 + 0.7) + (2*x[0] + 2)*(0.01*(x[0] - 1)**2 + 0.02*(x[1] - 1)**2)*((x[0]
+ 2)**2 + 2*(x[1] - 2)**2 + 0.7) + (2*x[0] + 4)*(0.01*(x[0] - 1)**2 + 0.02*(x[1] - 1)**
2)*((x[0] + 1)**2 + 2*(x[1] + 1)**2 + 0.5)
    diffY = (0.04*x[1] - 0.04)*((x[0] + 1)**2 + 2*(x[1] + 1)**2 + 0.5)*((x[0] + 2)**2 +
2*(x[1] - 2)**2 + 0.7) + (4*x[1] - 8)*(0.01*(x[0] - 1)**2 + 0.02*(x[1] - 1)**2)*((x[0]
+ 1)**2 + 2*(x[1] + 1)**2 + 0.5) + (4*x[1] + 4)*(0.01*(x[0] - 1)**2 + 0.02*(x[1] - 1)**
2)*((x[0] + 2)**2 + 2*(x[1] - 2)**2 + 0.7)

    x = np.array([[diffX], [diffY]]).reshape(len(x), 1)
    return  x
```

Metod najbržeg pada

Metoda koja nam omogućava da nađemo minimum funkcije $f(x) = f(x_1,$ međutim pod uslovom da je $x_2, \ldots, x_n)$ diferencijabilna. Postupak traženja minimuma se zasniva na tome da u svakoj iteraciji tekuće rešenje pomera u pravcu negativnog gradijenta od funkcije f, pri čemu se gradijent funkcije (pravac najbržeg rasta) definiše

$$\nabla f = [\frac{\partial f}{\partial x_1} \frac{\partial f}{\partial x_2} \cdot \cdot \cdot \frac{\partial f}{\partial x_n}]$$

Osnovni koraci metoda su:

1. **Inicijalizacija.** Odabir početne tačke tj. početnog rešenja x_0, dužine koraka $\gamma > 0$ i toleranicije (kriterijum zaustavljanja algoritma) $\epsilon > 0$, i alternativni kriterijum zaustavljanja, maksimalan broj koraka N.
2. **Algoritam svake iteracije.** U k-toj iteraciji se pomeramo prema sledećem rešenju,

$$x_{k+1} = x_k - \gamma \nabla f(x_k)$$

1. **Kriterijum zaustavljanja** Na kraju svake iteracije proveravamo uslov da li je

$$||\nabla f(x_k)|| \leq \epsilon$$

ili provjeravmo drugi uslov kojim se izbjegava beskonačno izvršavanje iteracije

$k > N$

In [142]:

```
def steepest_descent(gradf, x0,gamma,epsilon, N):

  x=np.array(x0).reshape(len(x0),1)

  path = [] #ovo je samo da bih cuvala istoriju kretanja
  path.append([]) #
  path.append([])#
  path[0].append(x[0].tolist())#
  path[1].append(x[1].tolist())#

  for k in range(N):
    g=gradf(x)
    x=x-gamma*g
    path[0].append(x[0].tolist())#
    path[1].append(x[1].tolist())#
    if np.linalg.norm(g)<epsilon:
      break

  return x,path
```

In [145]:

```
rez,pathSG =steepest_descent(lambda x:gradFunc(x), [-1, 1], 0.15, 1e-4, 100) #0.05, 0.15
, 0.19, 0.202, 0.6
# [1 ,2], [0, 0]
print(rez)
```

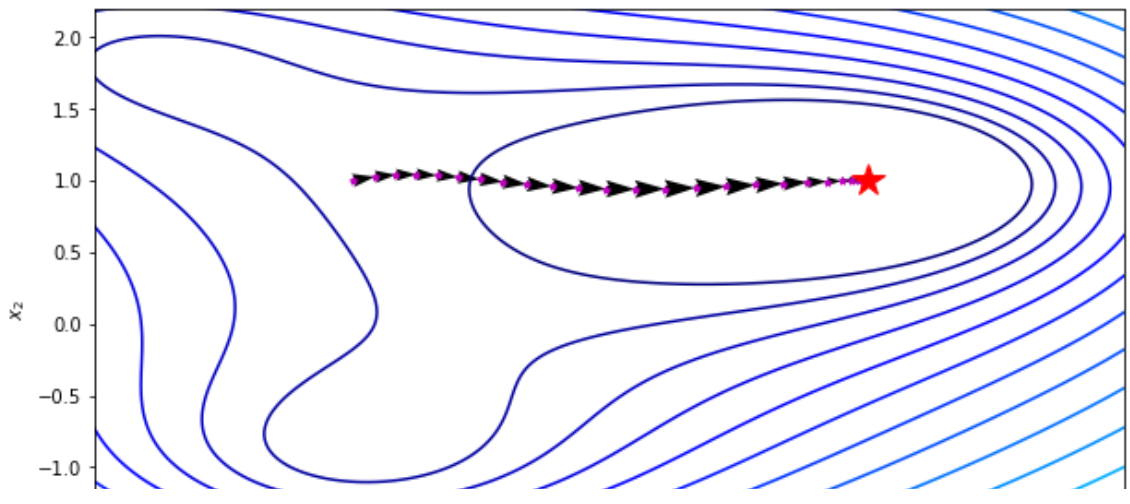
```
[[0.9999876]
 [1.       ]]
```

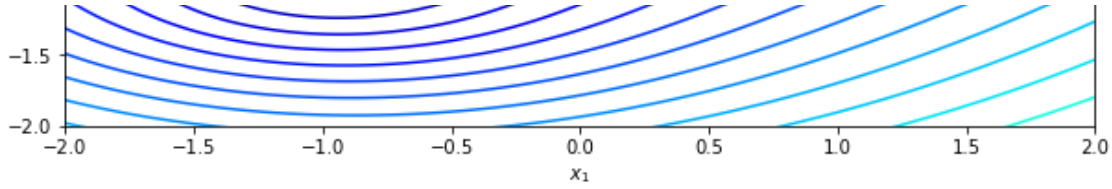
In [146]:

```
xmin, xmax, xstep = -2, 2, .05
ymin, ymax, ystep = -2, 2, .05

fig, ax = plt.subplots(figsize=(10, 6))

ax.contour(x1, x2, z, levels=np.logspace(0, 5, 35), norm=LogNorm(), cmap=plt.cm.jet)
path=np.copy(pathSG)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='k')

ax.plot(path[0][:-1], path[1][:-1], 'm*', markersize=4)
ax.plot(rez[0],rez[1], 'r*', markersize=18)

ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_xlim((xmin, xmax))
ax.set_ylim((ymin, ymax+0.2))
```

Out[146]:

```
(-2.0, 2.2)
```





Ideja bi bila da se ne menja dužina koraka $x_{k+1} = x_k - \frac{\gamma}{||\nabla f(x_k)||}\nabla f(x_k)$

In [147]:

```
def steepest_descent_fix(gradf, x0,gamma,epsilon, N):

  x=np.array(x0).reshape(len(x0),1)

  path = []
  path.append([])
  path.append([])
  path[0].append(x[0].tolist())
  path[1].append(x[1].tolist())

  for k in range(N):
    g=gradf(x)
    x=x-gamma/np.linalg.norm(g)*g
    path[0].append(x[0].tolist())
    path[1].append(x[1].tolist())
    if np.linalg.norm(g)<epsilon:
      break

  return x,path
```

In [150]:

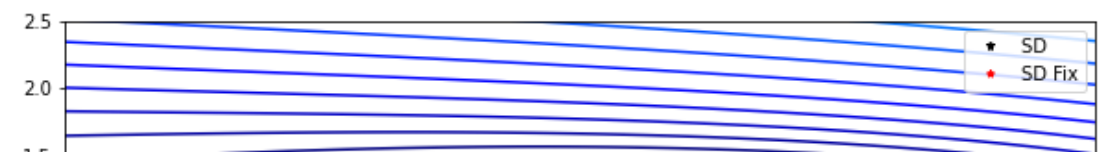
```
rez,pathFix=steepest_descent_fix(lambda x:gradFunc(x), [-1 ,1],0.15,1e-4, 100)
```

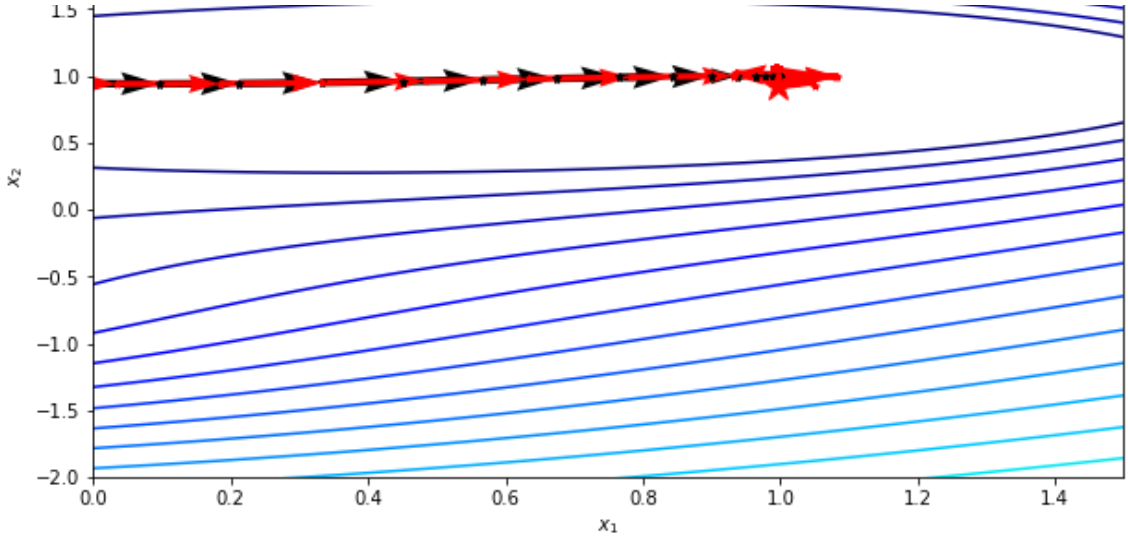
In [151]:

```
fig, ax = plt.subplots(figsize=(10, 6))

ax.contour(x1, x2, z, levels=np.logspace(0, 5, 35), norm=LogNorm(), cmap=plt.cm.jet)

path=np.copy(pathSG)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='k')
ax.plot(path[0][:-1], path[1][:-1], 'k*', markersize=5)
path=np.copy(pathFix)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='r')
ax.plot(path[0][:-1], path[1][:-1], 'r*', markersize=4)
plt.legend(['SD ', 'SD Fix'])

ax.plot(rez[0],rez[1], 'r*', markersize=18)

ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_xlim((xmin+2, xmax-0.5))
ax.set_ylim((ymin, ymax+0.5))
```

Out[151]:

```
(-2.0, 2.5)
```





Dobijena rešenja nisu najsrećnija, zbog toga prelazimo na druge modifikacije.

Osnovne modifikacije metode najbržeg pada

Gradijentna metoda sa momentom

Kod gradijentnog pravila sa momentom pomeranje tačke rešenja ne zavisi samo od tekućeg gradijenta nego kako se pomjerala tačka u prethodnom koraku, tj od prethodne "brzine" kretanja

$$v_k = \omega v_{k-1}$$
$$+ \gamma \nabla f(x_k)$$
$$x_{k+1} = x_k - v_k$$

In [152]:

```
def steepest_descent_momentum(gradf, x0,gamma,epsilon,omega, N):

  xp=np.array(x0).reshape(len(x0),1)
  v=np.zeros(shape=xp.shape)

  path = []
  path.append([])
  path.append([])
  path[0].append(xp[0].tolist())
  path[1].append(xp[1].tolist())

  for k in range(N):
    g=gradf(xp)
    v=omega*v+gamma*g
    x=xp-v
    xp=np.copy(x)
    path[0].append(x[0].tolist())
    path[1].append(x[1].tolist())
    if np.linalg.norm(g)<epsilon:
      break

  return x,path
```

In [153]:

```
rez,pathMSD=steepest_descent_momentum(lambda x:gradFunc(x), [-1, 1],0.15,1e-4,0.1, 100)
#gamma 0.15*0.1, 0.05, omega=0.15*0.9,  0.5
rez
```

Out[153]:

```
array([[0.99999211],
       [1.        ]])
```

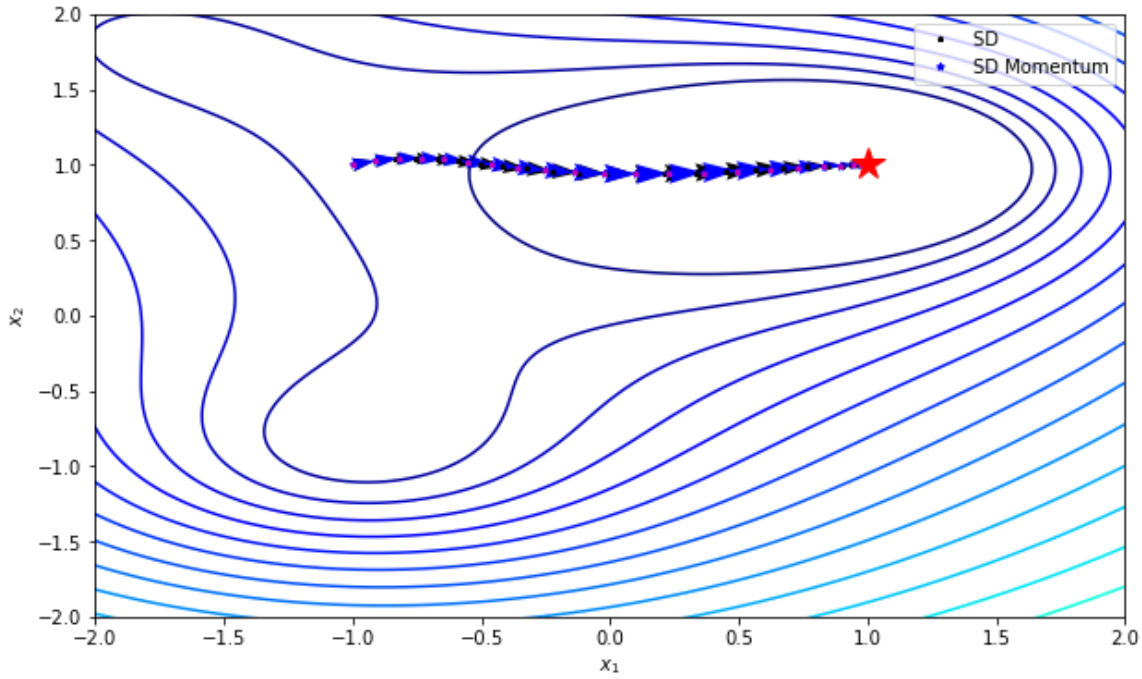
In [154]:

In [154]:

```
fig, ax = plt.subplots(figsize=(10, 6))

ax.contour(x1, x2, z, levels=np.logspace(0, 5, 35), norm=LogNorm(), cmap=plt.cm.jet)


path=np.copy(pathSG)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='k')
ax.plot(path[0][:-1], path[1][:-1], 'k*', markersize=3)
path=np.copy(pathMSD)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='b')
ax.plot(path[0][:-1], path[1][:-1], 'b*', markersize=4)
plt.legend(['SD ', 'SD Momentum'])

ax.plot(path[0][:-1], path[1][:-1], 'm*', markersize=3)
ax.plot(rez[0],rez[1], 'r*', markersize=18)

ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_xlim((xmin, xmax))
ax.set_ylim((ymin, ymax))
```

Out[154]:

```
(-2.0, 2.0)
```



Ubrzani gradijent Nestorova

Kod gradijentnog pravila sa momentom pomeranje tačke rešenja ne zavisi samo od tekućeg gradijenta nego kako se pomjerala tačka u prethodnom koraku,

$$
\begin{aligned}
x'_k &= x_{k-1} - \omega v_{k-1} \\
v_k &= \omega v_{k-1} \\
&\quad + \gamma \nabla f(x'_k) \\
x_{k+1} &= x_k - v_k
\end{aligned}
$$

Jedina razlika u odnosu na gradijentni algoritam sa momentom je ta što gradijent računamo u "predviđenoj budućoj" tački x'_k, i onda tu vrijednost koristimo dalje. Time uvodimo predikciju ponašanja u algoritam.

In [155]:

```
def nesterov_gradient(gradf, x0,gamma,epsilon,omega, N):

  xp=np.array(x0).reshape(len(x0),1)
```

```
  v=np.zeros(shape=xp.shape)

  path = []
  path.append([])
  path.append([])
  path[0].append(xp[0].tolist())
  path[1].append(xp[1].tolist())

  for k in range(N):
    xest=xp-omega*v
    g=gradf(xest)
    v=omega*v+gamma*g
    x=xp-v
    xp=np.copy(x)
    path[0].append(x[0].tolist())
    path[1].append(x[1].tolist())
    if np.linalg.norm(g)<epsilon:
      break

  return x,path
```

In [156]:

```
rez,pathNG=nesterov_gradient(lambda x:gradFunc(x), [-1 ,1],0.15,1e-4,0.1, 100)
rez
```

Out[156]:

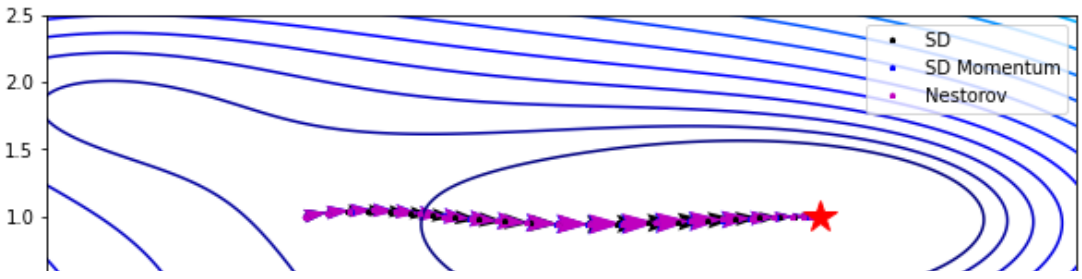
```
array([[0.99998937],
       [1.        ]])
```

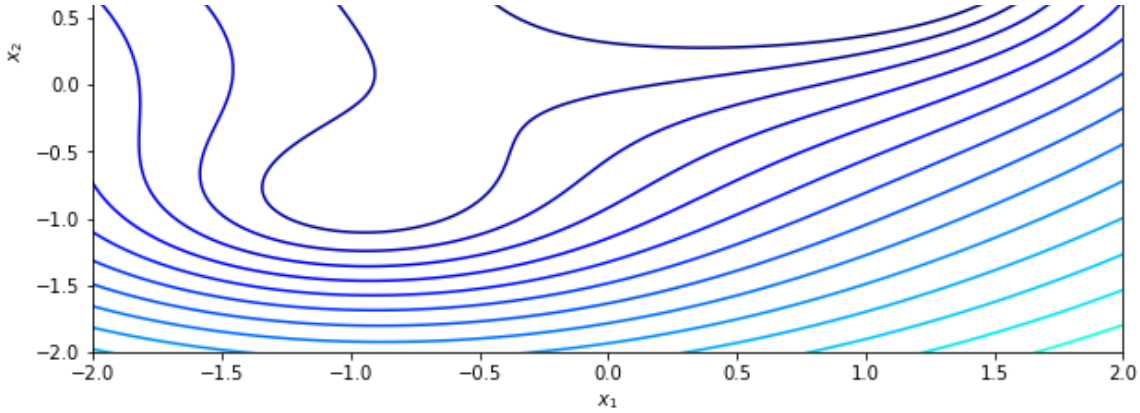
In [157]:

```
fig, ax = plt.subplots(figsize=(10, 6))
ax.contour(x1, x2, z, levels=np.logspace(0, 5, 35), norm=LogNorm(), cmap=plt.cm.jet)
path=np.copy(pathSG)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='k')
ax.plot(path[0][:-1], path[1][:-1], 'k*', markersize=3)
path=np.copy(pathMSD)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='b')
ax.plot(path[0][:-1], path[1][:-1], 'b*', markersize=3)

path=np.copy(pathNG)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='m')
ax.plot(path[0][:-1], path[1][:-1], 'm*', markersize=3)


plt.legend(['SD ', 'SD Momentum','Nestorov'])

ax.plot(path[0][:-1], path[1][:-1], 'm*', markersize=3)
ax.plot(rez[0],rez[1], 'r*', markersize=18)

ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_xlim((xmin, xmax))
ax.set_ylim((ymin, ymax+0.5))
```

Out[157]:

```
(-2.0, 2.5)
```





Adaptivni gradijentni metodi

Osnovni problem sa svim do sada razmatranim algoritmima leži u tome što je brzina adaptacije ista po svim osama. To znači da su ovi algoritmi neefikasni (teško je podesiti parametre) u slučajevima kada se kriterijum znatno brže menja po jednoj osi nego po drugoj. Ukoliko mala promena neke promenljive dovodi do velikih promena kriterijuma optimalnosti, tada tu promenljivu treba menjati polako, sa malim koracima. Nasuprot tome, ukoliko male promene neke promenljive dovode do neznatnih promena kriterijuma optimalnosti, tada te promenljive treba menjati brzo, sa velikim koracima. Drugačije rečeno: brzina adaptacije treba da bude različita za svaku osu!

ADAGRAD

Adagard koristi adaptivni gradijent, specifičan za svaku osu (svaku promenljivu).

Neka je $g_{k,i}$ gradijent kriterijuma optimalnosti po i-toj promenljivoj u k-toj iteraciji

$$g_{k,i} = \nabla f(x_k)_i = \frac{\partial f(x_k)}{\partial x_i}$$

Označimo sa $G_{k,i}$ sumu kvadrata gradijenata po i-toj promenljivo, zaključno sa k-tom iteracijom

$$G_{k,i} = \sum_{j=0}^{k} g_{j,i}^2$$

ADAGRAD algoritam podrazumeva sledeći mehanizam ažuriranja tekuće pozicije

$$x_{k+1,i} = x_{k,i} - \frac{\gamma}{\sqrt{G_{k,i} + \epsilon_1}} g_{k,i}$$

gde je ϵ_1 mali parametar koji služi za regularizaciju izraza u početnim iteracijama $\epsilon_1 \sim 10^{-8}$

In [158]:

```
def adagard_gradient(gradf, x0,gamma,epsilon1,epsilon, N):

  xp=np.array(x0).reshape(len(x0),1)
  v=np.zeros(shape=xp.shape)
  G=np.zeros(shape=xp.shape)
  path = []
  path.append([])
  path.append([])
  path[0].append(xp[0].tolist())
  path[1].append(xp[1].tolist())

  for k in range(N):

    g=gradf(xp)
    G=G+np.multiply(g,g)
    v=gamma*np.ones(shape=G.shape)/np.sqrt(G+epsilon1)*g
    x=xp-v
```

```
    xp=np.copy(x)
    path[0].append(x[0].tolist())
    path[1].append(x[1].tolist())
    if np.linalg.norm(g)<epsilon:
      break

  return x,path
```

In [159]:

```
rez,pathADAGARD=adagard_gradient(lambda x:gradFunc(x), [-1,1],0.3,1e-6,1e-6, 100)
rez
```

Out[159]:

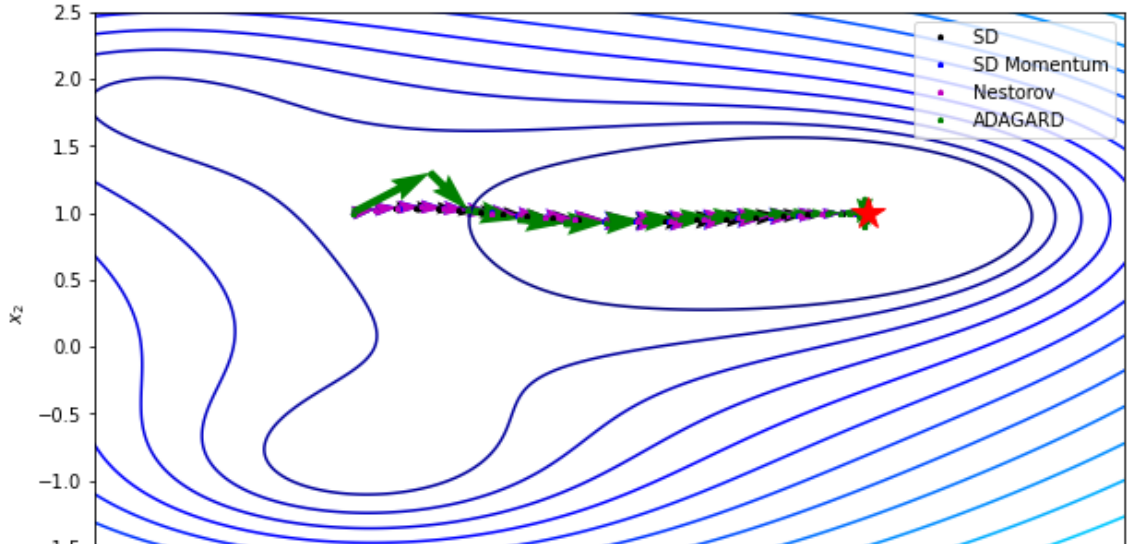
```
array([[0.99999982],
       [1.        ]])
```

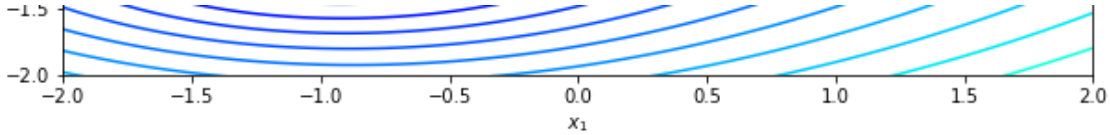
In [160]:

```
fig, ax = plt.subplots(figsize=(10, 6))

ax.contour(x1, x2, z, levels=np.logspace(0, 5, 35), norm=LogNorm(), cmap=plt.cm.jet)
path=np.copy(pathSG)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='k')
ax.plot(path[0][:-1], path[1][:-1], 'k*', markersize=3)
path=np.copy(pathMSD)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='b')
ax.plot(path[0][:-1], path[1][:-1], 'b*', markersize=3)

path=np.copy(pathNG)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='m')
ax.plot(path[0][:-1], path[1][:-1], 'm*', markersize=3)
path=np.copy(pathADAGARD)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='g')
ax.plot(path[0][:-1], path[1][:-1], 'g*', markersize=3)

plt.legend(['SD ', 'SD Momentum','Nestorov','ADAGARD'])

#ax.plot(path[0][:-1], path[1][:-1], 'm*', markersize=3)
ax.plot(rez[0],rez[1], 'r*', markersize=18)

ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_xlim((xmin, xmax))
ax.set_ylim((ymin, ymax+0.5))
```

Out[160]:

```
(-2.0, 2.5)
```





RMSProp

RMSProp algoritam sličan ADAGRAD, ali se kvadrati gradijenta ne akumuliraju neograničeno

$$G_{k+1,i} = \omega G_{k,j} + (1-\omega) g_{k,i}^2$$

Tipična vrednost ω je 0.9.

Kako algoritam konvergira, pretpostavimo da g^2 postaje konstantno, vrednost G u ustaljenom stanju će biti

$$G = \omega G + (1-\omega) g^2$$

tj. $G = g^2$.

In [161]:

```
def rmsprop_gradient(gradf, x0,gamma,omega,epsilon1,epsilon, N):

  xp=np.array(x0).reshape(len(x0),1)
  v=np.zeros(shape=xp.shape)
  G=np.zeros(shape=xp.shape)
  path = []
  path.append([])
  path.append([])
  path[0].append(xp[0].tolist())
  path[1].append(xp[1].tolist())

  for k in range(N):
    g=gradf(xp)
    G=omega*G+(1-omega)*np.multiply(g,g)
    v=gamma*np.ones(shape=G.shape)/np.sqrt(G+epsilon1)*g
    x=xp-v
    xp=np.copy(x)
    path[0].append(x[0].tolist())
    path[1].append(x[1].tolist())
    if np.linalg.norm(g)<epsilon:
      break

  return x,path
```

In [162]:

```
rez5,pathRMS=rmsprop_gradient(lambda x:gradFunc(x), [1 ,2],0.05, 0.9,1e-6,1e-6, 100)
rez5
```

Out[162]:

```
array([[1.        ],
       [1.00000003]])
```

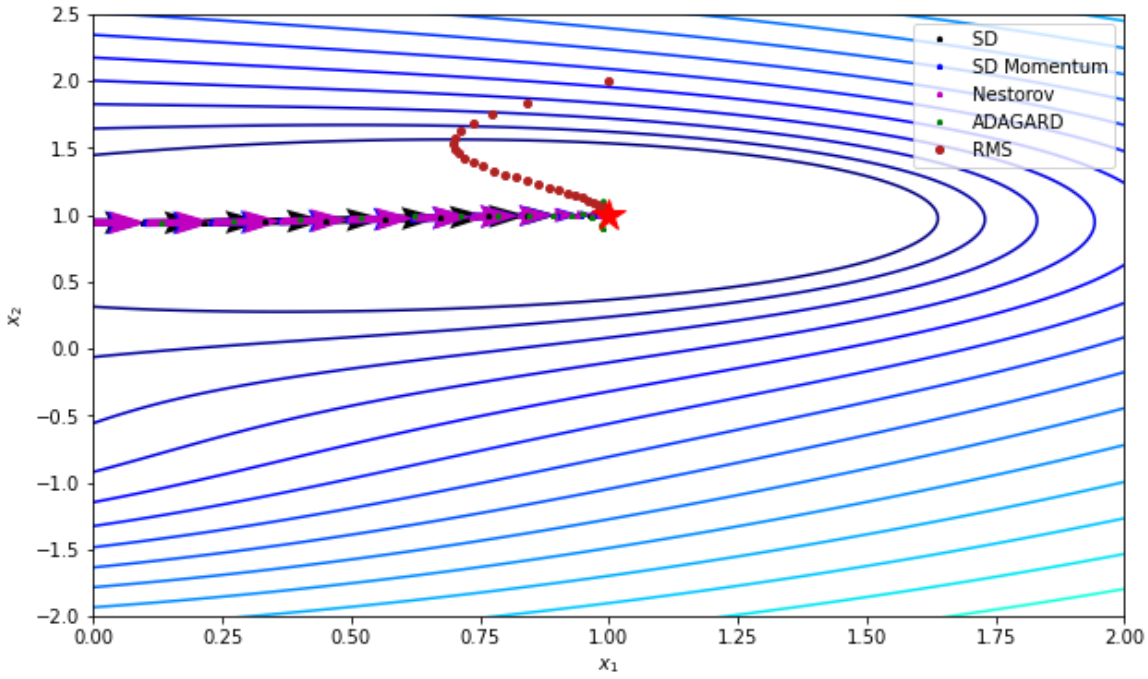
In [163]:

```
fig, ax = plt.subplots(figsize=(10, 6))

ax.contour(x1, x2, z, levels=np.logspace(0, 5, 35), norm=LogNorm(), cmap=plt.cm.jet)
path=np.copy(pathSG)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='k')
ax.plot(path[0][:-1], path[1][:-1], 'k*', markersize=3)

path=np.copy(pathMSD)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='b')
```

```
ax.plot(path[0][:-1], path[1][:-1], 'b*', markersize=3)

path=np.copy(pathNG)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='m')
ax.plot(path[0][:-1], path[1][:-1], 'm*', markersize=3)

path=np.copy(pathADAGARD)
#ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='g')
ax.plot(path[0][:-1], path[1][:-1], 'g*', markersize=3)

path=np.copy(pathRMS)
#ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='firebrick')
ax.plot(path[0][:-1], path[1][:-1], marker='o',color='firebrick', markersize=4,linestyle
= 'None')

plt.legend(['SD ', 'SD Momentum','Nestorov','ADAGARD','RMS'])
ax.plot(rez[0],rez[1], 'r*', markersize=18)

ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_xlim((xmin+2, xmax))
ax.set_ylim((ymin, ymax+0.5))
```

Out[163]:

```
(-2.0, 2.5)
```



ADAM

ADAM (ADAPTIV MOMENT ESTIMATION) je jedna od najčešćih savremenih modifikacija algoritma najstrmijeg pada.

Za pomeranje ka boljem rešenju koristi se pomoćne veličine

$$m_k = \omega_1 m_{k-1} + (1-\omega_1) g_k$$
$$v_k = \omega_2 v_{k-1} + (1-\omega_2) g_k^2$$

te njihove korigovane verzije

$$\hat{m}_k = \frac{m_k}{1-\omega_1}$$

$$\hat{v}_k = \frac{v_k}{1-\omega_2}$$

Potom se tekuće rešenje ažurira po algoritmu

$$x_{k+1} = x_k - \frac{\gamma}{\sqrt{\hat{v}_k + \epsilon_1}}\hat{m}_k$$

In [164]:

```
def adam_gradient(gradf, x0,gamma,omega1,omega2, epsilon1,epsilon, N):

  xp=np.array(x0).reshape(len(x0),1)
  v=np.zeros(shape=xp.shape)
  m=np.zeros(shape=xp.shape)

  path = []
  path.append([])
  path.append([])
  path[0].append(xp[0].tolist())
  path[1].append(xp[1].tolist())

  for k in range(N):

    g=gradf(xp)

    m=omega1*m+(1-omega1)*g
    v=omega2*v+(1-omega2)*np.multiply(g,g)
    hat_m=m/(1-omega1)
    hat_v=v/(1-omega2)

    x=xp-gamma*np.ones(shape=g.shape)/np.sqrt(hat_v+epsilon1)*hat_m
    xp=np.copy(x)
    path[0].append(x[0].tolist())
    path[1].append(x[1].tolist())
    if np.linalg.norm(g)<epsilon:
      break

  return x,path
```

In [165]:

```
rez,pathADAM=adam_gradient(lambda x:gradFunc(x), [1 ,2],0.1, 0.1,0.9,1e-6,1e-6, 100)
rez
```

Out[165]:

```
array([[1.        ],
       [1.00000001]])
```

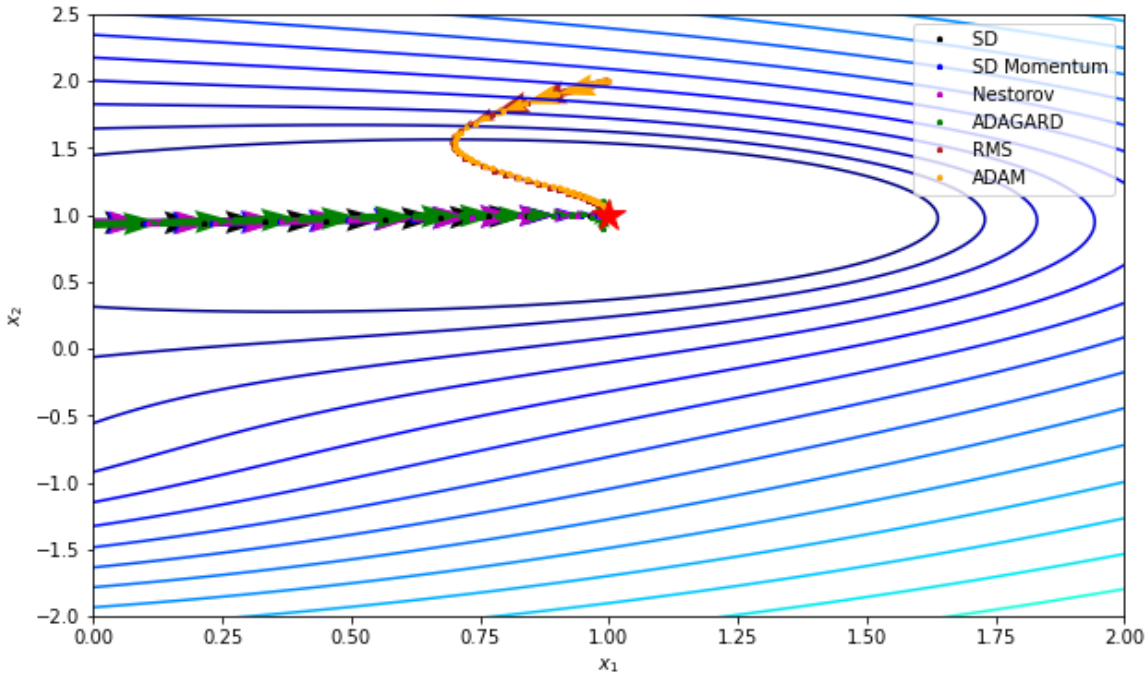
In [166]:

```
fig, ax = plt.subplots(figsize=(10, 6))

ax.contour(x1, x2, z, levels=np.logspace(0, 5, 35), norm=LogNorm(), cmap=plt.cm.jet)
path=np.copy(pathSG)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='k')
ax.plot(path[0][:-1], path[1][:-1], 'k*', markersize=3)

path=np.copy(pathMSD)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='b')
ax.plot(path[0][:-1], path[1][:-1], 'b*', markersize=3)

path=np.copy(pathNG)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='m')
ax.plot(path[0][:-1], path[1][:-1], 'm*', markersize=3)
```

```
path=np.copy(pathADAGARD)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='g')
ax.plot(path[0][:-1], path[1][:-1], 'g*', markersize=3)

path=np.copy(pathRMS)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='firebrick')
ax.plot(path[0][:-1], path[1][:-1], marker='*',color='firebrick', markersize=3,linestyle
= 'None')

path=np.copy(pathADAM)
ax.quiver(path[0][:-1], path[1][:-1], np.subtract(path[0][1:],path[0][:-1]),np.subtract(
path[1][1:],path[1][:-1]), scale_units='xy', angles='xy', scale=1, color='orange')
ax.plot(path[0][:-1], path[1][:-1], marker='*',color='orange', markersize=3,linestyle =
'None')

plt.legend(['SD ', 'SD Momentum','Nestorov','ADAGARD','RMS','ADAM'])
ax.plot(rez[0],rez[1], 'r*', markersize=18)

ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_xlim((xmin+2, xmax))
ax.set_ylim((ymin, ymax+0.5))
```

Out[166]:

```
(-2.0, 2.5)
```



f = x^2 + 10y^2

- **osnovni gradijentni metod**
- **gradijentni sa momentumom**
- **Nestorov**
- **2 adaptivna po izboru**