In [2]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LogNorm
from matplotlib import animation
from scipy.optimize import minimize, OptimizeResult

from answer import Answer
```

Implementation (Students do)

Methods

You will implement four optimization algorithms (descriptions available here (https://ruder.io/optimizing-gradient-descent/index.html)). For algorithms that keep a moving average (all but gradient descent), you should initialize the averaging variables to zeros. Keep in mind that we will be trying to use these algorithms to minimize, not maximize, an objective.

- Gradient descent (`gd`)
- Momentum gradient method (`momentum`)
- Nesterov's accelerated gradient method (`nag`)
- Adaptive gradient method (`adagrad`)

Make note of the function headers: `def gd(func, x, lr, num_iters, jac, tol, callback, *args, **kwargs):`. Each method will satisfy this header format in accordance with the specification of custom minimizers used with `scipy.optimize.minimize`. This function is well-documented (https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html), but the highlights of the arguments are below.

- `func`: [type: function] The loss function. Takes in a point of type np.ndarray (2,) and returns a float representing the value of the function at that point.
- `x`: [type: np.ndarray (2,)] The starting point of the optimization.
- `lr`: [type: function] Learning rate schedule. Takes in an argument of type int representing the iteration number, and returns the learning rate to be used for that iteration.
- `num_iters`: [type: int] The number of iterations of the optimization method to run.
- `jac`: [type: function] The gradient of the loss function. "jac" stands for Jacobian, which is out of scope for this class, but for scalar-valued functions, it is the transpose of the gradient. Takes in a point of type np.ndarray (2,) and returns an np.ndarray (2,) representing the gradient of the function at that point.
- `tol`: [type: float] The tolerance (with respect to the iterate) within which the optimization method is deemed to have converged.
- `callback`: [type: function] A function to be called on each iterate over the course of the optimization.
- `*args` and `**kwargs`: You will not need to use these, but they are present for compatibility with the `scipy.optimize.minimize` API.

Each function will need to return a two-tuple containing

- An instance of `scipy.optimize.OptimizeResult`, described here (https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.OptimizeResult.html).
- A `np.ndarray` containing the function value at the initial point and each iterate over the course of the optimization.

In [3]:

```python
# demonstration of scipy.optimize.minimize and the usage of the callback argument
minimize(lambda x: (x ** 2, 2 * x), 3, jac=True, method='Newton-CG', callback=print)
```

```
[0.]
[0.]
```

Out[3]:

```
     fun: array([0.])
     jac: array([0.])
 message: 'Optimization terminated successfully.'
    nfev: 3
    nhev: 0
     nit: 2
    njev: 6
  status: 0
 success: True
       x: array([0.])
```

In [4]:

```
def gd(func, x, lr, num_iters, jac, tol, callback, *args, **kwargs):
    losses = []
    x_per = 0
    losses.append(func(x))
    bol = False
    for itr in range(num_iters):
        # TODO implement:
        #   - gradient descent update for x
        #   - call `callback` on each iterate of x (excluding the initial value)
        #   - add function values to the `losses` list
        #   - stop the algorithm after `num_iters` iterations or if the iterates converge
        #     within a tolerance of `tol`. `np.linalg.norm` may be useful for this part.
        if abs(np.linalg.norm(x_per) - np.linalg.norm(x)) < tol:
            bol = True
            break
        else:
            x_per = x
            x = x_per - lr(itr) * jac(x_per)
            callback(x)
            losses.append(func(x))
    # TODO return an OptimizeResult object (documentation linked above) and a numpy array of the
losses above
    return OptimizeResult(x = x,success = bol,status = 0,nit = itr), np.array(losses)

def momentum(func, x, lr, num_iters, jac, tol, callback, alpha=0.9, *args, **kwargs):
    # TODO implement
    losses = []
    x_per = 0
    velocity = 0
    bol = False
    losses.append(func(x))
    for itr in range(num_iters):
        if abs(np.linalg.norm(x_per) - np.linalg.norm(x)) < tol:
            bol = True
            break
        else:
            x_per = x
            velocity = -alpha*velocity + lr(itr)*jac(x_per)
            x = x_per - velocity
            callback(x)
            losses.append(func(x))
    return OptimizeResult(x = x,success = bol,status = 0, nit = itr), np.array(losses)

def nag(func, x, lr, num_iters, jac, tol, callback, alpha=0.9, *args, **kwargs):
    # TODO implement
    losses = []
    x_per = 0
    velocity = 0
    bol = False
    losses.append(func(x))
    for itr in range(num_iters):
        if abs(np.linalg.norm(x_per) - np.linalg.norm(x)) < tol:
            bol = True
            break
        else:
            x_per = x
            velocity = -alpha*velocity + lr(itr)*jac(x_per + alpha*velocity)
            x = x_per - velocity
            callback(x)
            losses.append(func(x))
            OptimizeResult(x = x,success = bol,status = 0, nit = itr), np.array(losses)

     adagrad(func, x, lr, num_iters, jac, tol, callback, eps=1e-5, *args, **kwargs):

    losses = []
    x_per = 0
    sumval = 0
    bol =
    losses.append(func(x))
         itr    range(num_iters):
           abs(np.linalg.norm(x_per) - np.linalg.norm(x)) < tol:
            bol =

             :
            x_per = x
            sumval += jac(x_per)**2
            x = x_per - lr(itr)/np.sqrt(sumval + eps)*jac(x_per)
            callback(x)
            losses.append(func(x))
            OptimizeResult(x = x,success = bol,status = 0, nit = itr), np.array(losses)
```

Functions and gradients

Below you have been given the implementation of four functions ($\mathbb{R}^2 \to \mathbb{R}$). You will need to implement `grad`, which returns their gradients as `np.ndarray` (2,). There is a field below for you to submit the gradient in $\LaTeX$.

- Booth function: $f_1(x) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$
- Beale function: $f_2(x) = (1.5 - x_1 + x_1x_2)^2 + \left(2.25 - x_1 + x_1x_2^2\right)^2 + \left(2.625 - x_1 + x_1x_2^3\right)^2$
- Rosenbrock function: $f_3(x) = 100 \cdot \left(x_2 - x_1^2\right)^2 + (x_1 - 1)^2$
- Ackley function: $f_4(x) = -20 \cdot \exp\left(-\frac{1}{5}\sqrt{\frac{x_1^2+x_2^2}{2}}\right) - \exp\left(\frac{\cos 2\pi x_1 + \cos 2\pi x_2}{2}\right) + 20 + \exp(1)$

In [5]:

```
def func(fn, x1, x2):
    if fn == 'booth':
        return (x1 + 2*x2 - 7)**2 + (2*x1 + x2 - 5)**2
    elif fn == 'beale':
        return (1.5 - x1 + x1*x2)**2 + (2.25 - x1 + x1*x2**2)**2 + (2.625 - x1 + x1*x2**3)**2
    elif fn == 'rosen2d':
        return 100 * (x2 - x1**2)**2 + (x1 - 1)**2
    elif fn == 'ackley2d':
        return -20 * np.exp(-0.2 * np.sqrt((x1**2 + x2**2)/2)) - np.exp((np.cos(2*np.pi*x1) + n
p.cos(2*np.pi*x2))/2) + 20 + np.e
    else:
        raise ValueError('Function %s not supported.' % fn)

def grad(fn, x1, x2):
    # TODO for each function, place the two elements of the gradient in g1 and g2 respectively
    if fn == 'booth':
        g1 = 2*(x1 + 2*x2 - 7.0) + 2*(2*x1 + x2 - 5.0)*2
        g2 = 2*(x1 + 2*x2 - 7.0)*2 + 2*(2*x1 + x2 - 5.0)
    elif fn == 'beale':
        g1 = 2*(-1 + x2)*(1.5 - x1 + x1*x2) + 2*(-1+x2**2)*(2.25 - x1 + x1*x2**2) + 2*(-1 + x2**
3)*(2.625 - x1 + x1*x2**3)
        g2 = 2*x1*(1.5 - x1 + x1*x2) + 2*2*x1*x2*(2.25 - x1 + x1*x2**2) + 2*3*x1*x2**2*(2.625 -
x1 + x1*x2**3)
    elif fn == 'rosen2d':
        g1 = 100*2*(-2*x1)*(x2 - x1**2) + 2*(x1 - 1)
        g2 = 200*(x2 - x1**2)
    elif fn == 'ackley2d':
        g1 = 4*x1*(1/np.sqrt(2*(x1**2+x2**2)))*np.exp(-0.2 * np.sqrt((x1**2 + x2**2)/2)) + np.pi
*np.sin(2*np.pi*x1)*np.exp((np.cos(2*np.pi*x1) + np.cos(2*np.pi*x2))/2)
        g2 = 4*x2*(1/np.sqrt(2*(x1**2+x2**2)))*np.exp(-0.2 * np.sqrt((x1**2 + x2**2)/2)) + np.pi
*np.sin(2*np.pi*x2)*np.exp((np.cos(2*np.pi*x1) + np.cos(2*np.pi*x2))/2)
    else:
        raise ValueError('Function %s not supported.' % fn)
    return np.stack((g1, g2), axis=-1)
```

Submission: Gradient values $\LaTeX$

Enter the gradients you calculated below.

- Booth: $\nabla_x f_1(x) = \begin{bmatrix} 2\left(x_1 + 2x_2 - 7\right) + 4\left(2x_1 + x_2 - 5\right) \\ 4\left(x_1 + 2x_2 - 7\right) + 2\left(2x_1 + x_2 - 5\right) \end{bmatrix}$

  \
- Beale:

  $\nabla_x f_2(x) = \Big[\begin{matrix} 2\left(-1 + x_2\right)\left(1.5 - x_1 + x_1x_2\right) + 2\left(-1 + x_2^2\right)\left(2.25 - x_1 + x_1x_2^2\right) + 2\left(-1 + x \right. \\ 2x_1\left(1.5 - x_1 + x_1x_2\right) + 4x_1x_2\left(2.25 - x_1 + x_1x_2^2\right) + 6x_1x_2^2\left(2.625 - \right. \end{matrix}$

  \
- Rosenbrock: $\nabla_x f_3(x) = \begin{bmatrix} -400x_1\left(x_2 - x_1^2\right) + 2\left(x_1 - 1\right) \\ 200\left(x_2 - x_1^2\right) \end{bmatrix}$

  \
- Ackley: $\nabla_x f_4(x) = \begin{bmatrix} \frac{4x_1}{\sqrt{2\left(x_1^2+x_2^2\right)}} \exp\left(-\frac{1}{5}\sqrt{\frac{x_1^2+x_2^2}{2}}\right) + \pi \sin 2\pi x_1 \exp\left(\frac{\cos 2\pi x_1 + \cos 2\pi x_2}{2}\right) \\ \frac{4x_2}{\sqrt{2\left(x_1^2+x_2^2\right)}} \exp\left(-\frac{1}{5}\sqrt{\frac{x_1^2+x_2^2}{2}}\right) + \pi \sin 2\pi x_2 \exp\left(\frac{\cos 2\pi x_1 + \cos 2\pi x_2}{2}\right) \end{bmatrix}$

Testing your code

We are not providing much structure here, but now is a good time to make sure your optimization methods are working well. The cell below tests your gradient descent method on the function $f(x) = x^2$. We have included the output of our solution as a comment. Note that the function you feed it needs to take in a point as its sole argument and return the function as well as the gradient evaluated at that point.

In [5]:

```
# Maybe a useful starting example for testing gradient descent on a simple function
x_squared = lambda x: (x**2, 2*x)  # returns both the function value and the gradient.
opt_res, losses = minimize(x_squared, 3, jac=True, method=gd, callback=print,
                           options=dict(lr=lambda t: 0.25, x0=np.array([3]), num_iters=15, tol=
1e-3))
print('Final iterate: %.6f. Number of iterations: %d. Final loss: %.8f.' % (opt_res.x, opt_res.
nit, losses[-1]))
# ----------------
# Expected output (GD):
# ----------------
# 1.5
# 0.75
# 0.375
# 0.1875
# 0.09375
# 0.046875
# 0.0234375
# 0.01171875
# 0.005859375
# 0.0029296875
# 0.00146484375
# 0.000732421875
# Final iterate: 0.000732. Number of iterations: 12. Final loss: 0.00000054.
```

```
1.5
0.75
0.375
0.1875
0.09375
0.046875
0.0234375
0.01171875
0.005859375
0.0029296875
0.00146484375
0.000732421875
Final iterate: 0.000732. Number of iterations: 12. Final loss: 0.00000054.
```

In [6]:

```
x_squared = lambda x: (x**2, 2*x)  # returns both the function value and the gradient.
opt_res, losses = minimize(x_squared, 3, jac=True, method=momentum, callback=print,
                           options=dict(lr=lambda t: 0.12, x0=np.array([3]), num_iters=15, tol=
1e-3))
print('Final iterate: %.6f. Number of iterations: %d. Final loss: %.8f.' % (opt_res.x, opt_res.
nit, losses[-1]))
# -----------------
# Expected output (Momentum):
# -----------------
# 2.2800000000000002
# 2.3808000000000002
# 1.7186880000000002
# 1.9021036800000002
# 1.2805246848
# 1.532619856128
# 0.93790543646208
# 1.248051109410509
# 0.6693877374984007
# 1.029531715219682
# 0.4583145236178052
# 0.8624145103912211
# 0.29174503980125377
# 0.7353287537799235
# 0.15962451029193914
# Final iterate: 0.159625. Number of iterations: 15. Final loss: 0.02547998.
```

```
2.2800000000000002
2.3808000000000002
1.7186880000000002
1.9021036800000002
1.2805246848
1.532619856128
0.93790543646208
1.248051109410509
0.6693877374984007
1.029531715219682
0.4583145236178052
0.8624145103912211
0.29174503980125377
0.7353287537799235
0.15962451029193914
Final iterate: 0.159625. Number of iterations: 14. Final loss: 0.02547998.
```

In [7]:

```
x_squared = lambda x: (x**2, 2*x)  # returns both the function value and the gradient.
opt_res, losses = minimize(x_squared, 3, jac=True, method=nag, callback=print,
                           options=dict(lr=lambda t: 0.4, x0=np.array([3]), num_iters=15, tol=1
e-3))
print('Final iterate: %.6f. Number of iterations: %d. Final loss: %.8f.' % (opt_res.x, opt_res.
nit, losses[-1]))
# -----------------
# Expected output (NAG):
# -----------------
# 0.5999999999999996
# 0.552
# 0.11903999999999987
# 0.10174079999999996
# 0.023462015999999974
# 0.01878258431999999
# 0.004598814566399996
# 0.0034728414689279997
# 0.0008972434513305587
# 0.0006430563334336508
# Final iterate: 0.000643. Number of iterations: 10. Final loss: 0.00000041.
```

```
0.5999999999999996
0.552
0.11903999999999987
0.10174079999999996
0.023462015999999974
0.01878258431999999
0.004598814566399996
0.0034728414689279997
0.0008972434513305587
0.0006430563334336508
Final iterate: 0.000643. Number of iterations: 10. Final loss: 0.00000041.
```

In [8]:

```
x_squared = lambda x: (x**2, 2*x)  # returns both the function value and the gradient.
opt_res, losses = minimize(x_squared, 3, jac=True, method=adagrad, callback=print,
                           options=dict(lr=lambda t: 2, x0=np.array([3]), num_iters=15, tol=1e-
3))
print('Final iterate: %.6f. Number of iterations: %d. Final loss: %.8f.' % (opt_res.x, opt_res.
nit, losses[-1]))
# ----------------
# Expected output (Adagrad):
# ----------------
# 1.0000002777777202
# 0.3675446666871427
# 0.13664342397366747
# 0.050879388377382734
# 0.018949088326907643
# 0.007057448115379032
# 0.002628505330734456
# 0.0009789720193835473
# 0.00036461264876190005
# Final iterate: 0.000365. Number of iterations: 9. Final loss: 0.00000013.
```

```
1.0000002777777204
0.3675446666871429
0.13664342397366758
0.050879388377382775
0.018949088326907663
0.0070574481153790405
0.0026285053307344586
0.0009789720193835482
0.0003646126487619004
Final iterate: 0.000365. Number of iterations: 9. Final loss: 0.00000013.
```

Student-facing `Answer` class (provided)

You have been provided a class called `Answer` which will be helpful for the remainder of the project. It can be found in the `answer.py` file. You are welcome to read and modify it, but this is not required. All information you need about this class is documented here, and examples of usage are given below.

Documentation

- `__init__(self, methods, func, grad)`
  - Instantiates the `Answer` class with the functions you have implemented. `methods` is a dictionary mapping algorithm names to the functions that implement them, and `func` and `grad` are the functions of the same name that you have implemented.
- `set_fn_settings(self, fn_name)`
  - Sets the instance variables needed for visualizing `fn_name` with `plot2d` and `plot3d`. Needs to be called before calling these functions.
- `set_settings(self, fn_name, method, x0, **kwargs)`
  - Sets the instance variables needed for visualizing `method` optimizing `fn_name` starting at `x0` with `path2d`, `path3d`, `video2d`, and `video3d`. Any additional `kwargs` (likely `lr` and `num_iters`) will be passed on to `method`. Needs to be called before calling these functions or `compare`.
- `get_settings(self)`
  - Returns the arguments passed into `set_settings`: `fn_name`, `method`, `x0`, and `kwargs`.
- `compare(self, method, start_iter=0, **kwargs)`
  - Generates training loss graph comparing `method` with the previously set method on the previously set loss function and starting point, starting at iteration `start_iter`. Additional `kwargs` (likely `lr` and `num_iters`) will be passed on to `method`.
- `get_xs_losses(self)`
  - Returns a tuple containing
    - [type: `np.ndarray` (1 + `n_iters`, 2)] All iterates (including the initial point).
    - [type: `np.ndarray` (1 + `n_iters`,)] The loss at each iterate.
- `get_min_errs(self)`
  - Returns a tuple containing
    - `float` representing the closest (in L2 norm) the optimization procedure got to the global minimizer.
    - `float` representing the closest the optimization procedure got to the global minimum function value.
- `func_val(self, x)`
  - Returns `float` value of the previously set loss function evaluated at `x`. Convenience tool for debugging.
- `grad_val(self, x)`
  - Returns `np.ndarray` (2,) gradient of the previously set loss function evaluated at `x`. Convenience tool for debugging.
- `plot2d(self)`
  - Plots contours of the previously set loss function.
- `plot3d(self)`
  - Plots the previously set loss function.
- `path2d(self)`
  - Plots the sequence of iterates produced by the set method on the set loss function on a 2D contour.
- `path3d(self)`

  - Plots the sequence of iterates produced by the set method on the set loss function on a 3D graph. **NOTE:** This one does not work very well.
- `video2d(self, filename=None)`
  - Creates and saves an MP4 video of the path taken in `path2d` at `filename`. File name defaults to "{function}_{method}_2d.mp4"
- `video3d(self, filename=None)`
  - Creates and saves an MP4 video of the path taken in `path3d` at `filename`. File name defaults to "{function}_{method}_3d.mp4". **NOTE:** This works better than `path3d`

In [6]:

```
# instantiate the Answer class with the methods you have implemented! (You can implement and add
more if you like!)
ans = Answer(
    {  # a mapping of algorithm names to functions implementing them
        'gd': gd,
        'momentum': momentum,
        'nag': nag,
        'adagrad': adagrad
    },
    func,
    grad
)
```

Playground and Exploration

You are free to use the functions described above to explore the behavior of the optimization algorithms you have implemented. Pick different starting points, learning rate schedules, and even tolerances to explore! Example usage of the `Answer` class is below.

Exploration

For each of the functions, start at the given initial points (x_0) and use any choice of optimization algorithm and associated hyperparameters to get within the specified distance of the global minimizer and minimum (ϵ_x, ϵ_f). *Hint: The `get_min_errs` function will be helpful*. There is a spot below for you to submit your results for each challenge.

- Booth function
  - $x_0 = [8, 9], \epsilon_x = 10^{-7}, \epsilon_f = 10^{-14}$
- Beale function
  - $x_0 = [3, 4], \epsilon_x = 0.5, \epsilon_f = 0.07$
- Rosenbrock function
  - $x_0 = [8, 9], \epsilon_x = 10^{-7}, \epsilon_f = 10^{-14}$
- Ackley function
  - $x_0 = [25, 20]$. This function is challenging to optimize. Tell us in your write-up what approaches you tried and how close you got.

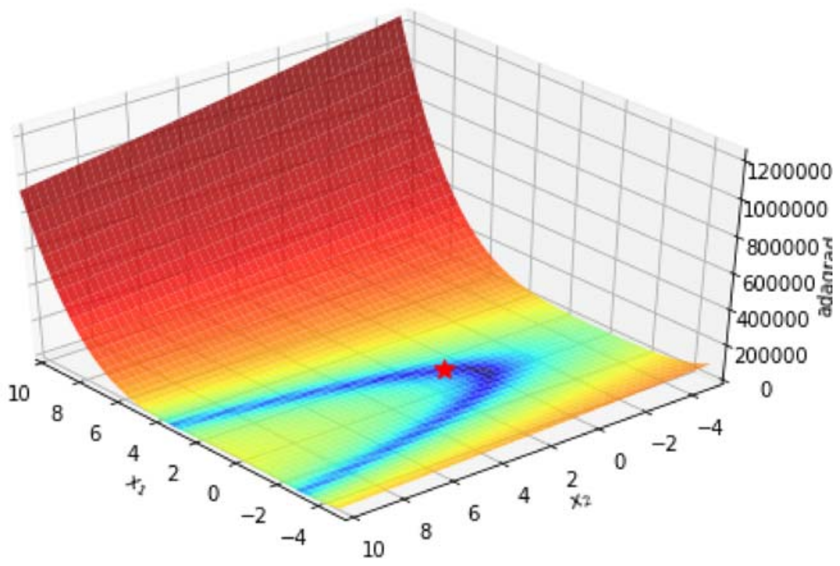
In [10]:

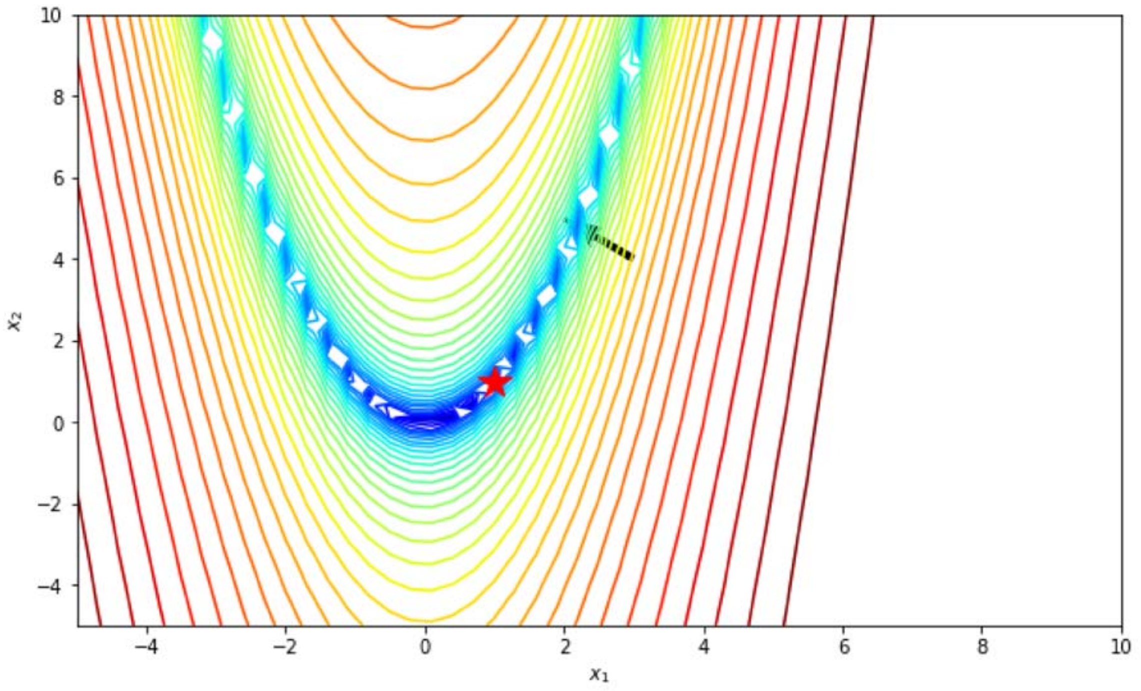
```
def lr_(t):
    if t < 10:
        return 1
    elif t < 20:
        return 1e-3
    else:
        return 1e-5
ans.set_settings(fn_name='rosen2d', method='adagrad', x0=np.array([3, 4]), lr=lr_, num_iters=30,
tol=1e-8)
```

In [11]:

```
# How close the optimization got to the red star, and how far the min loss was from the global m
in.
print(ans.get_min_errs())

# The gradient at the end of the optimization. This can be helpful for tuning your learning rate
schedule!
last_grad = ans.grad_val(ans.get_xs_losses()[0][-1])
print(last_grad)

# Some visuals
ans.plot3d()
ans.path2d()
ans.get_xs_losses()
# ans.video3d()  # This saves a video to the folder this notebook is in!
```

```
(3.605551275463989, 1.3838997168564693)
[0.5608264  0.41163231]
```

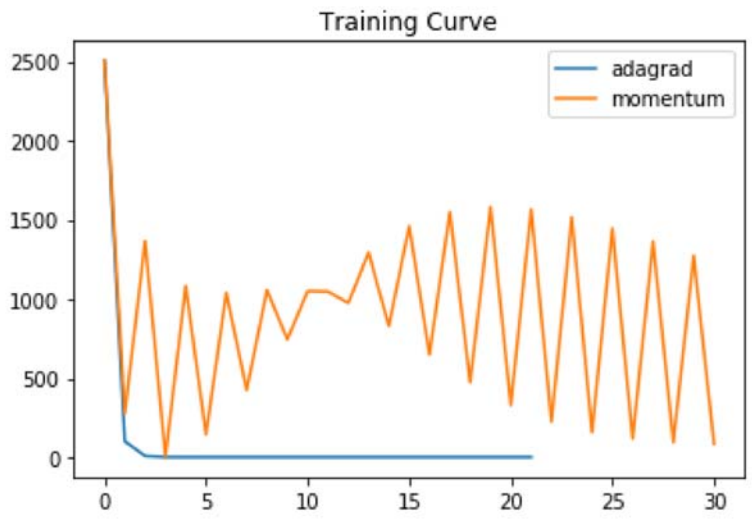




```
Out[11]:

(array([[3.        , 4.        ],
        [2.        , 5.        ],
        [2.13175275, 4.80388386],
        [2.16789087, 4.75305476],
        [2.17513187, 4.74261508],
        [2.17638294, 4.74037915],
        [2.17653143, 4.73964736],
        [2.17648195, 4.73918549],
        [2.17639707, 4.73877189],
        [2.17630587, 4.73836693],
        [2.17621354, 4.73796354],
        [2.17621345, 4.73796314],
        [2.17621336, 4.73796273],
        [2.17621326, 4.73796233],
        [2.17621317, 4.73796193],
        [2.17621308, 4.73796152],
        [2.17621299, 4.73796112],
        [2.17621289, 4.73796072],
        [2.1762128 , 4.73796031],
        [2.17621271, 4.73795991],
        [2.17621262, 4.73795951],
        [2.17621261, 4.7379595 ]]),
 array([2.50400000e+03, 1.01000000e+02, 8.01561952e+00, 1.64809982e+00,
        1.39396843e+00, 1.38527293e+00, 1.38478236e+00, 1.38455575e+00,
        1.38433760e+00, 1.38411973e+00, 1.38390190e+00, 1.38390168e+00,
        1.38390146e+00, 1.38390124e+00, 1.38390103e+00, 1.38390081e+00,
        1.38390059e+00, 1.38390037e+00, 1.38390015e+00, 1.38389994e+00,
        1.38389972e+00, 1.38389972e+00]))
```

In [12]:

```
ans.compare('momentum', lr=lambda t: 1e-4, num_iters=30)
```



```
[Method    adagrad] Final loss: 1.3839, Final x: [2.1762, 4.7380]
[Method   momentum] Final loss: 84.4012, Final x: [2.1531, 3.7245]
```

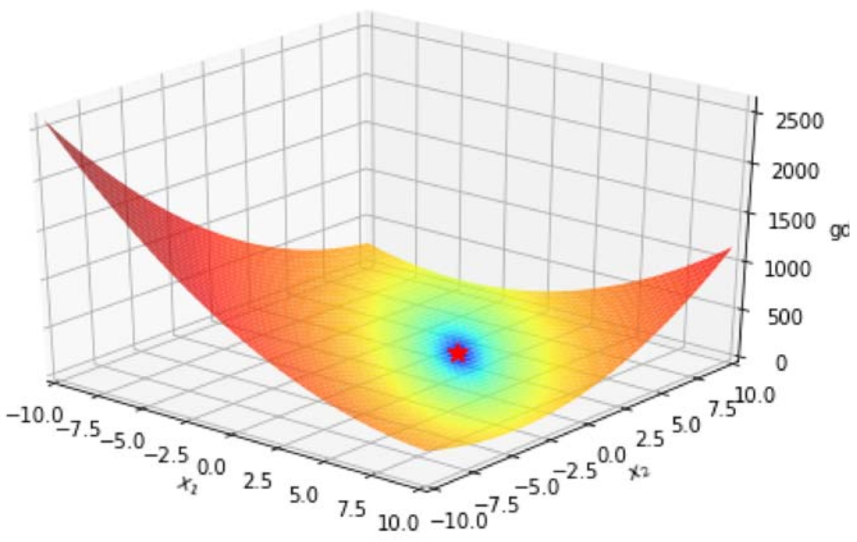
Submission: Challenge

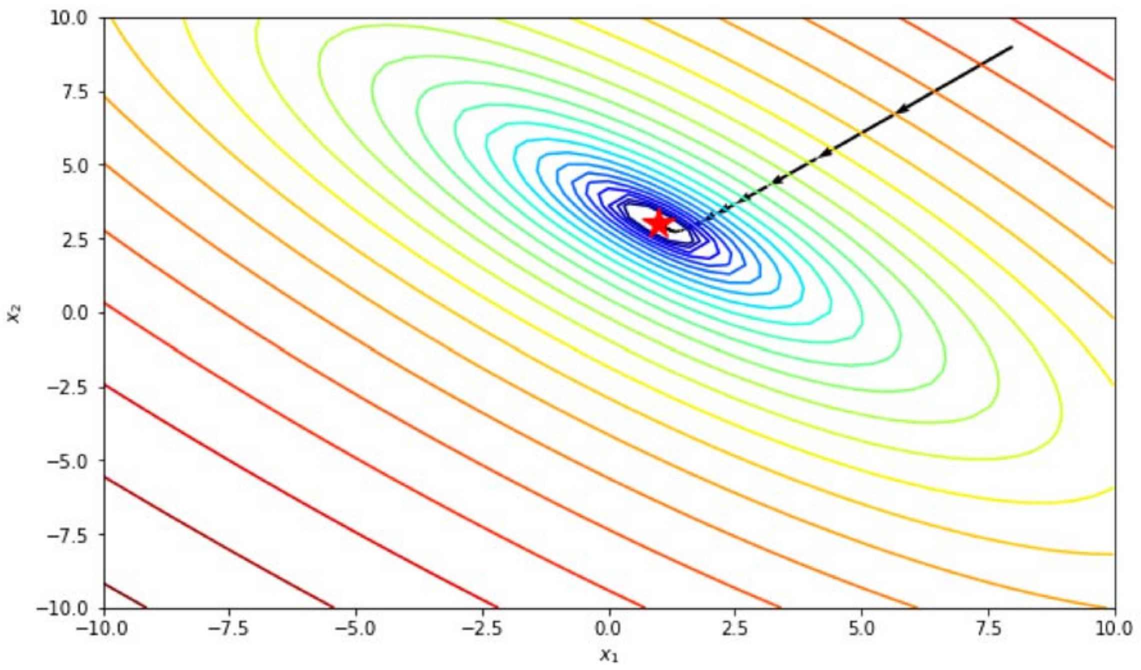
Place code in the below cells that demonstrates your results for each challenge. Each cell should end with `get_min_errs()` displaying the achieved error.

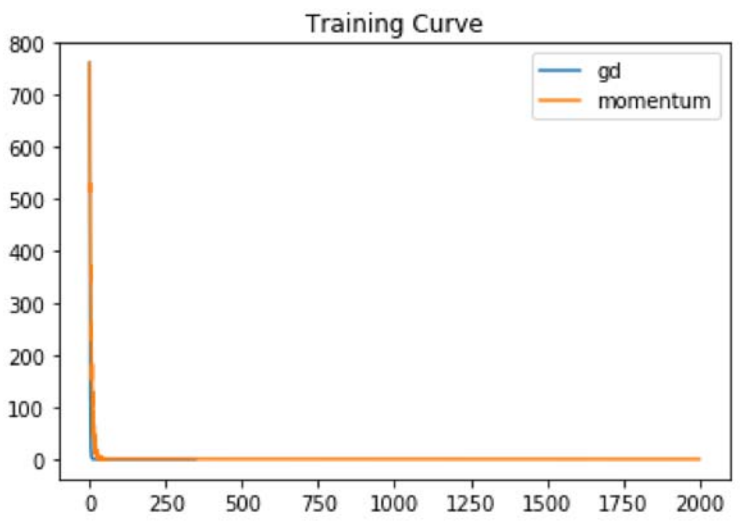
Booth function

In [13]:

```
# Replace parameters here
params = dict(
    method='gd',
    lr= lambda t: 0.02,
    num_iters=2500,
    alpha=0.8
)
#ans.set_settings(fn_name='rosen2d', method='adagrad', x0=np.array([3, 4]), lr=lr_, num_iters=3
0, tol=1e-8)
ans.set_settings(fn_name='booth', x0=np.array([8, 9]), tol=1e-8,**params)
ans.plot3d()
ans.path2d()
ans.compare('momentum', lr=lambda t: 0.01, num_iters=2000)
#ans.compare('nag', lr=lambda t: 0.01, num_iters=200)
#ans.compare('adagrad', lr=lambda t: 0.01, num_iters=200)
ans.get_min_errs()
```







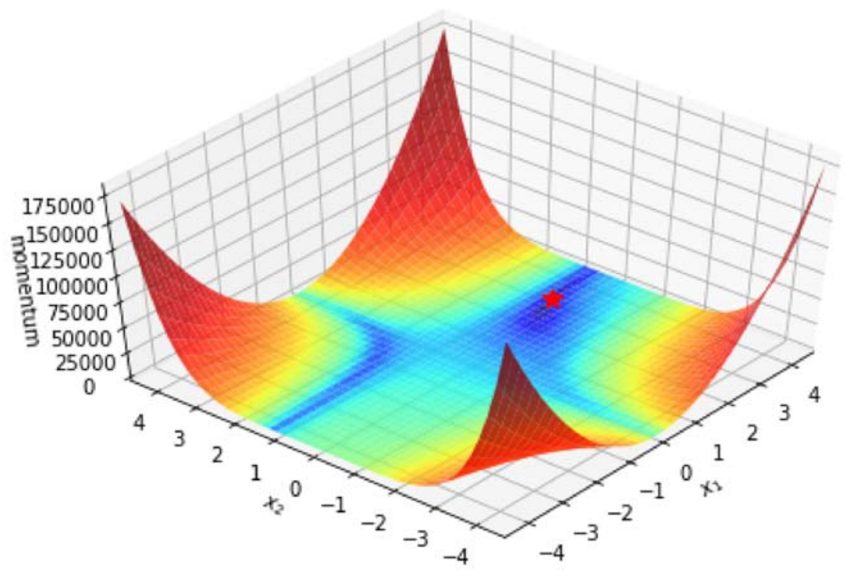
```
[Method          gd] Final loss: 0.0000, Final x: [1.0000, 3.0000]
[Method    momentum] Final loss: 0.0000, Final x: [1.0000, 3.0000]
```

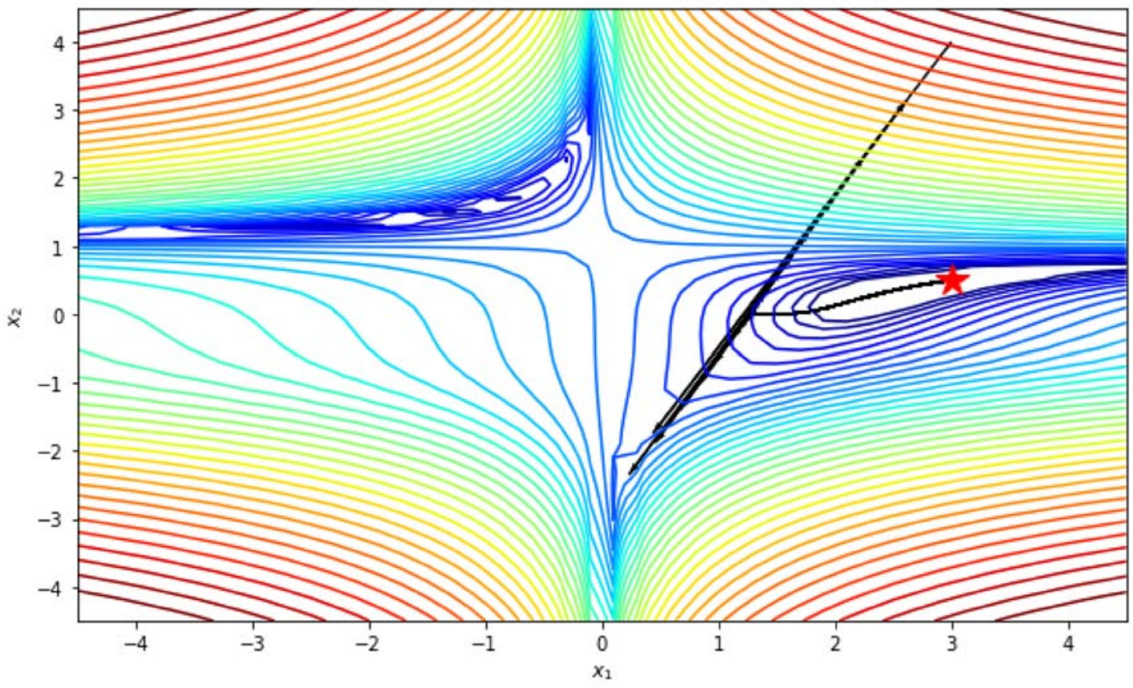
```
Out[13]:

(5.191963541863726e-07, 2.695648542004213e-13)
```

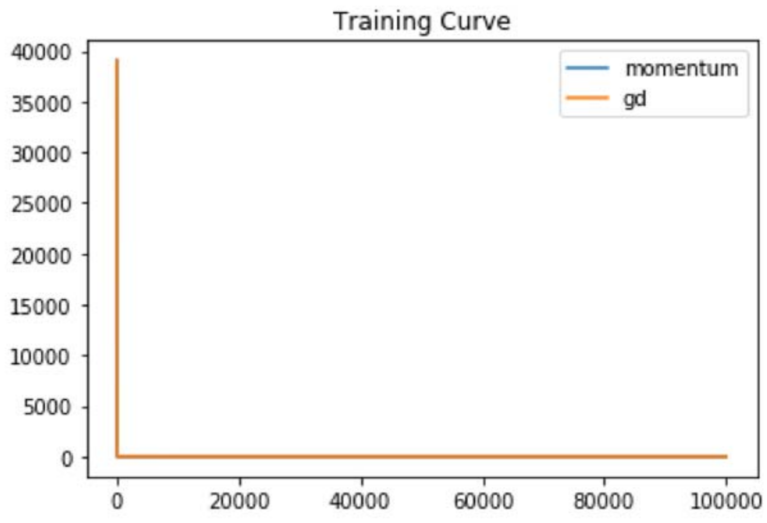
Beale function

In [7]:

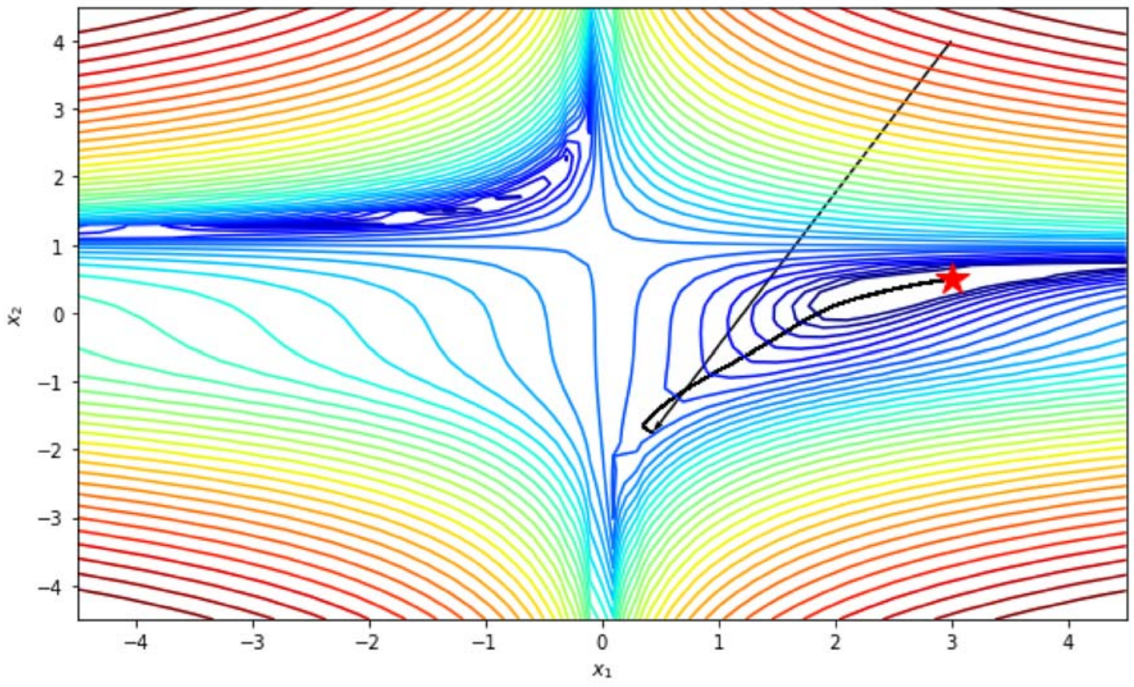
```
# Replace parameters here
#ans.set_settings(fn_name='beale', method='nag', x0=np.array([3, 4]), lr=1e-4, num_iters=1e-5, t
ol=0.5)
params = dict(
    method='momentum',
    lr= lambda t: 1e-4,
    num_iters=100000,
    alpha=0.85
)
ans.set_settings(fn_name='beale', x0=np.array([3, 4]), tol=1e-8,**params)
ans.plot3d()
ans.path2d()
ans.compare('gd', lr=lambda t: 1e-4, num_iters=100000)
#ans.compare('nag', start_iter=0, **params)
#ans.compare('gd', start_iter=0, **params)
ans.get_min_errs()
params = dict(
    method='gd',
    lr= lambda t: 1e-4,
    num_iters=100000,
    alpha=0.85
)
ans.set_settings(fn_name='beale', x0=np.array([3, 4]), tol=1e-8,**params)
ans.path2d()
```


momentum
175000
150000
125000
100000
75000
50000
25000
0
x_2
x_1
4
3
2
1
0
−1
−2
−3
−4


x_2
x_1
4
3
2
1
0
−1
−2
−3
−4



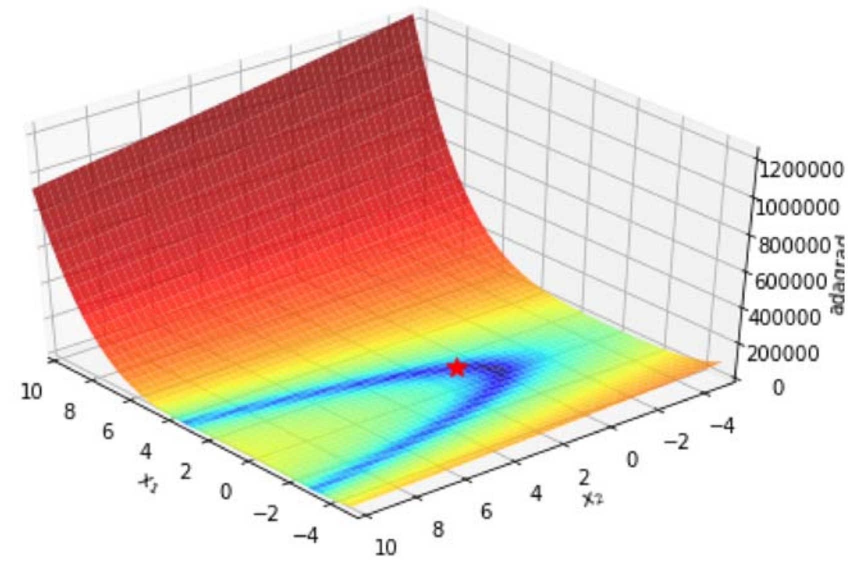
```
[Method   momentum] Final loss: 0.0011, Final x: [2.9223, 0.4799]
[Method         gd] Final loss: 0.0001, Final x: [2.9793, 0.4948]
```

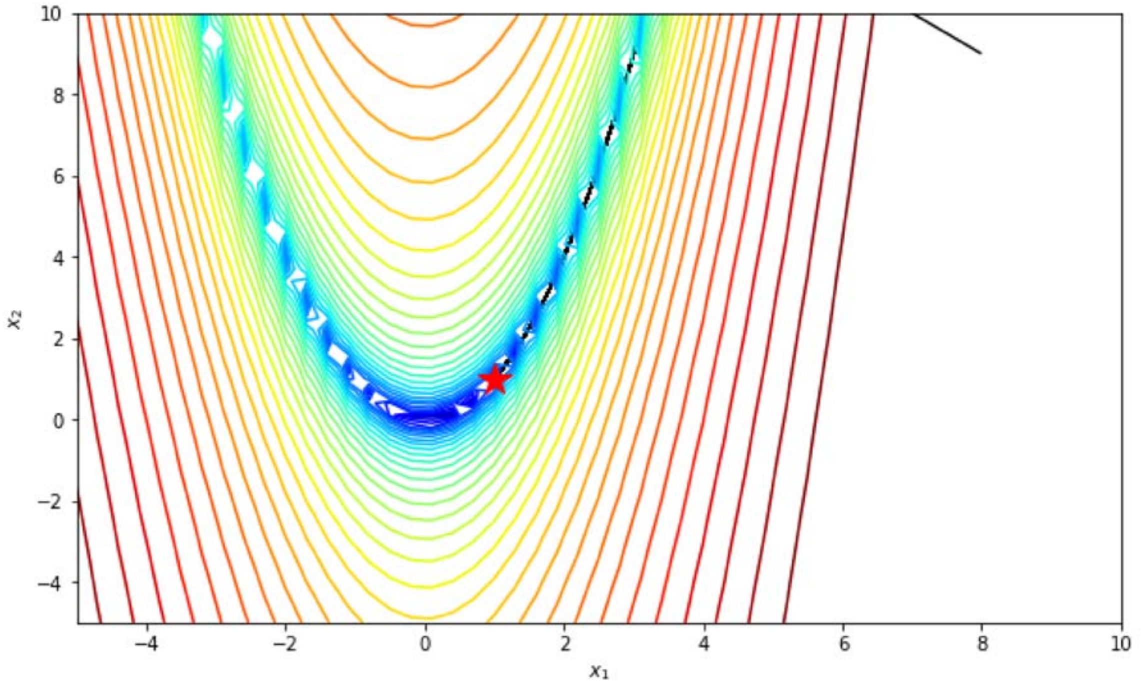


Rosenbrock function

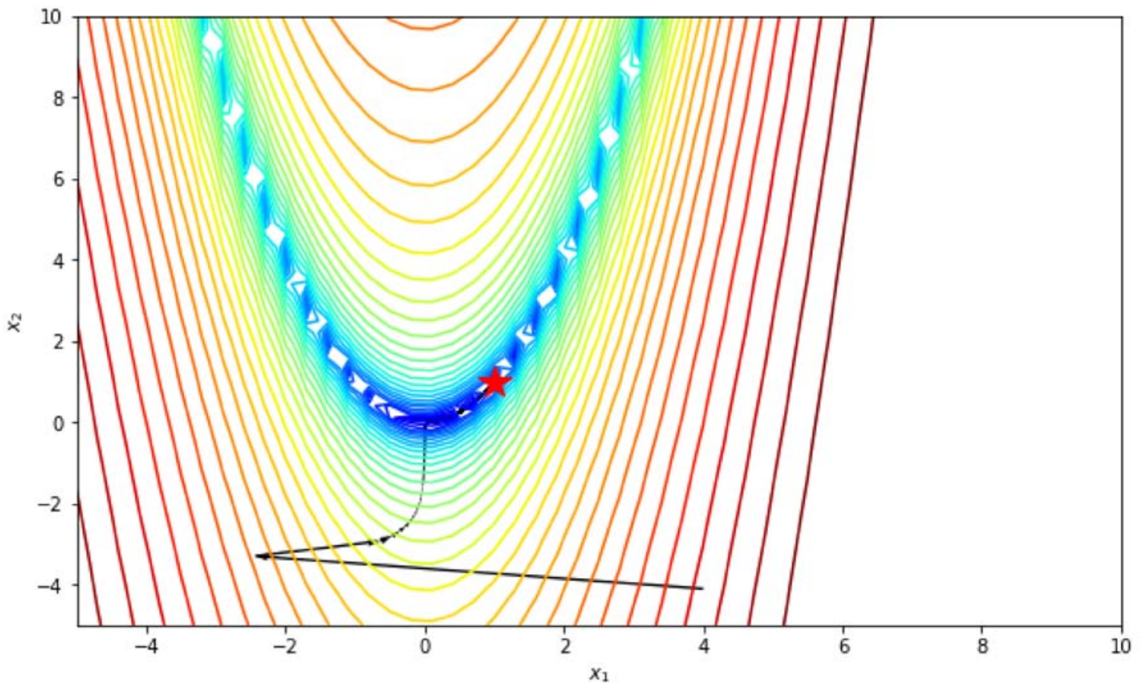
In [18]:

```
# Replace parameters here
def lr_1(t):
    if t < 1000:
        return 1
    else:
        return 0.0002
params = dict(
    method='adagrad',
    lr=lambda t:2,
    #lr=0.5,
    num_iters=500000000
)
ans.set_settings(fn_name='rosen2d', x0=np.array([8, 9]),tol=1e-14, **params)
ans.plot3d()
ans.path2d()
print(ans.get_min_errs())
params = dict(
    method='gd',
    lr=lambda t:0.0002,
    #lr=0.5,
    num_iters=500000
)
ans.set_settings(fn_name='rosen2d', x0=np.array([4, -4.1]),tol=1e-10, **params)
ans.path2d()
ans.get_min_errs()
```





```
(6.580397824891067e-10, 8.653853518666686e-20)
```



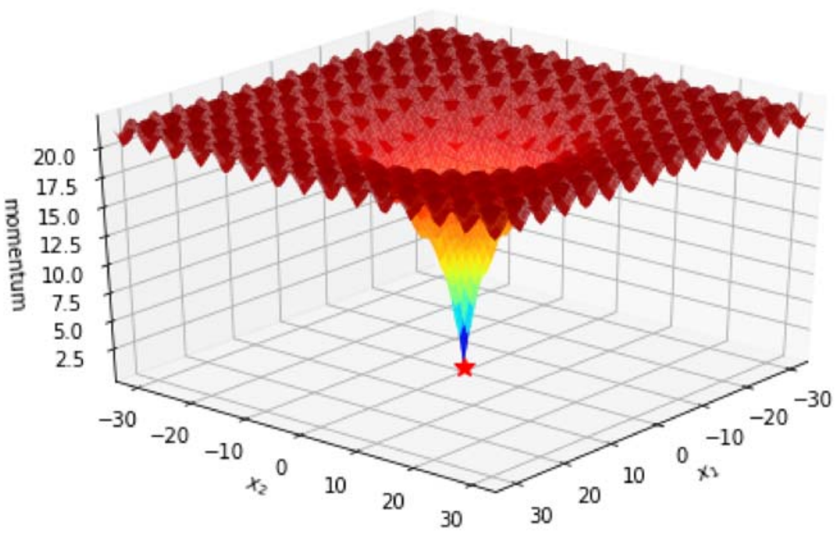
```
Out[18]:

(1.3198736580623927e-06, 3.478566656773554e-13)
```

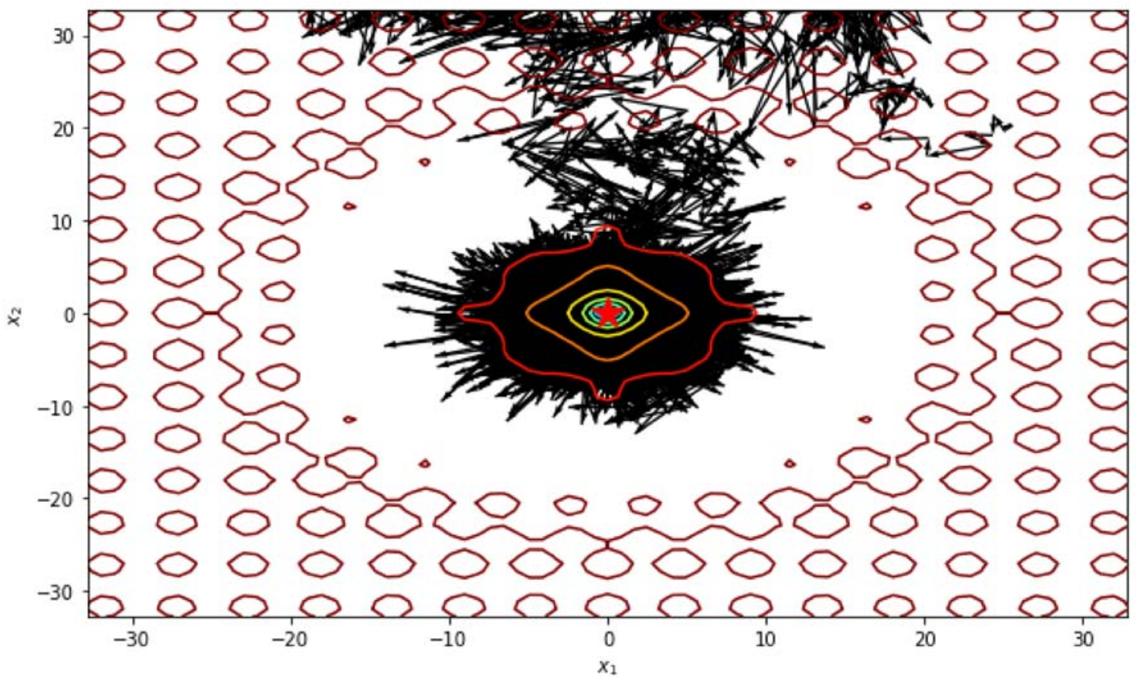
Ackley function

In [16]:

```
# Replace parameters here
def lr_t(t):
    c1 = 10000
    a = 1
    c2 = 15500
    rate = c1/(t**a+c2)
    return rate
params = dict(
    method='momentum',
    #lr=lambda t:0.425,
    lr=lr_t,
    num_iters=1000000,
    alpha=0.9
)
ans.set_settings(fn_name='ackley2d', x0=np.array([25, 20]),tol=1e-5, **params)
#ans.set_settings(fn_name='ackley2d',method='momentum', x0=np.array([25, 20]),tol=1e-5, **param
s)
#ans.compare('adagrad', lr=lr_t, num_iters=100000)
ans.plot3d()
ans.path2d()
ans.get_min_errs()
```

```
C:\Users\Znj\Anaconda3\lib\site-packages\ipykernel_launcher.py:25: RuntimeWarning:
divide by zero encountered in true_divide
C:\Users\Znj\Anaconda3\lib\site-packages\ipykernel_launcher.py:25: RuntimeWarning:
invalid value encountered in multiply
C:\Users\Znj\Anaconda3\lib\site-packages\ipykernel_launcher.py:26: RuntimeWarning:
divide by zero encountered in true_divide
C:\Users\Znj\Anaconda3\lib\site-packages\ipykernel_launcher.py:26: RuntimeWarning:
invalid value encountered in multiply
```





```
Out[16]:

(2.1740479700816824e-07, 6.1491488523302e-07)
```

Submission: Project Report

Create a report (which will probably be at least half a page) explaining the avenues you explored after the implementation phase of the project, the process you used to select the function values for each combination of functions and initial points, and what you found or learned. You are encouraged to include explanatory images or links to videos generated in the process, showcasing the process you describe or any interesting or unusual phenomena you observe over the course of your investigation!

Project Report

Simon Zhan, EECS 127 05/02/2020

Booth function

In Booth function part, since this function's global minimum matches its local minimum, we could use any choice of gradient descent, momentum, or NAG to achieve minimum point. After contrasting between gd and NAG and gd and $Momentum$ under the same parameter, we find out that gd has a better performance, but all the algorithms can achieve the required error with similar path. (Contrasting graph can be seen in the Booth code part).
The parameters I set is as follow:

- $learning_rate = 0.02$
- $max_iteration = 2500$
- $alpha(nag_momentum) = 0.8$
- $tolerance = 1^{-8}$

Beale function

In the Beale function, both $Momentum$ and gd method can reach the global minimum point. However, the path for $Momentum$ and gd is different. (For each different path graph can be seen from the code). The parameters I set for gd algorithm are:

- $learning_rate = 1^{-4}$
- $max_iteration = 100000$

For the $Momentum$ approach, the parameters I set are as followed:

- $learning_rate = 1^{-4}$
- $max_iteration = 100000$
- $alpha = 0.85$
- $tolerance = 1^{-8}$

From the contrasting result and descent path, we can find out that gd method has more direct path and closer result to the global minimum point than $Momentum$ does (specific result can be seen from the coding block of Beale function).

Rosenbrock function

For the Rosenbrock function, I did not find out the optimal parameters for gd, $Momentum$, and NAG at the initial point we plan to start. Thus, I use $Adagrad$ method with parameters:

- $learning_rate = 2$
- $max_iteration = 500000000$
- $initial_point = (8, 9)$
- $tolerance = 1^{-10}$

However, during the exploration, I find out that if we change the initial point, there are some other methods can be implemented to achieve minimum. For example, if we change to coordinate $(4, -4.1)$, we actually can use gd to reach the minimum point.(Detail implementation can be seen from the code block).

Ackley function

Optimizing Ackley function is tricky, since it has lots of local minims and maxims, which might trap in those local area if the step-size is too small. Besides, if the step-size is too big, those SDG algorithm may also wandering around the surrounding region of the global minimum (can be seen from the function graph). Therefore, I implement a step size changing strategy in this case. At the beginning, I want the step-size to be big, since I don't want to trap into the local minimum. Then, I want the step size starts to decreasing to get into the convex part of the function visualized from the graph and to get more accurate result. Thus, I define my step-size as:

$$\begin{aligned} \eta_{\mathrm{t}} &= \mathrm{c}_1 / \left(\mathrm{t}^{\mathrm{a}} + \mathrm{c}_2\right) \\ \text{where} \quad 0.5 &< \mathrm{a} < 2 \\ \mathrm{c1} &> 0 \\ \mathrm{c2} &\geq 0 \end{aligned} \tag{1}$$

t is the number of iteration.
Then by using $Momentum$ method with the following parameters

- $c_1 = 10000$
- $a = 1$
- $c_2 = 15500$
- $learning_rate = calculated$
- $max_iteration = 1000000$
- $initial_point = (25, 20)$
- $tolerance = 1^{-5}$