Quadratic analysis

In [10]:
```python
import numpy as np


def generate_sym_pd_matrix(n, rs, condition_number):
    R = rs.randint(-1000, 1000, (n, n))
    q, _ = np.linalg.qr(R, mode='complete')
    eigs = rs.permutation(1000)[:n] + 1
    eigs = eigs.astype(float)

    if condition_number is not None:
        idx = np.argmin(eigs)
        midx = np.argmax(eigs)
        k = eigs[midx]/eigs[idx]

        if k < condition_number:
            factor = np.sqrt(condition_number/k)
            eigs[midx] *= factor
            eigs[idx] /= factor
        elif k > condition_number:
            splice = np.where(eigs>eigs[idx] * condition_number)
            clip = eigs[idx] * condition_number
            eigs[splice] = clip

    A = np.matmul(np.matmul(q, np.diag(eigs)), q.T)
    return A


def generate_problem(n, *, seed=None, condition_number=None):
    if seed is None:
        seed = np.random.randint(1,10000)
    rs = np.random.RandomState(seed=seed)
    Q = generate_sym_pd_matrix(n, rs, condition_number)
    b = rs.randint(-1000, 1000, (n, 1))
    return {'Q': Q, 'b': b}
```

In [17]:

```python
import numpy as np


def grad(Q, b, x_k):
    return np.matmul(Q, x_k) - b


def norm(v):
    return np.sqrt(np.sum(np.square(v)))


def conjugate_gradient(Q, b, n, eps, *, seed=None):
    if seed is None:
        seed = np.random.randint(1, 10000)
    rs = np.random.RandomState(seed=seed)
    x = rs.rand(n).reshape(-1, 1)
    print("Q:\n{}\nb:\n{}\nstarting x:\n{}\n".format(Q, b, x))
    # x in R^n, with each component iid distributed in (0, 1)
    g = grad(Q, b, x)
    d = -g
    MM = np.matmul # shorthand
    niter = 0
    while norm(g) >= eps:
        niter += 1
        g = grad(Q, b, x)
        alpha = -MM(g.T, d)/MM(d.T, MM(Q, d))
        x = x + alpha * d
        g_1 = grad(Q, b, x)
        beta = MM(g_1.T, MM(Q, d))/MM(d.T, MM(Q, d))
        d = -g_1 + beta * d
        if niter == n:
            break
    return x, niter
```

In [29]:

```python
n = 10
eps = 1e-3
seed = None
print("Got args: n = {}, eps = {}, seed = {}".format(n, eps, seed))

problem = generate_problem(n, seed=seed, condition_number=10000)
Q, b = problem['Q'], problem['b']

x, niter = conjugate_gradient(Q, b, n, eps, seed=seed)
print("x* :\n{}".format(x))

x_star = np.matmul(np.linalg.inv(Q), b)
print("Actual x* :\n{}".format(x_star))
print("It took niter = {} iterations to reach this point".format(niter))

err = norm(np.abs(x_star - x))/norm(x_star)
print("Error = {}".format(err))
```

```
Got args: n = 10, eps = 0.001, seed = None
Q:
[[ 2057.73075847 -2298.78427685  1124.23780673 -1737.3808475
  -1406.26131995  1103.47565582   955.16808217 -1522.82091432
   2790.53830625  2242.74512043]
 [-2298.78427685  3815.28083099 -1635.35611851  2778.89614658
   1972.43007955 -1433.5743954  -1548.7048904   2184.92033141
  -4227.52725458 -3125.99832003]
 [ 1124.23780673 -1635.35611851  1263.18653638 -1192.35778203
   -837.59241233   556.63190093   635.08016606  -923.30540102
   2048.21566761  1723.00255669]
 [-1737.3808475   2778.89614658 -1192.35778203  2572.74979982
   1497.03536361 -1309.10598478 -1213.27767774  1657.550548
  -3090.65122502 -2635.73960321]
 [-1406.26131995  1972.43007955  -837.59241233  1497.03536361
   1505.28884413 -1000.51622207  -818.62108374  1144.78963787
  -2561.79698551 -1927.141723  ]
 [ 1103.47565582 -1433.5743954    556.63190093 -1309.10598478
  -1000.51622207  1078.9457485    721.12180362 -1190.17926049
   2107.1923668   1454.7834009 ]
 [  955.16808217 -1548.7048904    635.08016606 -1213.27767774
   -818.62108374   721.12180362  1018.55724155 -1055.64052071
   1735.58813883  1299.3335512 ]
 [-1522.82091432  2184.92033141  -923.30540102  1657.550548
   1144.78963787 -1190.17926049 -1055.64052071  2013.33864347
  -2894.20586146 -2321.04898421]
 [ 2790.53830625 -4227.52725458  2048.21566761 -3090.65122502
  -2561.79698551  2107.1923668   1735.58813883 -2894.20586146
   5789.71189528  4142.49009194]
 [ 2242.74512043 -3125.99832003  1723.00255669 -2635.73960321
  -1927.141723    1454.7834009   1299.3335512  -2321.04898421
   4142.49009194  4005.72469887]]
b:
[[ -49]
 [ -81]
 [ 644]
 [ 489]
 [ 952]
 [ 351]
 [ 853]
 [-430]
 [-305]
 [ 872]]
starting x:
[[0.66806938]
 [0.32825422]
 [0.94021108]
 [0.53547362]
 [0.68167334]
 [0.48861826]
 [0.66830164]
 [0.91731571]
 [0.82414468]
 [0.70471485]]

x* :
[[-3.30914670e+01]
 [-1.75142312e+02]
 [ 7.64911887e-02]
 [ 1.62774264e+02]
 [ 9.12125219e+01]
 [ 2.52636049e+02]
 [-2.16880513e+00]
```

```
 [ 1.10172202e+02]
 [-9.66990091e+01]
 [ 1.05814853e+02]]
Actual x* :
[[-3.30750310e+01]
 [-1.75075160e+02]
 [ 1.78245043e-01]
 [ 1.62718619e+02]
 [ 9.12761319e+01]
 [ 2.52735735e+02]
 [-2.17456986e+00]
 [ 1.10194112e+02]
 [-9.66844786e+01]
 [ 1.05767431e+02]]
It took niter = 10 iterations to reach this point
Error = 0.00046445693340865373
```

In [31]:

```
def steepest_descent(Q, b, n, eps, *, seed=None):
    if seed is None:
        seed = np.random.randint(1, 10000)
    rs = np.random.RandomState(seed=seed)
    x = rs.rand(n).reshape(-1, 1)
    print("Q:\n{}\nb:\n{}\nstarting x:\n{}\n".format(Q, b, x))
    # x in R^n, with each component iid distributed in (0, 1)
    g = grad(Q, b, x)
    MM = np.matmul # shorthand
    niter = 0
    while norm(g) >= eps:
        niter += 1
        g = grad(Q, b, x)
        alpha = MM(MM(MM(x.T, Q), Q), x) - 2 * MM(MM(x.T, Q), b) + MM(b.T, b
        alpha /= MM(MM(MM(g.T, Q), Q), x) - MM(MM(g.T, Q), b)
        x = x - alpha * g
    return x, niter

x, niter = steepest_descent(Q, b, n, eps, seed=seed)
print("x* :\n{}".format(x))

x_star = np.matmul(np.linalg.inv(Q), b)
print("Actual x* :\n{}".format(x_star))
print("It took niter = {} iterations to reach this point".format(niter))

err = norm(np.abs(x_star - x))/norm(x_star)
print("Error = {}".format(err))
```

```
Q:
[[ 2057.73075847 -2298.78427685  1124.23780673 -1737.3808475
  -1406.26131995  1103.47565582   955.16808217 -1522.82091432
   2790.53830625  2242.74512043]
 [-2298.78427685  3815.28083099 -1635.35611851  2778.89614658
   1972.43007955 -1433.5743954  -1548.7048904   2184.92033141
  -4227.52725458 -3125.99832003]
 [ 1124.23780673 -1635.35611851  1263.18653638 -1192.35778203
   -837.59241233   556.63190093   635.08016606  -923.30540102
   2048.21566761  1723.00255669]
 [-1737.3808475   2778.89614658 -1192.35778203  2572.74979982
   1497.03536361 -1309.10598478 -1213.27767774  1657.550548
  -3090.65122502 -2635.73960321]
 [-1406.26131995  1972.43007955  -837.59241233  1497.03536361
   1505.28884413 -1000.51622207  -818.62108374  1144.78963787
  -2561.79698551 -1927.141723  ]
 [ 1103.47565582 -1433.5743954    556.63190093 -1309.10598478
  -1000.51622207  1078.9457485    721.12180362 -1190.17926049
   2107.1923668   1454.7834009 ]
 [  955.16808217 -1548.7048904    635.08016606 -1213.27767774
   -818.62108374   721.12180362  1018.55724155 -1055.64052071
   1735.58813883  1299.3335512 ]
 [-1522.82091432  2184.92033141  -923.30540102  1657.550548
   1144.78963787 -1190.17926049 -1055.64052071  2013.33864347
  -2894.20586146 -2321.04898421]
 [ 2790.53830625 -4227.52725458  2048.21566761 -3090.65122502
  -2561.79698551  2107.1923668   1735.58813883 -2894.20586146
   5789.71189528  4142.49009194]
 [ 2242.74512043 -3125.99832003  1723.00255669 -2635.73960321
  -1927.141723    1454.7834009   1299.3335512  -2321.04898421
   4142.49009194  4005.72469887]]
b:
[[ -49]
 [ -81]
 [ 644]
 [ 489]
 [ 952]
 [ 351]
 [ 853]
 [-430]
 [-305]
 [ 872]]
starting x:
[[0.36822007]
 [0.20958677]
 [0.79692408]
 [0.81862509]
 [0.89737427]
 [0.74897576]
 [0.6138    ]
 [0.86437151]
 [0.15213447]
 [0.94802124]]

x* :
[[-3.30749919e+01]
 [-1.75074955e+02]
 [ 1.78245539e-01]
 [ 1.62718430e+02]
 [ 9.12760280e+01]
 [ 2.52735440e+02]
 [-2.17456388e+00]
 [ 1.10193984e+02]
```

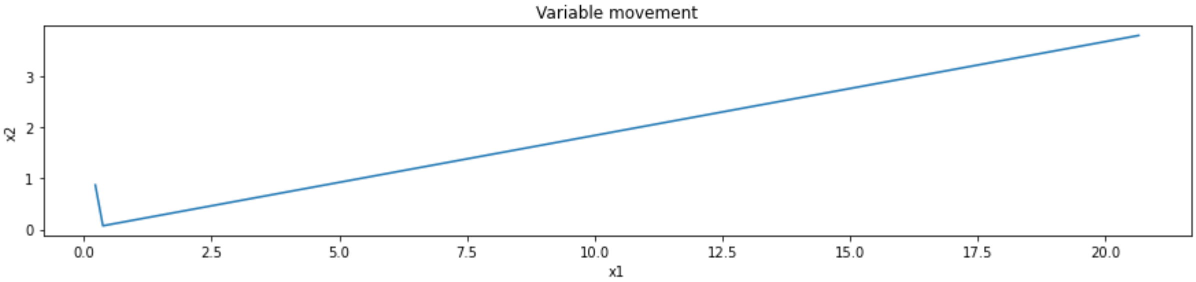
```
 [-9.66843653e+01]
 [ 1.05767309e+02]]
Actual x* :
[[-3.30750310e+01]
 [-1.75075160e+02]
 [ 1.78245043e-01]
 [ 1.62718619e+02]
 [ 9.12761319e+01]
 [ 2.52735735e+02]
 [-2.17456986e+00]
 [ 1.10194112e+02]
 [-9.66844786e+01]
 [ 1.05767431e+02]]
It took niter = 40388 iterations to reach this point
Error = 1.164459804289519e-06
```

Notice It takes conjugate gradients 10 iterations while it takes steepest descent $\mathbf{40388!}$ iterations.

In [48]:

```
%matplotlib inline

import matplotlib.pyplot as plt


def cg_return_x(Q, b, n, eps, *, seed=None):
    if n != 2:
        raise NotImplementedError
    if seed is None:
        seed = np.random.randint(1, 10000)
    rs = np.random.RandomState(seed=seed)
    xs = []
    x = rs.rand(n).reshape(-1, 1)
    xs.append(x)
    #print("Q:\n{}\nb:\n{}\nstarting x:\n{}\n".format(Q, b, x))
    # x in R^n, with each component iid distributed in (0, 1)
    g = grad(Q, b, x)
    d = -g
    MM = np.matmul # shorthand
    niter = 0
    while norm(g) >= eps:
        niter += 1
        g = grad(Q, b, x)
        alpha = -MM(g.T, d)/MM(d.T, MM(Q, d))
        x = x + alpha * d
        xs.append(x)
        g_1 = grad(Q, b, x)
        beta = MM(g_1.T, g_1)/MM(g.T, g)
        d = -g_1 + beta * d
        if niter == n:
            break
    return np.asarray(xs)

n = 2
eps = 1e-3
seed = 5
print("Got args: n = {}, eps = {}, seed = {}".format(n, eps, seed))

problem2 = generate_problem(n, seed=seed, condition_number=10000)
Q2, b2 = problem2['Q'], problem2['b']

xs = cg_return_x(Q2, b2, n, eps, seed=seed)
fig = plt.figure(figsize=(15,15))
plt.plot(xs[:,0],xs[:,1])
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Variable movement")
plt.gca().set_aspect('equal')
print()
```

```
Got args: n = 2, eps = 0.001, seed = 5
```



General function case

Scheme to follow:

- Choose $x^0 \in \mathbb{R}^n$,$d^0 = -g^0 = -\nabla f(x^0)$.
- At each iteration:
  - $x^{k+1} = x^k + \alpha_k d^k$
  - $\alpha_k = \arg\min_\alpha f(x^k + \alpha d^k)$
  - $d^{k+1} = -g^{k+1} + \beta_k d^k$
  - $\beta_k = \dfrac{(g^{k+1})^T g^{k+1}}{(g^k)^T g^k}$
- To find α_k using $\alpha_k = \frac{(d^k)^T d^k}{(d^k)^T Q d^k}$, estimating the denominator term with $(d^k)^T H d^k$, H is the Hessian and estimating that as:
  - $\hat{f} \equiv f(x^k + \hat{\alpha}_k d^k)$ and using Taylor's on this.
  - $(d^k)^T H d^k = 2\dfrac{\hat{f} - f(x^k) - \hat{\alpha}_k (d^k)^T \nabla f(x^k)}{\hat{\alpha}_k^2}$

In [50]:
```python
import autograd.numpy as np


def sphere(d):
    r"""Minima at 0"""
    def function(x):
        # assert isinstance(x, np.ndarray)
        assert x.shape == (d, 1)
        return np.sum(np.square(x))
    return function


def sum_of_squares(d):
    r"""Minima at 0"""
    def function(x):
        # assert isinstance(x, np.ndarray)
        assert x.shape == (d, 1)
        return np.sum(np.arange(1, d + 1).reshape(-1, 1) * np.square(x))
    return function


def sum_of_diff_pow(d):
    r"""Minima at 0"""
    def function(x):
        # assert isinstance(x, np.ndarray)
        assert x.shape == (d, 1)
        s = 0
        for j, i in enumerate(x,start=2):
            s += np.power(np.abs(i), j)
        return s
    return function


def booth_function(d):
    r"""Minima at (1, 3)"""
    assert d == 2, "Booth function is defined for 2D inputs only."
    def function(x):
        assert x.shape == (d, 1)
        return (x[0]+2*x[1]-7)**2+(2*x[0]+x[1]-5)**2
    return function


def matyas(d):
    r"""Minima at (0, 0)"""
    assert d == 2, "Matyas function is defined for 2D inputs only."
    def function(x):
        assert x.shape == (d, 1)
        return 0.26*(x[0]**2+x[1]**2)-0.48*x[0]*x[1]
    return function


def six_humped_camel(d):
    r"""Minima at (0.0898, -0.7126) and (-0.0898, 0.7126)"""
    assert d == 2, "Six Humped Camel function is defined for 2D inputs only.
    def function(x):
        assert x.shape == (d, 1)
        t1 = (4 - 2.1 * x[0] ** 2 + x[0] ** 4 / 3) * x[0] ** 2
        t2 = x[0] * x[1]
        t3 = (-4 + 4 * x[1] ** 2) * x[1] ** 2
        return t1 + t2 + t3
    return function
```

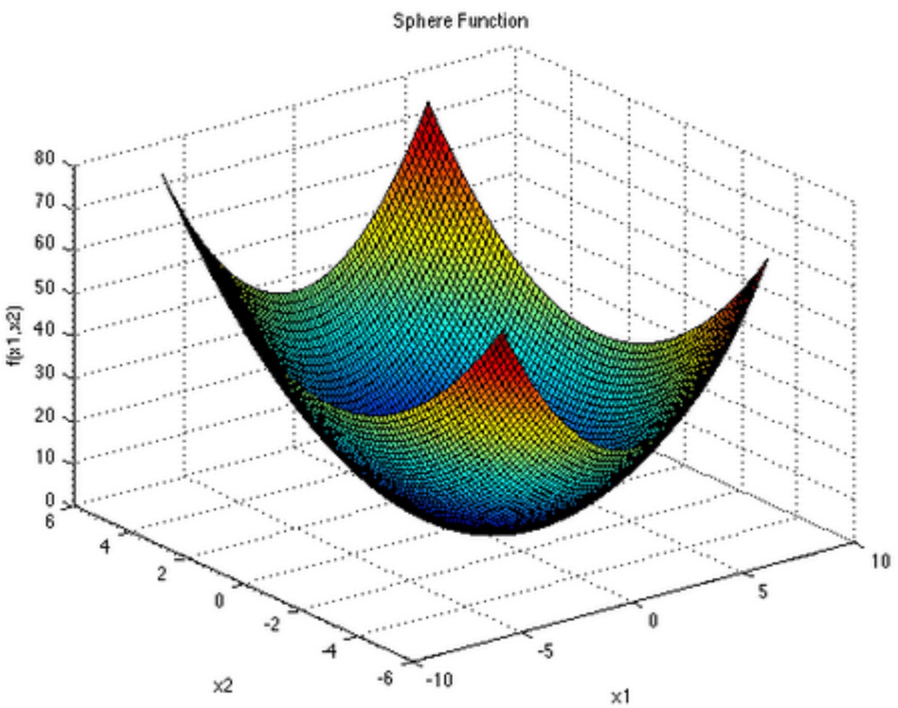
```
def bukin_n6(d):
    r"""Global minima at (-10, 1)"""
    assert d == 2, "Bukin function is defined for 2D inputs only."
    def function(x):
        assert x.shape == (d, 1)
        t0 = np.sqrt(np.abs(x[1] - 0.01 * x[0] ** 2))
        t1 = 100 * t0
        t2 = 0.01 * np.abs(x[0] + 10)
        return t1 + t2
    return function


def drop_wave(d):
    r"""Global minima at (0, 0)"""
    assert d == 2, "Drop wave undefined for non 2D inputs."
    def function(x):
        assert x.shape == (d, 1)
        num = 1 + np.cos(12 * np.sqrt(np.sum(np.square(x))))
        denom = 0.5 * np.sum(np.square(x)) + 2
        return -num/denom
    return function


def beale(d):
    r"""Global minima at (3, 0.5)"""
    assert d == 2, "Beale function defined for 2D case."
    def function(x):
        assert x.shape == (d, 1)
        t1 = 1.5 - x[0] + x[0] * x[1]
        t2 = 2.25 - x[0] + x[0] * x[1] * x[1]
        t3 = 2.625 - x[0] + x[0] * x[1] * x[1] * x[1]
        return t1 ** 2 + t2 ** 2 + t3 ** 2
    return function
```

In [57]:

```
import autograd.numpy as np
from autograd import grad


def norm(v):
    return np.sqrt(np.sum(np.square(v)))


def estimate_alpha_k(f, x_k, alpha_hat, gradient_xk, d_k):
    x_k_hat = x_k + alpha_hat * d_k
    f_k_hat = f(x_k_hat)
    norm_d_sq = np.matmul(d_k.T, d_k)
    norm_g_sq = np.matmul(d_k.T, gradient_xk).reshape(-1)
    alpha_k = norm_d_sq * alpha_hat * alpha_hat
    alpha_k /= 2 * (f_k_hat - f(x_k) - alpha_hat * norm_g_sq)
    return alpha_k


def conjugate_gradient_general(f, *, n, eps, alpha_hat, seed=None):
    if seed is None:
        seed = np.random.randint(1, 1000)
    rs = np.random.RandomState(seed=seed)
    x = rs.randn(n).reshape(-1, 1)
    fs = [f(x)]
    print("starting x:\n{}\n".format(x))
    gradf = grad(f) # f needs to be a pure function
    g = gradf(x)
    d = -g
    MM = np.matmul
```

```python
    MM = np.matmul
    niter = 0
    while norm(g) >= eps:
        if niter > 10000:
            print("Did not converge in 10000 iterations")
            break
        niter += 1
        g = gradf(x)
        alpha_hat = estimate_alpha_k(f, x, alpha_hat, g, d)
        x = x + alpha_hat * d
        g_plus_1 = gradf(x)
        beta = MM(g_plus_1.T,g_plus_1)/MM(g.T,g)
        d = -g_plus_1+beta*d
        if niter % n == 0:
            d = -g_plus_1
        fs.append(f(x))
    return x, niter, fs
```

Sphere function



$$f(\mathbf{x}) = \sum_{i=1}^{d} x_i^2$$

Note that the optimization function here is quadratic and equal to:

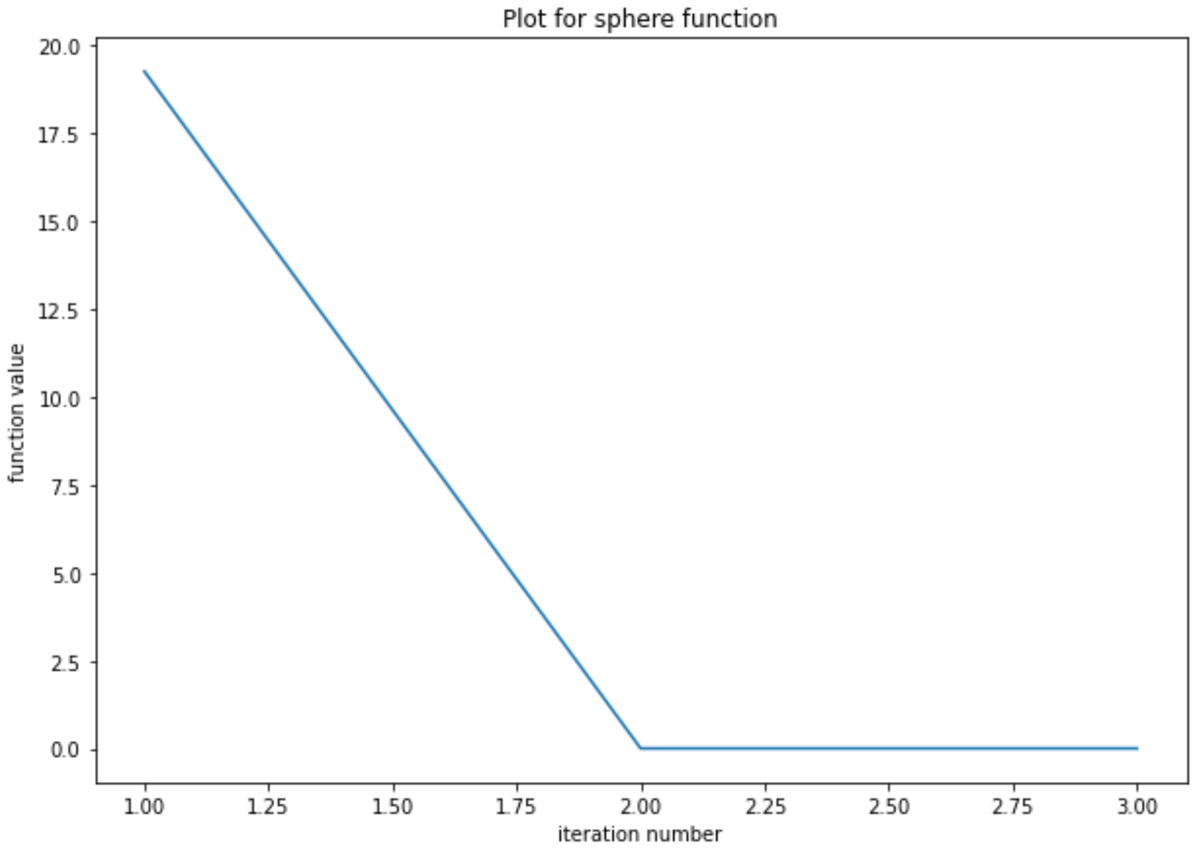
$$f(\mathbf{x}) = \mathbf{x}^T \mathbf{I}_{d\times d} \mathbf{x}$$

so this is a very well behaved function.

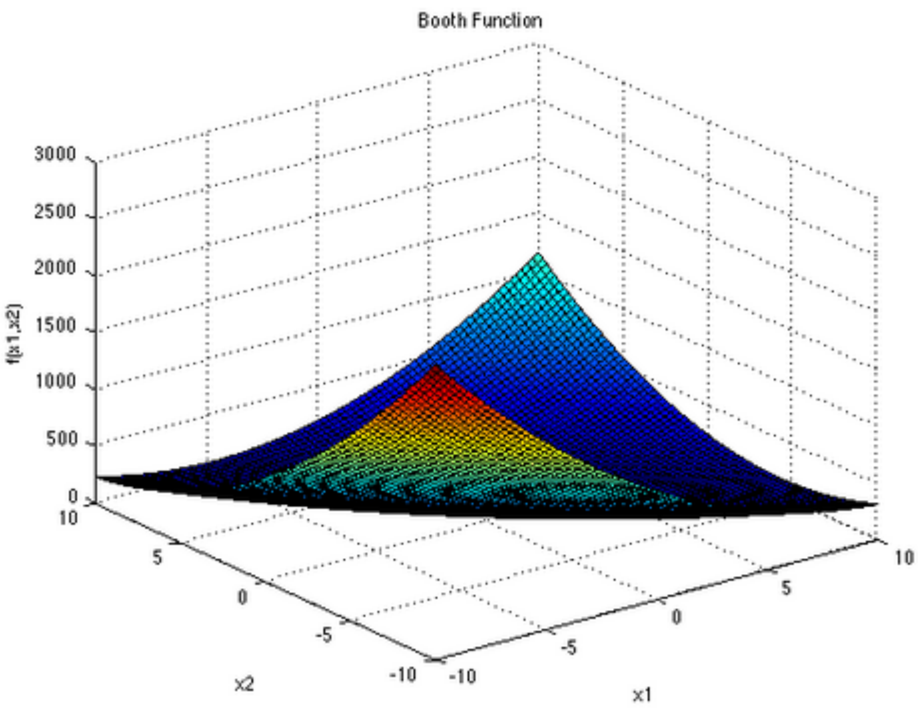
In [56]:

```
n = 10
eps = 1e-3
seed = None
alpha_hat = 1e-3
print("Got args: n = {}, eps = {}, seed = {}, alpha_hat = {}".format(
    n,
    eps,
    seed,
    alpha_hat,
    ))
function = sphere(n)
x, niter, fs = conjugate_gradient_general(
        function,
        n=n,
        eps=eps,
        seed=seed,
        alpha_hat=alpha_hat)
print(x, niter)
figure = plt.figure(figsize=(10,7))
plt.plot(range(1, niter + 2),fs,label='Function value')
plt.xlabel('iteration number')
plt.ylabel('function value')
_=plt.title('Plot for sphere function')
```

```
Got args: n = 10, eps = 0.001, seed = None, alpha_hat = 0.001
starting x:
[[-1.04369259]
 [ 1.36570416]
 [ 0.1451076 ]
 [ 0.45646149]
 [-1.99049059]
 [ 1.70943087]
 [ 2.71164754]
 [ 0.6546749 ]
 [-0.17996299]
 [ 1.17106345]]

[[ 1.24802167e-21]
 [-1.63308071e-21]
 [-1.73516482e-22]
 [-5.45826113e-22]
 [ 2.38018636e-21]
 [-2.04409254e-21]
 [-3.24253508e-21]
 [-7.82845670e-22]
 [ 2.15196206e-22]
 [-1.40032630e-21]] 2
```



Booth function



$$f(\mathbf{x}) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$$

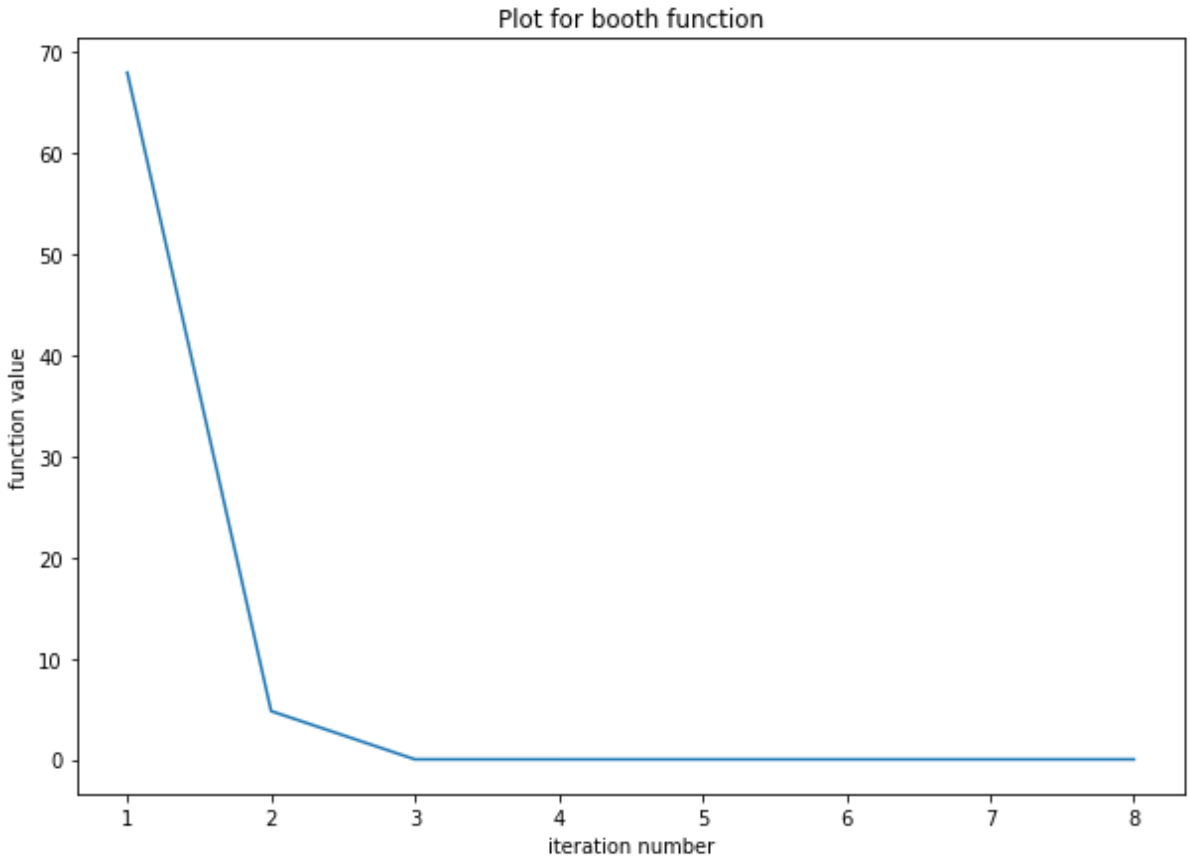
This is also a quadratic problem:

$$f(\mathbf{x}) = \mathbf{x}^T \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix} \mathbf{x} - \begin{bmatrix} 34 \\ 48 \end{bmatrix}^T \mathbf{x} - 75$$

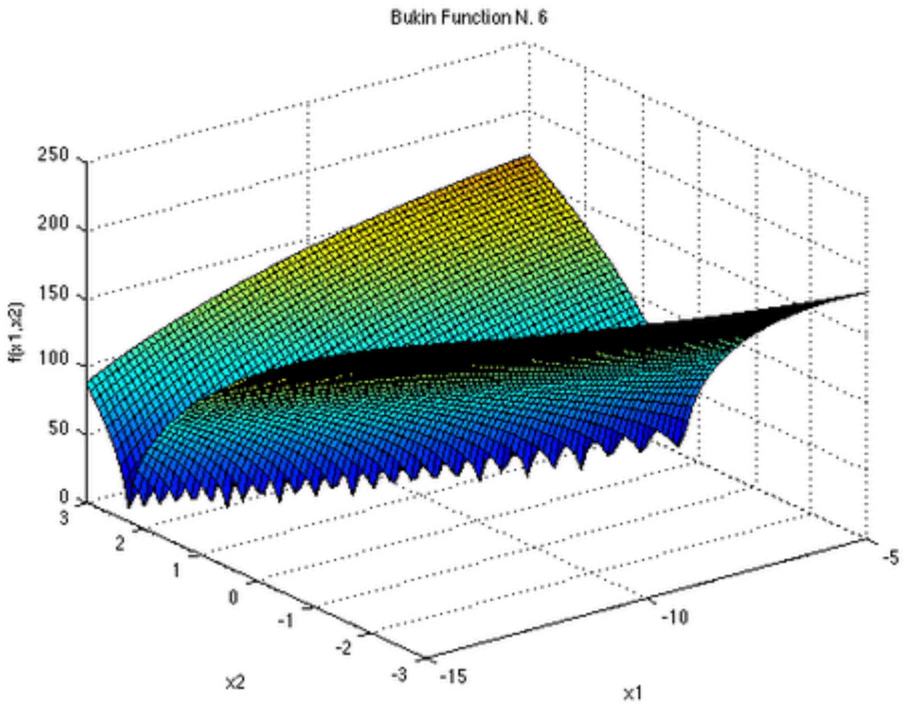
In [59]:

```
n = 2
eps = 1e-3
seed = None
alpha_hat = 1e-3
print("Got args: n = {}, eps = {}, seed = {}, alpha_hat = {}".format(
    n,
    eps,
    seed,
    alpha_hat,
    ))
function = booth_function(n)
x, niter, fs = conjugate_gradient_general(
        function,
        n=n,
        eps=eps,
        seed=seed,
        alpha_hat=alpha_hat)
print(x, niter)
figure = plt.figure(figsize=(10,7))
plt.plot(range(1, niter + 2),fs,label='Function value')
plt.xlabel('iteration number')
plt.ylabel('function value')
_=plt.title('Plot for booth function')
```

```
Got args: n = 2, eps = 0.001, seed = None, alpha_hat = 0.001
starting x:
[[ 0.88478703]
 [-0.59327974]]

[[1.00000002]
 [3.00000075]] 7
```



Bukin N6



$$f(\mathbf{x}) = 100\sqrt{|x_2 - 0.01x_1^2|} + 0.01|x_1 + 10|$$

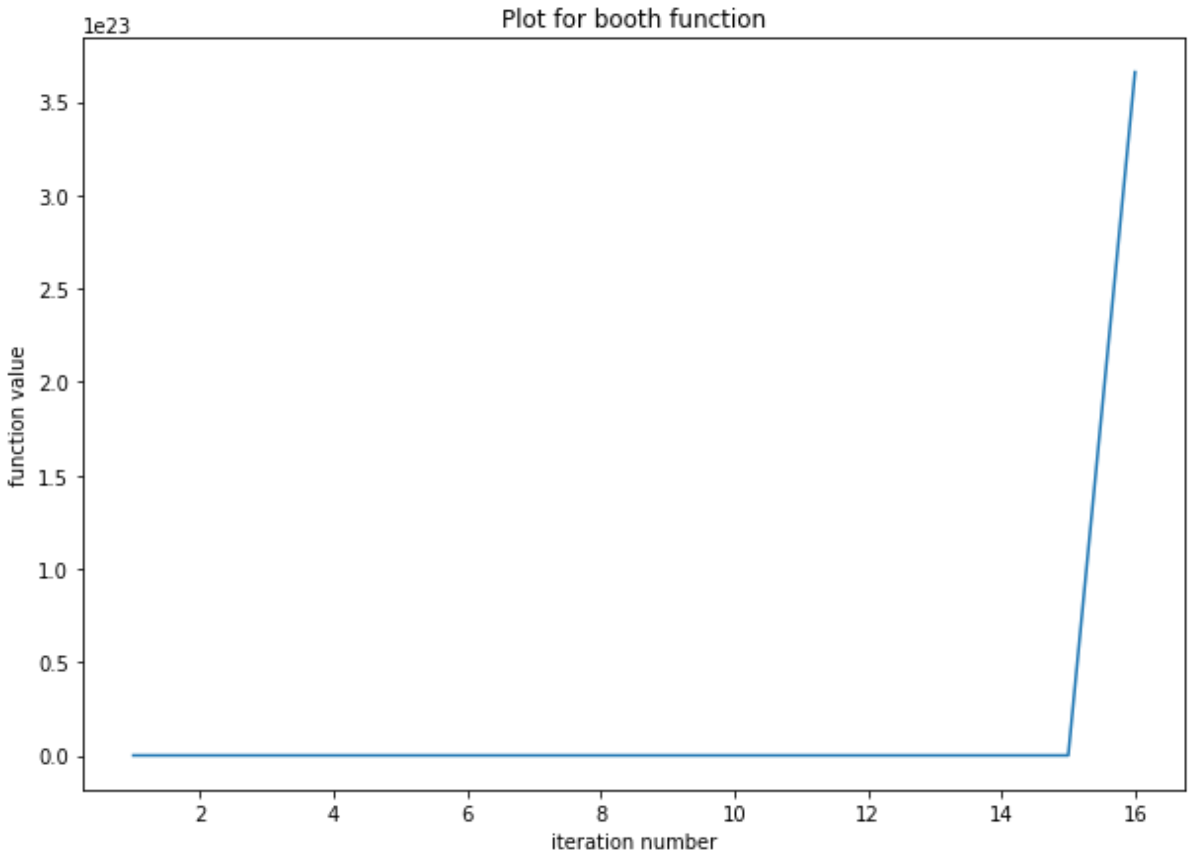
Global minima is at $\mathbf{x}^* = (-10, 1)$ and $f(\mathbf{x}^*) = 0$

In [65]:

```python
n = 2
eps = 1e-3
seed = None
alpha_hat = 1e-3
print("Got args: n = {}, eps = {}, seed = {}, alpha_hat = {}".format(
    n,
    eps,
    seed,
    alpha_hat,
    ))
function = bukin_n6(n)
x, niter, fs = conjugate_gradient_general(
        function,
        n=n,
        eps=eps,
        seed=seed,
        alpha_hat=alpha_hat)
print(x, niter)
figure = plt.figure(figsize=(10,7))
plt.plot(range(1, niter + 2),fs,label='Function value')
plt.xlabel('iteration number')
plt.ylabel('function value')
_=plt.title('Plot for booth function')
```

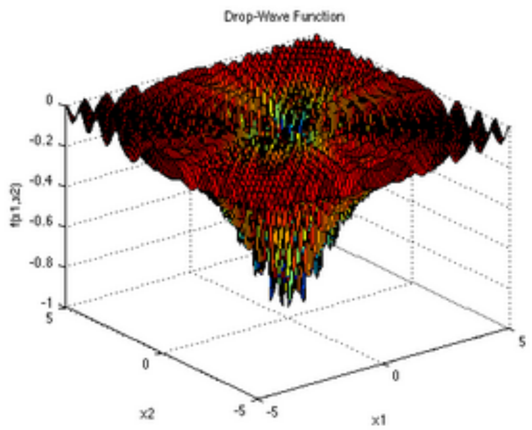
```
Got args: n = 2, eps = 0.001, seed = None, alpha_hat = 0.001
starting x:
[[1.71718429]
 [0.32469333]]

[[nan]
 [nan]] 17
```

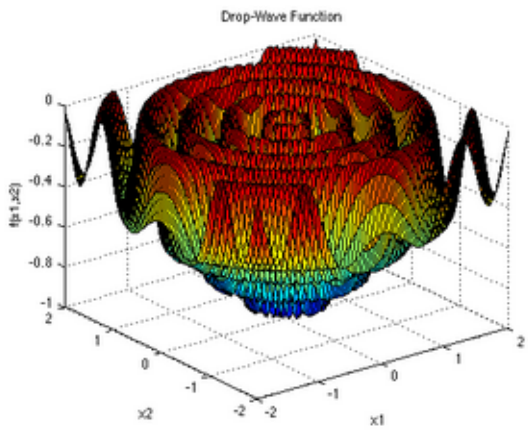
```
/tmp/ipykernel_58321/2384222872.py:15: RuntimeWarning: divide by zero encoun
tered in true_divide
  alpha_k /= 2 * (f_k_hat - f(x_k) - alpha_hat * norm_g_sq)
/tmp/ipykernel_58321/3484441021.py:69: RuntimeWarning: invalid value encount
ered in subtract
  t0 = np.sqrt(np.abs(x[1] - 0.01 * x[0] ** 2))
```



The function is highly unstable, the function value actually increases instead of decreasing.

Dropwave



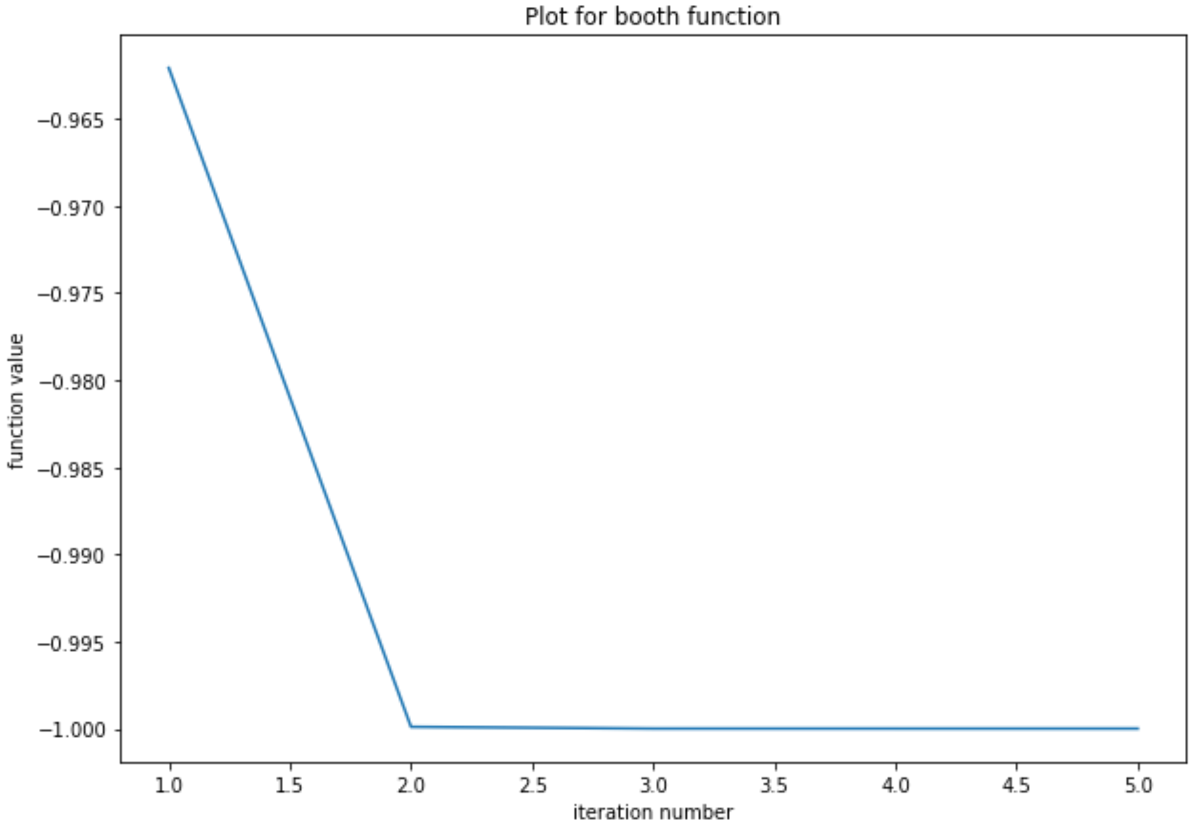


$$f(\mathbf{x}) = -\frac{1 + \cos\left(12\sqrt{x_1^2 + x_2^2}\right)}{0.5(x_1^2 + x_2^2) + 2}$$

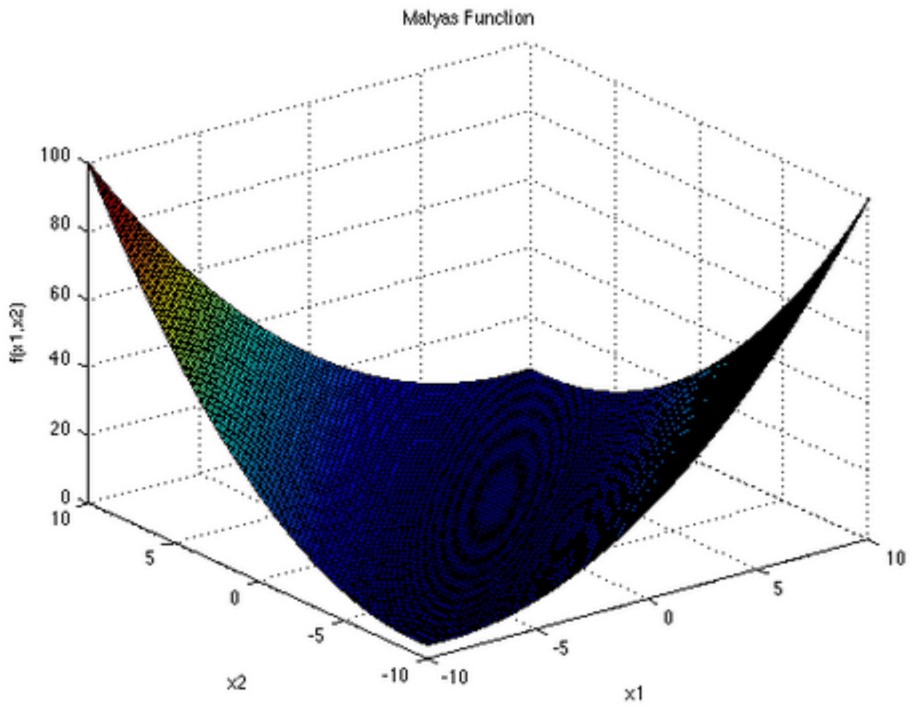
Global minimum is $f(\mathbf{x}^*) = -1$ at $\mathbf{x}^* = (0, 0)$

In [66]:
```python
n = 2
eps = 1e-3
seed = None
alpha_hat = 1e-3
print("Got args: n = {}, eps = {}, seed = {}, alpha_hat = {}".format(
    n,
    eps,
    seed,
    alpha_hat,
    ))
function = drop_wave(n)
x, niter, fs = conjugate_gradient_general(
        function,
        n=n,
        eps=eps,
        seed=seed,
        alpha_hat=alpha_hat)
print(x, niter)
figure = plt.figure(figsize=(10,7))
plt.plot(range(1, niter + 2),fs,label='Function value')
plt.xlabel('iteration number')
plt.ylabel('function value')
_=plt.title('Plot for booth function')
```

```
Got args: n = 2, eps = 0.001, seed = None, alpha_hat = 0.001
starting x:
[[0.03012462]
 [0.01232506]]

[[3.67495349e-12]
 [1.50355473e-12]] 4
```



Matyas



$$f(\mathbf{x}) = 0.26(x_1^2 + x_2^2) - 0.48x_1x_2$$

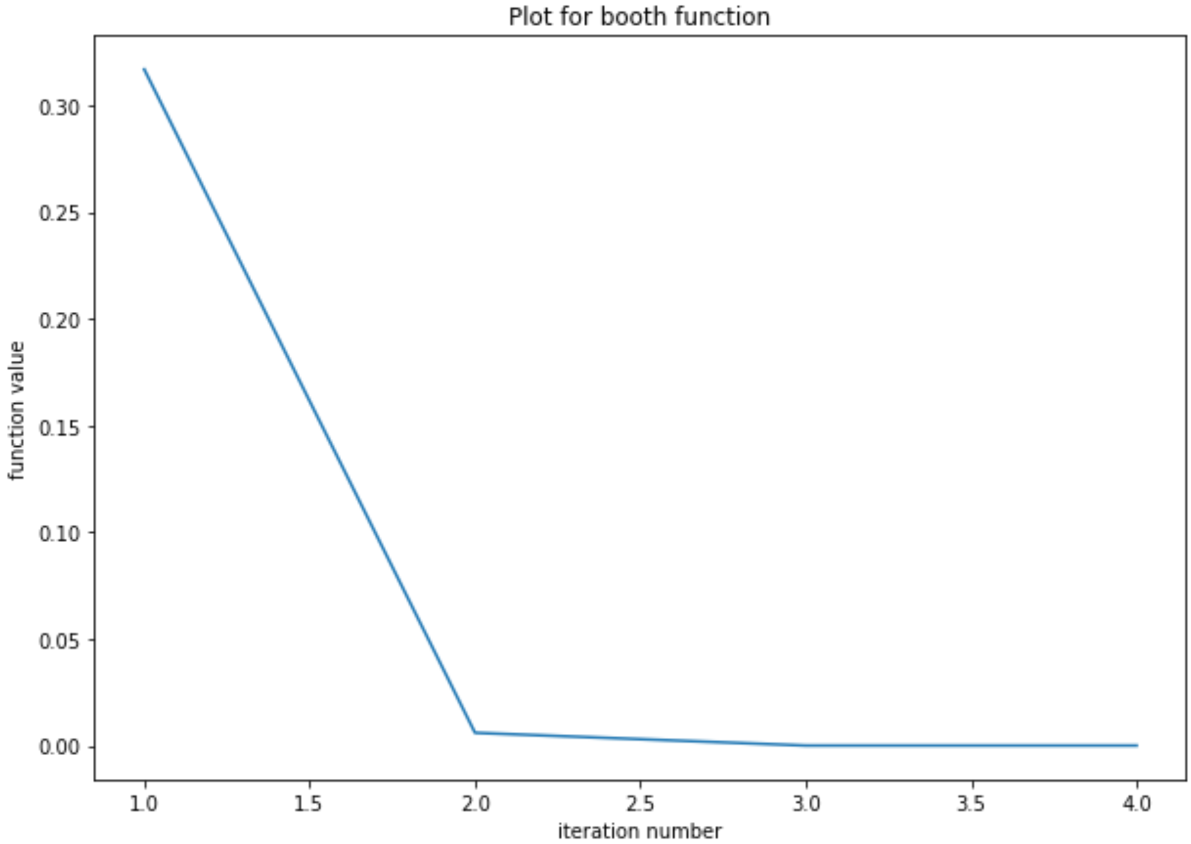
This is equal to:

$$f(x) = \mathbf{x}^T \begin{bmatrix} 0.26 & -0.24 \\ -0.24 & 0.26 \end{bmatrix} \mathbf{x}$$

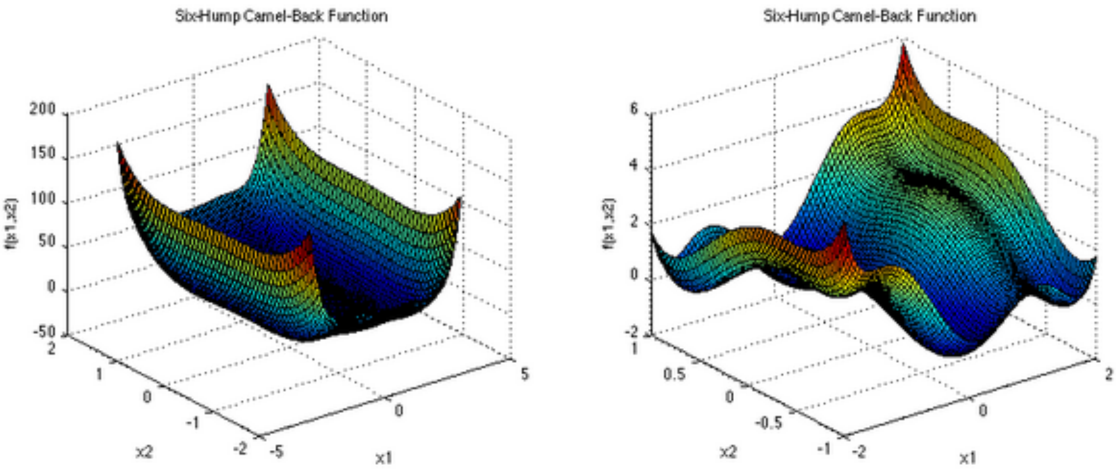
In [67]:

```
n = 2
eps = 1e-3
seed = None
alpha_hat = 1e-3
print("Got args: n = {}, eps = {}, seed = {}, alpha_hat = {}".format(
    n,
    eps,
    seed,
    alpha_hat,
    ))
function = matyas(n)
x, niter, fs = conjugate_gradient_general(
        function,
        n=n,
        eps=eps,
        seed=seed,
        alpha_hat=alpha_hat)
print(x, niter)
figure = plt.figure(figsize=(10,7))
plt.plot(range(1, niter + 2),fs,label='Function value')
plt.xlabel('iteration number')
plt.ylabel('function value')
_=plt.title('Plot for booth function')
```

```
Got args: n = 2, eps = 0.001, seed = None, alpha_hat = 0.001
starting x:
[[ 0.95943203]
 [-0.15479881]]

[[-1.40103043e-05]
 [ 2.21845620e-06]] 3
```



Six hump camel

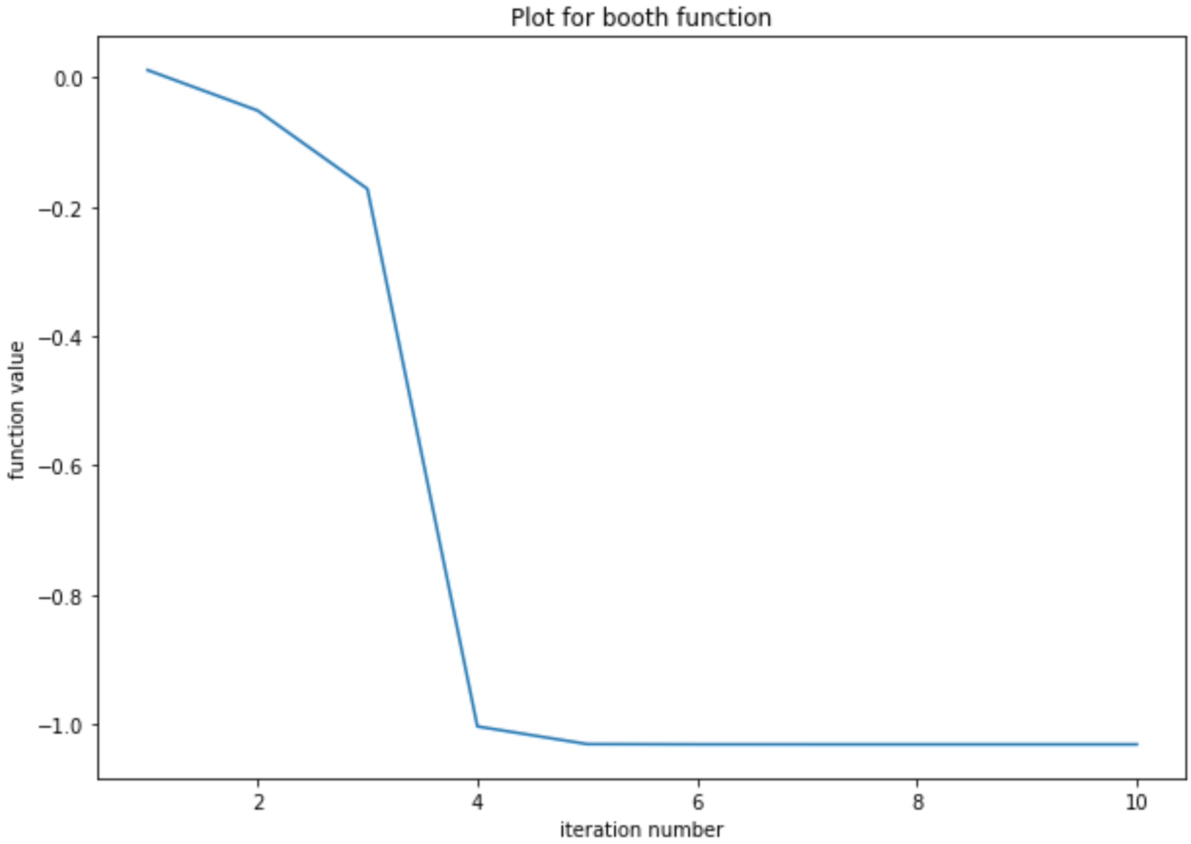


$$f(\mathbf{x}) = \left(4 - 2.1x_1^2 + \frac{x_1^4}{3}\right)x_1^2 + x_1x_2 + \left(-4 + 4x_2^2\right)x_2^2$$

Global minimum $f(\mathbf{x}^*) = -1.0316$ at $\mathbf{x}^* = (0.0898, -0.7126)$ and $(-0.0898, 0.7126)$.

In [68]:
```
n = 2
eps = 1e-3
seed = None
alpha_hat = 1e-3
print("Got args: n = {}, eps = {}, seed = {}, alpha_hat = {}".format(
    n,
    eps,
    seed,
    alpha_hat,
    ))
function = six_humped_camel(n)
x, niter, fs = conjugate_gradient_general(
        function,
        n=n,
        eps=eps,
        seed=seed,
        alpha_hat=alpha_hat)
print(x, niter)
figure = plt.figure(figsize=(10,7))
plt.plot(range(1, niter + 2),fs,label='Function value')
plt.xlabel('iteration number')
plt.ylabel('function value')
_=plt.title('Plot for booth function')
```

```
Got args: n = 2, eps = 0.001, seed = None, alpha_hat = 0.001
starting x:
[[ 0.07136572]
 [-0.0397406 ]]

[[ 0.08984456]
 [-0.71265884]] 9
```

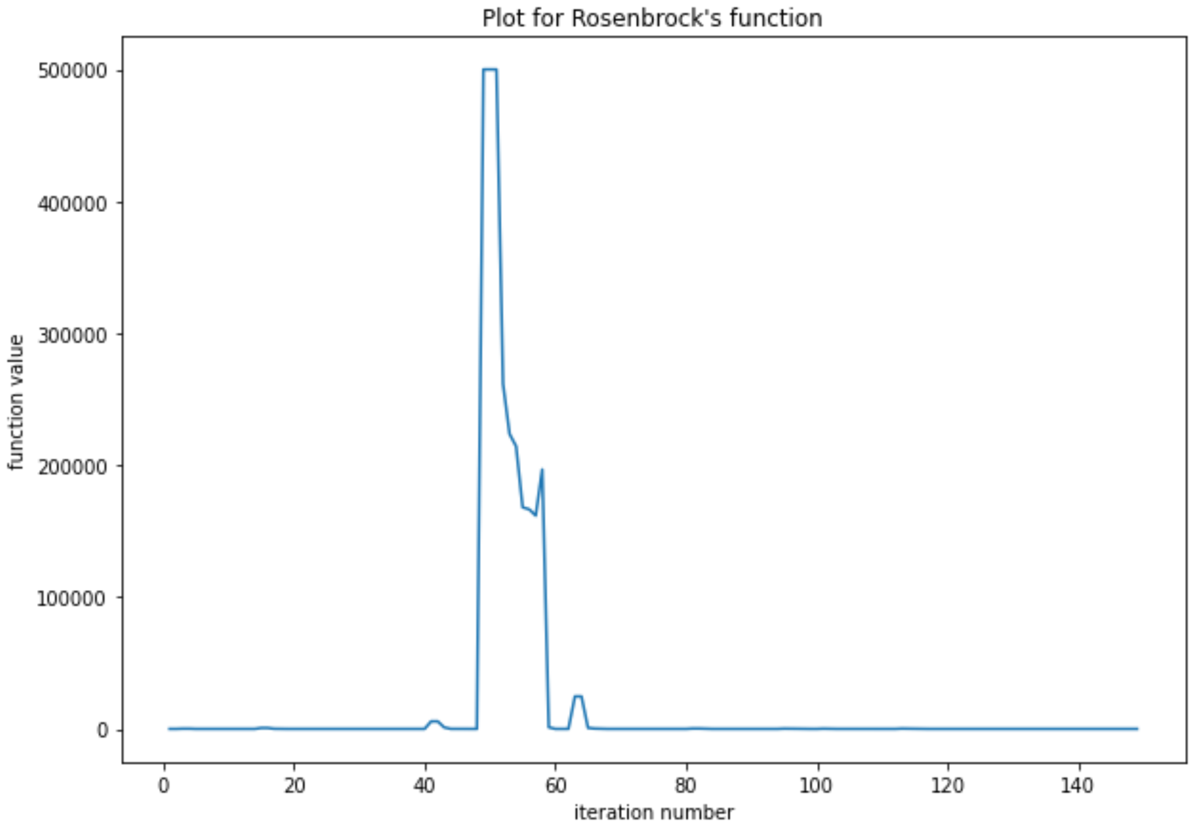


Rosenbrock's function

$f(\mathbf{x}) = (a - x_1)^2 + b(x_2 - x_1^2)^2$ with global minimum $f(\mathbf{x}^*) = 0$ at $\mathbf{x}^* = (a, a^2)$. Usually $a = 1$ and $b = 100$ [1].

In [72]:
```
def rosenbrock(x):
    a = 2
    b = 100
    return (a-x[0])**2+b*(x[1]-x[0]**2)**2

n = 2
eps = 1e-3
seed = None
alpha_hat = 1e-3
print("Got args: n = {}, eps = {}, seed = {}, alpha_hat = {}".format(
    n,
    eps,
    seed,
    alpha_hat,
    ))
#function = rosenbrock(n)
x, niter, fs = conjugate_gradient_general(
        rosenbrock,
        n=n,
        eps=eps,
        seed=seed,
        alpha_hat=alpha_hat)
print(x, niter)
figure = plt.figure(figsize=(10,7))
plt.plot(range(1, niter + 2),fs,label='Function value')
plt.xlabel('iteration number')
plt.ylabel('function value')
_=plt.title('Plot for Rosenbrock\'s function')
```

```
Got args: n = 2, eps = 0.001, seed = None, alpha_hat = 0.001
starting x:
[[-0.19143381]
 [ 0.16667605]]

[[1.99964853]
 [3.99859343]] 148
```


Plot for Rosenbrock's function
500000
400000
300000
200000
100000
0
function value
0
20
40
60
80
100
120
140
iteration number