In [1]:
```
import numpy as np
import numpy.linalg as la

import scipy.optimize as sopt
from scipy.optimize import minimize

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
%matplotlib inline
```

Numerical Optimization

1) Example of a constrained optimization problem

Suppose we want to maximize the area of a rectangle with sides `d1` and `d2`.

We store these design variables in the array `x = np.array([d1,d2])`.

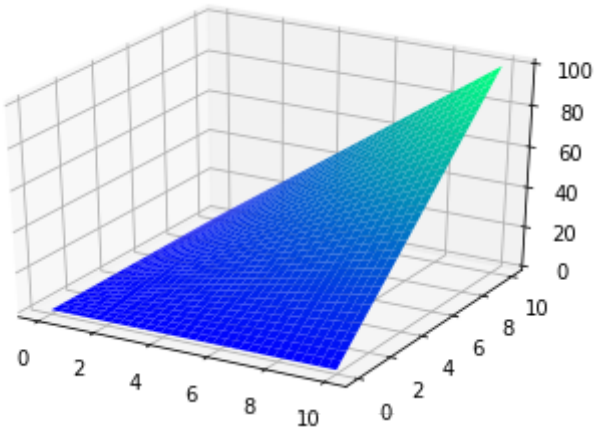
We add to this problem a constraint that the perimeter should be less than a given quantity. The functions below evaluate the area and perimeter for input variable `x`.

In [2]:
```
def A(x):
    d1 = x[0]
    d2 = x[1]
    return d1*d2


def P(x):
    d1 = x[0]
    d2 = x[1]
    return 2*(d1+d2)
```

We can visualize how the area and perimeter change with `x` using 3d plots:

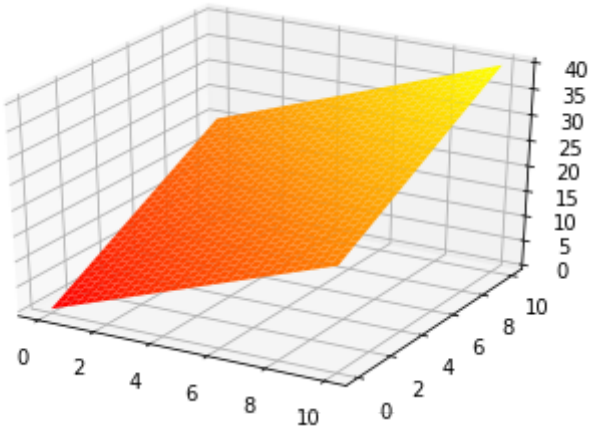
In [3]:
```
fig = plt.figure()
ax = fig.gca(projection="3d")

xmesh, ymesh = np.mgrid[0:10:100j,0:10:100j]
AMesh = A(np.array([xmesh, ymesh]))
ax.plot_surface(xmesh, ymesh, AMesh, cmap=plt.cm.winter, rstride=3, cstr
ide=3)
```

Out[3]:
```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x10232928d0>
```



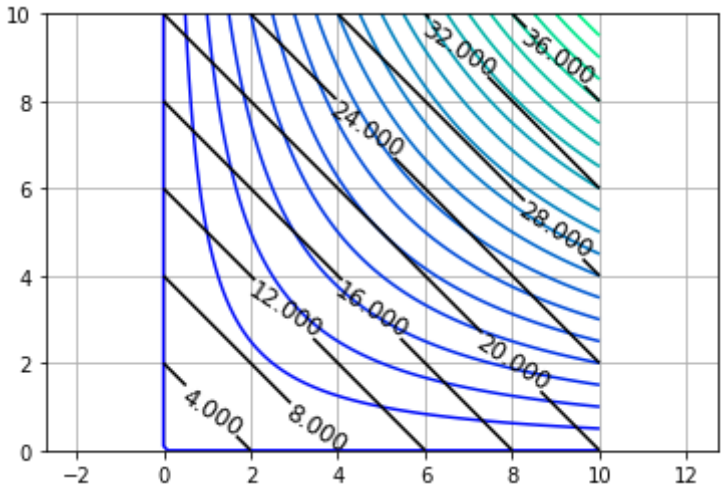
In [4]:
```
fig2 = plt.figure()
ax = fig2.gca(projection="3d")

PMesh = P(np.array([xmesh, ymesh]))
ax.plot_surface(xmesh, ymesh, PMesh,  cmap=plt.cm.autumn, rstride=3, cst
ride=3)
```

Out[4]:
```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x10235b59d0>
```



In [5]:
```
plt.axis("equal")
plt.grid()
figc1 = plt.contour(xmesh, ymesh, AMesh, 20, cmap=plt.cm.winter)
figc2 = plt.contour(xmesh, ymesh, PMesh, 10, colors='k')
plt.clabel(figc2, inline=1, fontsize=12)
```

Out[5]:
```
<a list of 9 text.Text objects>
```



Let's say we want to solve:

```
max A
st  P <= 20
```

or to use with minimize function

```
min f = -A
st g = 20 - P >= 0
```

In [6]:
```
#Initial guess
x0 = np.array([9,1])
f = lambda x: -A(x)
g = lambda x: 20 - P(x)
minimize(f, x0, constraints=({'type': 'ineq', 'fun': lambda x: g(x)}))
```

Out[6]:
```
      fun: -24.999999999999815
      jac: array([-4.99999976, -4.99999976])
  message: 'Optimization terminated successfully.'
     nfev: 8
      nit: 2
     njev: 2
   status: 0
  success: True
        x: array([5., 5.])
```

2) 1D Optimization Methods

In this activity, we will find the optimizer of functions in 1d using two iterative methods. For each one, you should be thinking about cost and convergence rate.

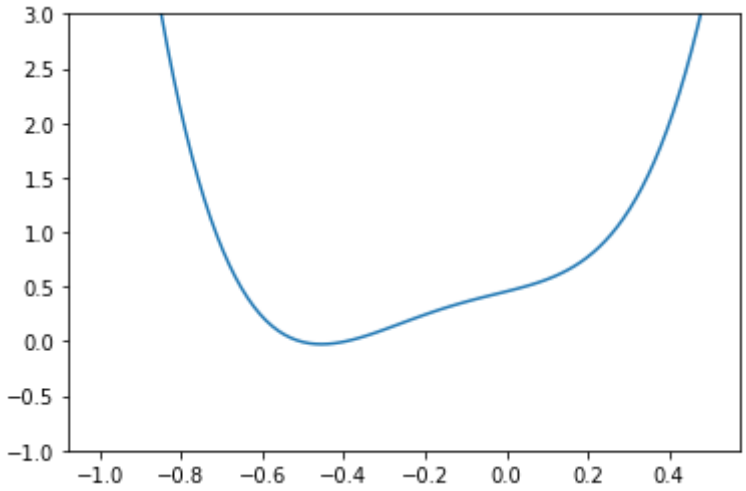
The iterative methods below can be applied to more complex equations, but here we will use a simple polynomial function of the form:

$$f(x) = ax^4 + bx^3 + cx^2 + dx + e$$

The code snippet below provides the values for the constants, and functions to evaluate $f(x)$, $f'(x)$ and $f''(x)$.

In [7]:
```
a = 17.09
b = 9.79
c = 0.6317
d = 0.9324
e = 0.4565

def f(x):
    return a*x**4 + b*x**3 + c*x**2 + d*x + e

def df(x):
    return 4*a*x**3 + 3*b*x**2 + 2*c*x + d

def d2f(x):
    return 3*4*a*x**2 + 2*3*b*x + 2*c

xmesh = np.linspace(-1, 0.5, 100)
plt.ylim([-1, 3])
plt.plot(xmesh, f(xmesh))
```

Out[7]:
```
[<matplotlib.lines.Line2D at 0x102285d710>]
```



a) Golden Section Search

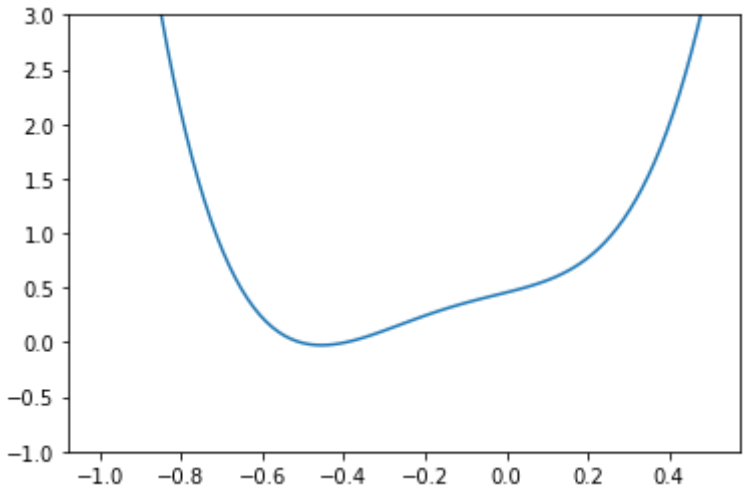
In [8]:
```
tau = (np.sqrt(5)-1)/2

a0 = -0.9 #-2
b0 = -0.2 #1

h_k = b0 - a0

x1 = a0 + (1-tau) * h_k
x2 = a0 + tau * h_k
print(x1,x2)
f1 = f(x1)
f2 = f(x2)

errors = [np.abs(h_k)]
count = 0

while (count < 30 and np.abs(h_k) > 1e-6):

    if  f1>f2:
        a0 = x1
        x1 = x2
        f1 = f2
        h_k = b0-a0
        x2 = a0 + tau * h_k
        f2 = f(x2)
    else:
        b0 = x2
        x2 = x1
        f2 = f1
        h_k = b0-a0
        x1 = a0 + (1-tau) * h_k
        f1 = f(x1)
    errors.append(np.abs(h_k))

    print("%10g \t %10g \t %12g %12g" % (a0, b0, errors[-1], errors[-1]/
errors[-2]))
```

```
-0.6326237921249265 -0.4673762078750736
 -0.632624              -0.2          0.432624     0.618034
 -0.632624         -0.365248          0.267376     0.618034
 -0.530495         -0.365248          0.165248     0.618034
 -0.530495         -0.428367          0.102129     0.618034
 -0.491486         -0.428367          0.063119     0.618034
 -0.467376         -0.428367         0.0390097     0.618034
 -0.467376         -0.443267         0.0241093     0.618034
 -0.458167         -0.443267         0.0149004     0.618034
 -0.458167         -0.448958        0.00920893     0.618034
 -0.458167         -0.452476        0.00569143     0.618034
 -0.455993         -0.452476         0.0035175     0.618034
 -0.455993         -0.453819        0.00217393     0.618034
 -0.455993          -0.45465        0.00134357     0.618034
  -0.45548          -0.45465       0.000830369     0.618034
 -0.455163          -0.45465       0.000513196     0.618034
 -0.455163         -0.454846       0.000317173     0.618034
 -0.455042         -0.454846       0.000196024     0.618034
 -0.454967         -0.454846       0.000121149     0.618034
 -0.454967         -0.454892       7.48743e-05     0.618034
 -0.454938         -0.454892       4.62749e-05     0.618034
 -0.454938          -0.45491       2.85994e-05     0.618034
 -0.454927          -0.45491       1.76754e-05     0.618034
 -0.454921          -0.45491        1.0924e-05     0.618034
 -0.454921         -0.454914       6.75141e-06     0.618034
 -0.454921         -0.454917        4.1726e-06     0.618034
 -0.454919         -0.454917       2.57881e-06     0.618034
 -0.454919         -0.454917       1.59379e-06     0.618034
 -0.454919         -0.454918       9.85018e-07     0.618034
```

b) Newton's method in 1D

In [9]:
```
plt.ylim([-1, 3])
plt.plot(xmesh, f(xmesh))
```

Out[9]:
```
[<matplotlib.lines.Line2D at 0x1023733510>]
```



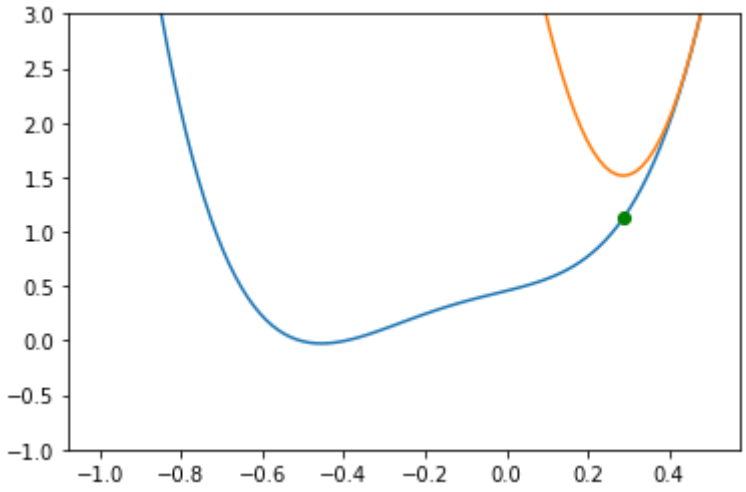
Let's fix an initial guess:

In [10]:
```
x_exact = -0.4549
```

In [11]:
```
x = 0.5
```

In [12]:
```
dfx = df(x)
d2fx = d2f(x)

# carry out the Newton step
xnew = x - dfx / d2fx

# plot approximate function
plt.plot(xmesh, f(xmesh))
plt.plot(xmesh, f(x) + dfx*(xmesh-x) + d2fx*(xmesh-x)**2/2)
plt.plot(x, f(x), "o", color="red")
plt.plot(xnew, f(xnew), "o", color="green")
plt.ylim([-1, 3])

# update
x = xnew
print(x)
```

```
0.2869245965368959
```



In [13]:
```
x = -0.1 #0.2
for i in range(30):

    dfx = df(x)
    d2fx = d2f(x)
    xnew = x - dfx / d2fx
    if np.abs(xnew-x) < 1e-8:
        break
    print(" %i %10g " % (i,x) )
    x = xnew
```

```
0       -0.1
1   0.302922
2   0.146876
3 0.00938739
4  -0.507232
5  -0.462945
6  -0.455151
7  -0.454919
```

c) Using scipy library

In [14]:
```
import scipy.optimize as sopt
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html (https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html)

In [15]:
```
sopt.minimize?
```

In [16]:
```
x0 = 2
sopt.minimize(f, x0)
```

Out[16]:
```
      fun: -0.02668057317931294
 hess_inv: array([[0.05883205]])
      jac: array([3.7252903e-08])
  message: 'Optimization terminated successfully.'
     nfev: 51
      nit: 13
     njev: 17
   status: 0
  success: True
        x: array([-0.45491836])
```

In [17]:
```
sopt.golden(f,brack=(-8,2))
```

Out[17]:
```
-0.454918360046204
```

3) ND Optimization Methods

We provide three example of functions. You will be able to observe difference convergence carachteristics among them.

Function 1:

$$f(x, y) = 0.5x^2 + 2.5y^2$$

In [18]:
```
def f1(x):
    return 0.5*x[0]**2 + 2.5*x[1]**2

def df1(x):
    return np.array([x[0], 5*x[1]])

def ddf1(x):
    return np.array([
                     [1,0],
                     [0,5]
                     ])
```

Function 2:

$$f(x, y) = (x - 1)^2 + (y - 1)^2$$

In [19]:
```
def f2(x):
    return (x[0]-1)**2 + (x[1]-1)**2

def df2(x):
    return np.array([2*(x[0]-1),2*(x[1]-1) ])

def ddf2(x):
    return np.array([
                     [2,0],
                     [0,2]
                     ])
```

Function 3:

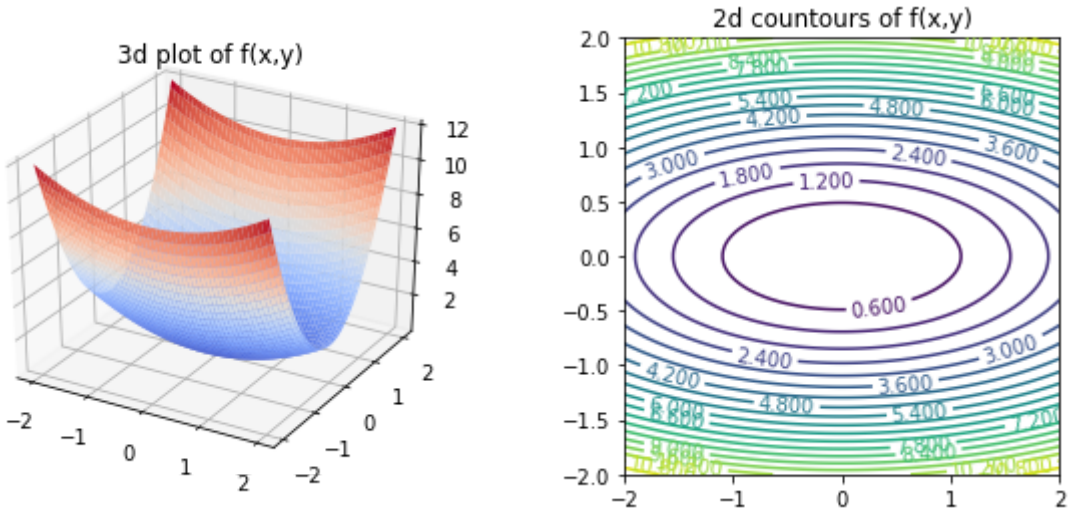
$$f(x, y) = 100(y - x^2)^2 + (1 - x)^2$$

In [20]:
```
def f3(X):
    x = X[0]
    y = X[1]
    val = 100.0 * (y - x**2)**2 + (1.0 - x)**2
    return val

def df3(X):
    x = X[0]
    y = X[1]
    val1 = -400.0 * (y - x**2) * x - 2 * (1 - x)
    val2 = 200.0 * (y - x**2)
    return np.array([val1, val2])

def ddf3(X):
    x = X[0]
    y = X[1]
    val11 = -400.0 * (y - x**2) + 800.0 * x**2 + 2
    val12 = -400.0 * x
    val21 = -400.0 * x
    val22 = 200.0
    return np.array([[val11, val12], [val21, val22]])
```

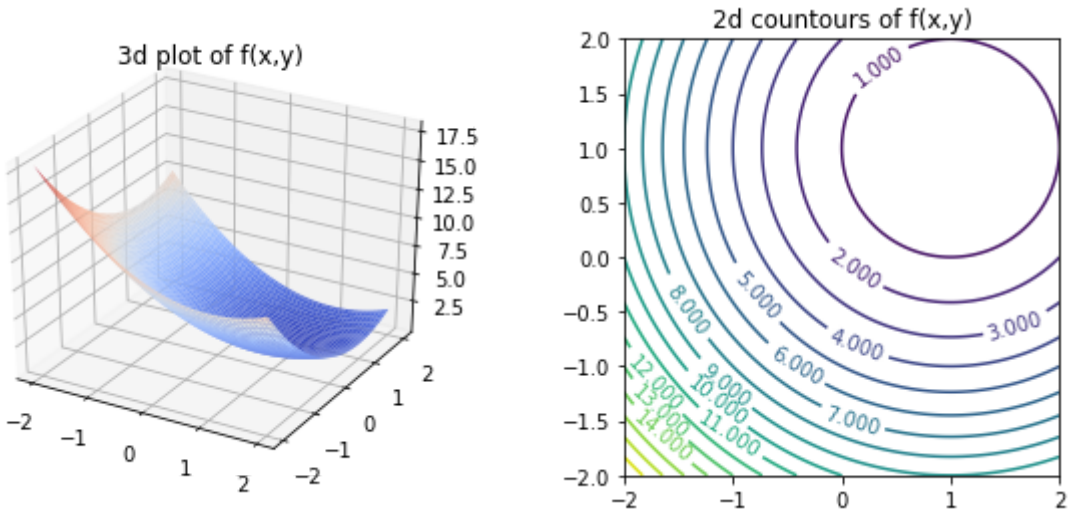
Helper functions for plotting

In [21]:
```
def plotFunction(f, interval=(-2,2), levels=20, steps=None, fhist=None):

    a,b = interval

    xmesh, ymesh = np.mgrid[a:b:100j,a:b:100j]
    fmesh = f(np.array([xmesh, ymesh]))


    fig = plt.figure(figsize=(16,4))

    ax = fig.add_subplot(131,projection="3d")
    ax.plot_surface(xmesh, ymesh, fmesh,cmap=plt.cm.coolwarm);
    plt.title('3d plot of f(x,y)')

    ax = fig.add_subplot(132)
    ax.set_aspect('equal')
    c = ax.contour(xmesh, ymesh, fmesh, levels=levels)

    plt.title('2d countours of f(x,y)')
    ax.clabel(c, inline=1, fontsize=10)

    if steps is not None:
        plt.plot(steps.T[0], steps.T[1], "o-", lw=3, ms=10)

    if fhist is not None:
        ax = fig.add_subplot(133)
        plt.semilogy(fhist, '-o')
        plt.xlabel('iteration')
        plt.ylabel('f')
        plt.grid()
```

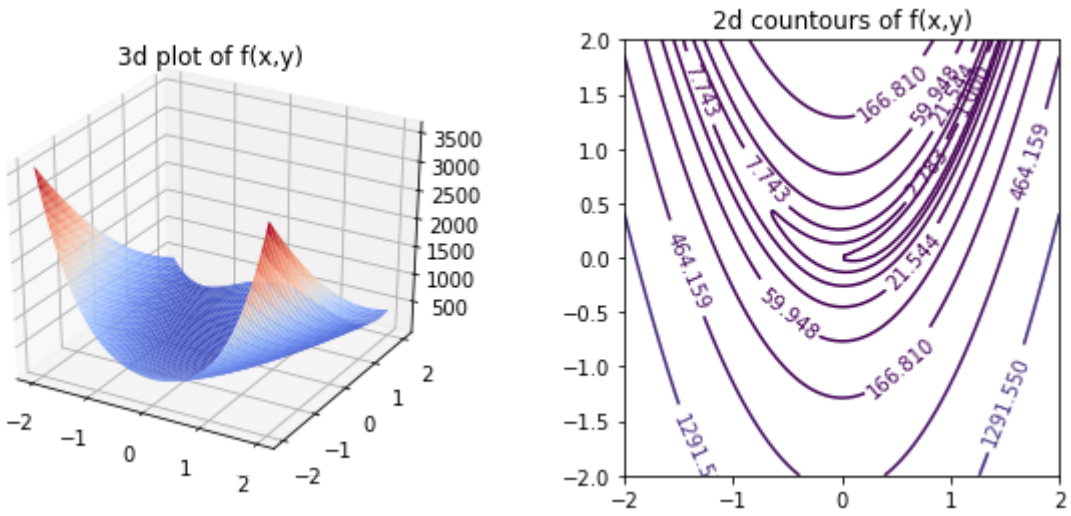
In [22]:
```
plotFunction(f1)
```



In [23]:
```
plotFunction(f2)
```



In [24]:
```
plotFunction(f3,levels=np.logspace(0,4,10))
```



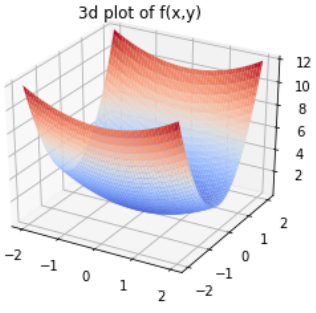
In [25]:
```
def plotConvergence( steps, exact , r ):

    error = la.norm(np.array(steps) - np.array(exact),axis=1)
    ratio = []
    for k in range(len(error)-1):
        ratio.append( error[k+1]/error[k]**r )

    fig = plt.figure(figsize=(4,4))

    plt.plot(ratio, "o-", lw=3, ms=10)
    plt.ylim(0,2)
```

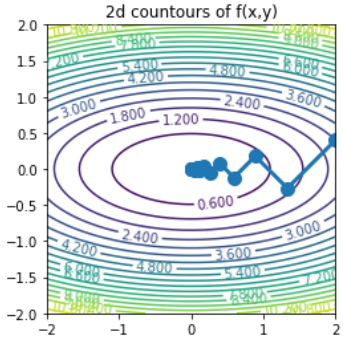
A) Steepest Descent

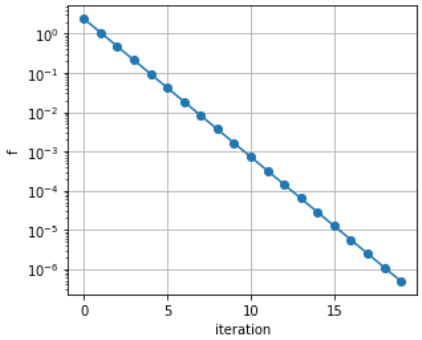
In [26]:
```
def steepestDescent(f,df,x0,maxiter,tol,alpha = 0):

    # Line search function
    def f_line(alpha):
        fnew = f(x + alpha*s)
        return fnew

    steps = [x0]
    x = x0
    fhist = [f(x)]

    # Steepest descent with line search
    for i in range(maxiter):

        # Get the gradient
        s = -df(x)

        # Learning rate:
        if alpha == 0:
            # Perform line search
            alpha_opt = sopt.golden(f_line)
        else:
            alpha_opt = alpha

        # Steepest descent update
        xnew = x + alpha_opt * s

        # Save optimized solution for plotting
        steps.append(xnew)

        fhist.append(f(xnew))

        # Check convergence

        if ( np.abs(fhist[-1] - fhist[-2]) < tol ):
            break

        x = xnew

    print('optimal solution is:', x)

    return steps, fhist, i
```

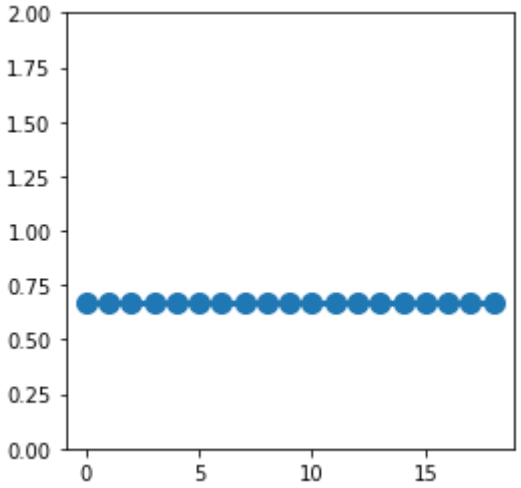
In [27]:
```
# Initial guess
x0 = np.array([2, 2./5])
# Steepest descent
steps, fhist, iterations = steepestDescent(f1,df1,x0,50,1e-6)
print('converged in', iterations, 'iterations')
# Plot convergence
plotFunction(f1,steps=np.array(steps),fhist=np.array(fhist))

plotConvergence( steps, [0,0] , 1 )
```

```
optimal solution is: [0.00135328 0.00027066]
converged in 18 iterations
```



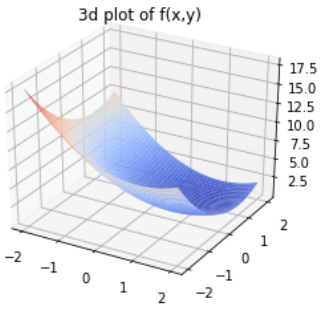


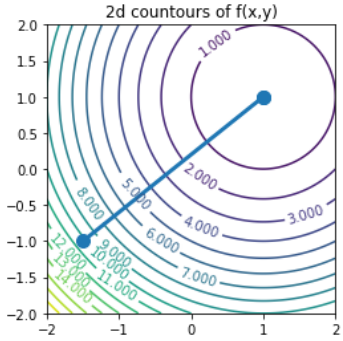


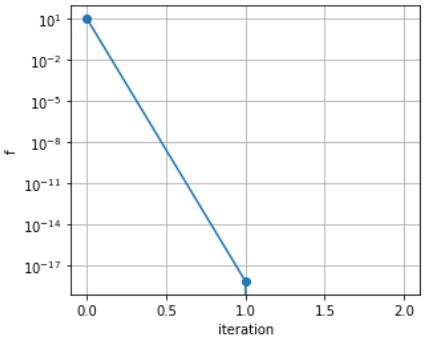


In [28]:
```
# Initial guess
x0 = np.array([-1.5, -1])
# Steepest descent
steps, fhist, iterations = steepestDescent(f2,df2,x0,50,1e-4)
print('converged in', iterations, 'iterations')
# Plot convergence
plotFunction(f2,steps=np.array(steps),fhist=np.array(fhist))
```

```
optimal solution is: [1. 1.]
converged in 1 iterations
```

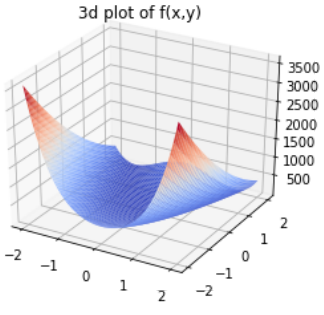


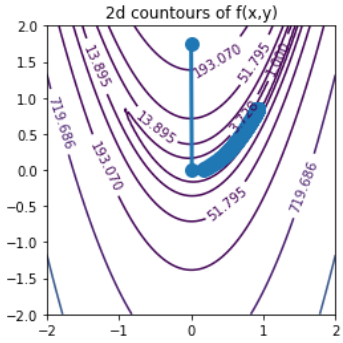


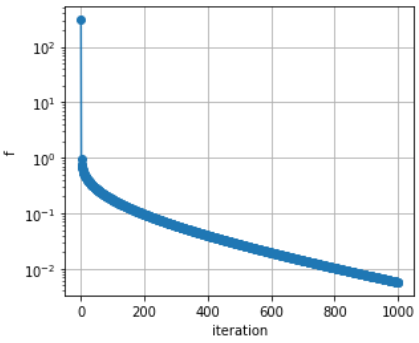


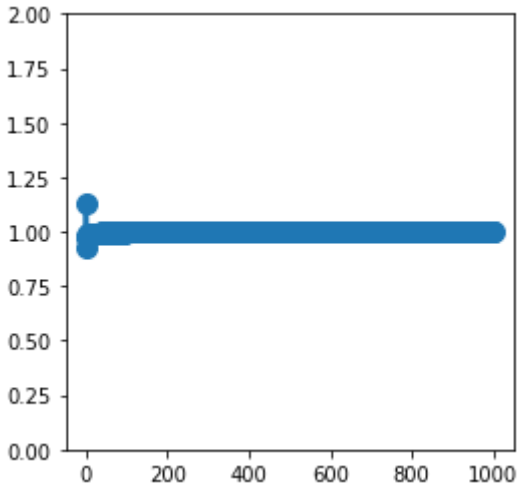
In [29]:
```
# Initial guess
x0 = np.array([0, 1.75])
# Steepest descent
steps, fhist, iterations = steepestDescent(f3,df3,x0,1000,1e-6)
print('converged in', iterations, 'iterations')
# Plot convergence
plotFunction(f3,steps=np.array(steps),levels=np.logspace(0,4,8), fhist=n
p.array(fhist))

plotConvergence( steps, [1 , 1] , 1 )
```

```
optimal solution is: [0.92446598 0.85422754]
converged in 999 iterations
```



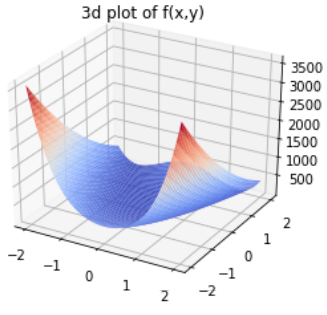


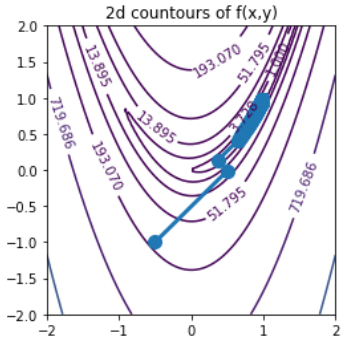


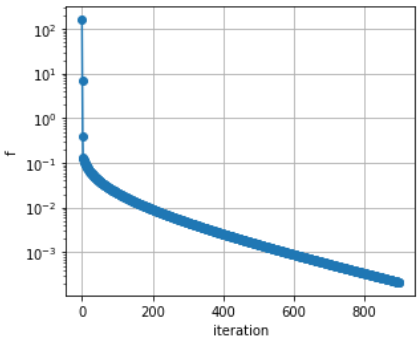


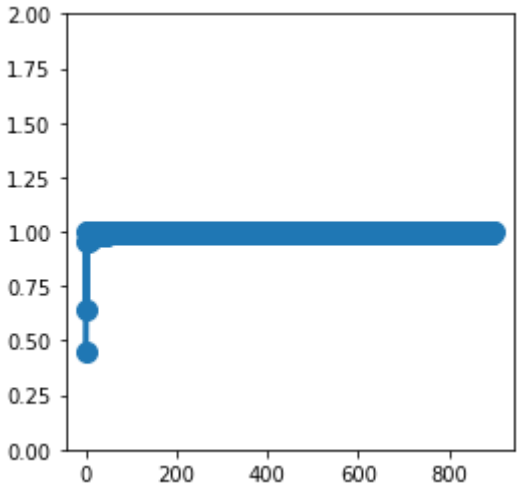
In [30]:
```
# Initial guess
x0 = np.array([-0.5, -1])
# Steepest descent
steps, fhist, iterations = steepestDescent(f3,df3,x0,1000,1e-6)
print('converged in', iterations, 'iterations')
# Plot convergence
plotFunction(f3,steps=np.array(steps),levels=np.logspace(0,4,8), fhist=n
p.array(fhist))

plotConvergence( steps, [1 , 1] , 1 )
```

```
optimal solution is: [0.98536754 0.97080029]
converged in 897 iterations
```







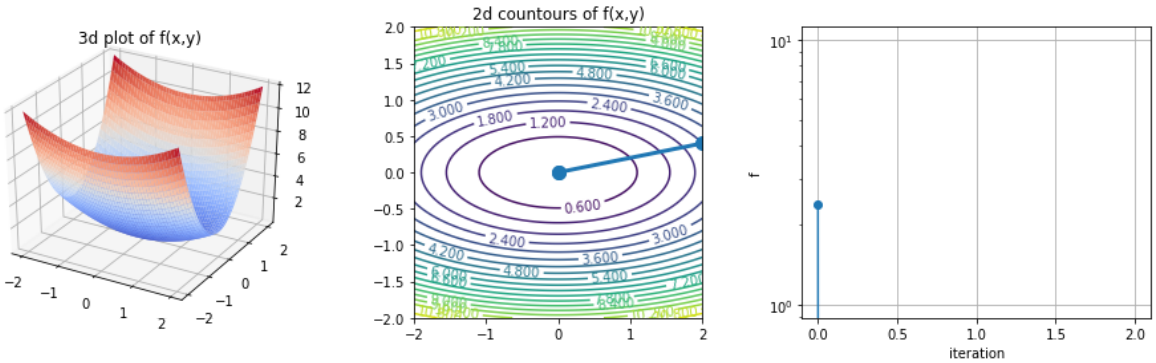


B) Newton's method

In [31]:
```
def NewtonMethod(f,df,ddf,x0,maxiter,tol):

    steps = [x0]
    x = x0
    fhist = [f(x)]

    # Steepest descent with line search
    for i in range(maxiter):

        # Get the newton step
        s = la.solve(ddf(x), -df(x))

        # Steepest descent update
        xnew = x + s

        # Save optimized solution for plotting
        steps.append(xnew)

        fhist.append(f(xnew))

        # Check convergence

        if ( np.abs(fhist[-1] - fhist[-2]) < tol ):
            break

        x = xnew

    print('optimal solution is:', x)

    return steps, fhist, i
```

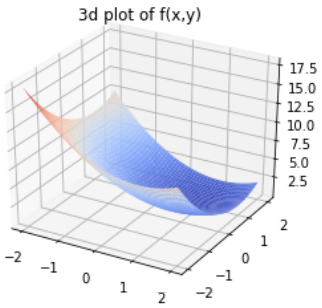
In [32]:
```
# Initial guess
x0 = np.array([2, 2./5])
# Newton's method
steps, fhist, iterations = NewtonMethod(f1,df1,ddf1,x0,50,1e-6)
print('converged in', iterations, 'iterations')
# Plot convergence
plotFunction(f1,steps=np.array(steps),fhist=np.array(fhist))
```

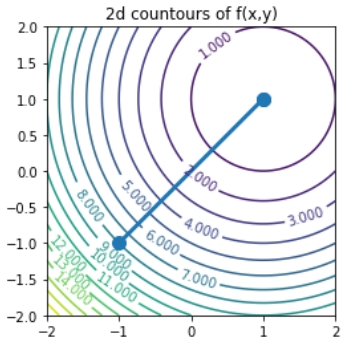
```
optimal solution is: [0. 0.]
converged in 1 iterations
```

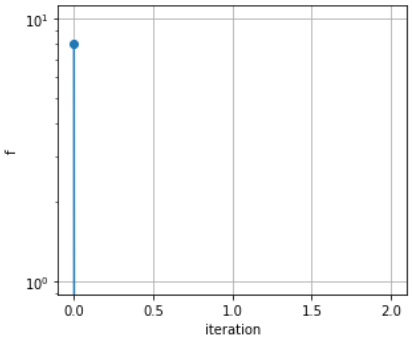


In [33]:
```
# Initial guess
x0 = np.array([-1, -1.0])
# Newton's method
steps, fhist, iterations = NewtonMethod(f2,df2,ddf2,x0,50,1e-6)
print('converged in', iterations, 'iterations')
# Plot convergence
plotFunction(f2,steps=np.array(steps),fhist=np.array(fhist))
```

```
optimal solution is: [1. 1.]
converged in 1 iterations
```

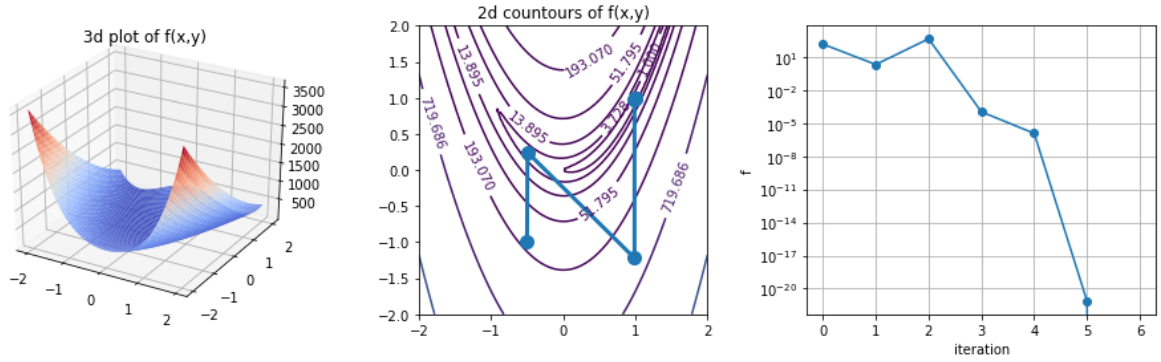


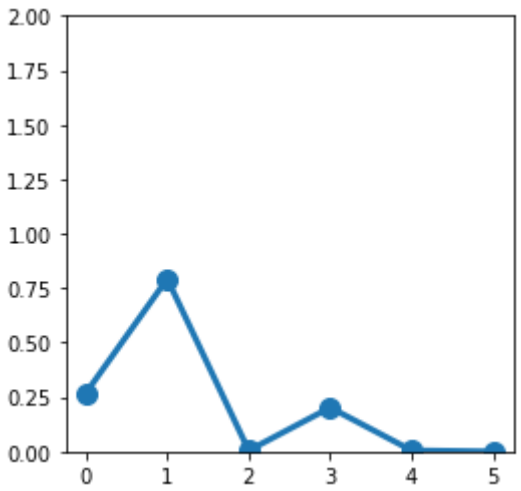




In [34]:
```
# Initial guess
x0 = np.array([-0.5, -1])
# Newton's method
steps, fhist, iterations = NewtonMethod(f3,df3,ddf3,x0,50,1e-8)
print('converged in', iterations, 'iterations')
# Plot convergence
plotFunction(f3,steps=np.array(steps),levels=np.logspace(0,4,8), fhist=n
p.array(fhist))

plotConvergence( steps, [1 , 1] , 2 )
```

```
optimal solution is: [1. 1.]
converged in 5 iterations
```





4) Location of Cities

[http://www.benfrederickson.com/numerical-optimization/ (http://www.benfrederickson.com/numerical-optimization/)](http://www.benfrederickson.com/numerical-optimization/)

In this example, given data on the distance between different cities, we want map out the cities by finding their locations in a 2-dimensional coordinate system.

Below is an example of distance data that we may have available that will allow us to map a list of cities.

Let's load the data that we will be working with in this example. `city_data` is an $n \times n$ numpy array where n is the number of cities. `city_data` will store the table of distances between cities similar to the one above.

In [35]:
```
city_data = np.load('additional_files/city_data.npy')
```

Before we start working with the data, we need to normalize the data by dividing by the largest element. This will allow us to more easily generate an initial guess for the location of each city.

In [36]:
```
city_data = city_data/np.max(city_data)
```

Below is a list of cities that we want to locate on a map.

In [37]:
```
city_names = [
    "Vancouver",
    "Portland",
    "New York",
    "Miami",
    "Mexico City",
    "Los Angeles",
    "Toronto",
    "Panama City",
    "Winnipeg",
    "Montreal",
    "San Francisco",
    "Calgary",
    "Chicago",
    "Halifax",
    "New Orleans",
    "Saskatoon",
    "Guatemala City",
    "Santa Fe",
    "Austin",
    "Edmonton",
    "Washington",
    "Phoenix",
    "Atlanta",
    "Seattle",
    "Denver"
  ]
```

Similar to the first example, the first step is to define the function that we need to minimize.

One way to formulate this problem is using the following loss function:

$$loss(\mathbf{X}) = \sum_i \sum_j ((\mathbf{X}_i - \mathbf{X}_j)^T(\mathbf{X}_i - \mathbf{X}_j) - D_{ij}^2)^2$$

- $\mathbf{X}_i$ and $\mathbf{X}_j$ are the positions for cities i and j. Each position $\mathbf{X}$ has two components, the x and y coordinates. **These are the variables we want to find!**
- $(\mathbf{X}_i - \mathbf{X}_j)^T(\mathbf{X}_i - \mathbf{X}_j)$ is the squared-distance between cities i and j, given the positions $\mathbf{X}_i$ and $\mathbf{X}_j$.
- D_{ij} is the known distance between cities i and j. **These are the given (known) variables provided in `city_data`.**

The loss function measures how much the actual location and the guess location differ. The optimization problem becomes:

$$\min_{\mathbf{X}} loss(\mathbf{X})$$

Assume that the location of cities is stored as `city_loc`, a 1D numpy array of size $2n$, such that the x-coordinate of a given city is stored first followed by it's y-coordinate.

I.e.

$$\text{city\_loc} = \begin{bmatrix} X_1[0] \\ X_1[1] \\ \vdots \\ X_n[0] \\ X_n[1] \end{bmatrix}$$

For example, if we had the cities Los Angeles, San Francisco and Chicago with their locations $(0.2, 0.1), (0.2, 0.5), (0.6, 0.7)$, respectively, then `city_loc` would be

$$\text{city\_loc} = \begin{bmatrix} 0.2 \\ 0.1 \\ 0.2 \\ 0.5 \\ 0.6 \\ 0.7 \end{bmatrix}.$$

The objective functin is defined below:

In [38]:
```
def loss(city_loc, city_data):
    totalLoss = 0.
    n = len(city_loc)//2

    for i in range(n):
        for j in range(n):
            xij = city_loc[2*i:2*i+2] - city_loc[2*j:2*j+2]
            totalLoss += ( np.inner(xij,xij) - city_data[i,j]**2)**2
    return totalLoss
```

Now that we have the function that we want to minimize, we need to compute it's gradient to use steepest descent.

$$\frac{\partial loss}{\partial X_k} = \sum_i -4((X_i - X_k)^T(X_i - X_k) - D_{ik}^2)(X_i - X_k) + \sum_j 4((X_k - X_j)^T(X_k - X_j)D_{kj}^2)(X_k - X_j)$$

In [39]:
```
def gradientLoss(city_loc, city_data):
    n = len(city_loc)
    grad = np.zeros(n)

    for k in range(n//2):
        for i in range(n//2):
            xik = city_loc[2*i:2*i+2] - city_loc[2*k:2*k+2]
            grad[2*k:2*k+2] += -4 * (np.dot(xik,xik) - city_data[i,k]**2
) * xik

        for j in range(n//2):
            xkj = city_loc[2*k:2*k+2] - city_loc[2*j:2*j+2]
            grad[2*k:2*k+2] += 4 * (np.dot(xkj,xkj) - city_data[k,j]**2)
* xkj
    return grad
```

We should now have everything that we need to use steepest descent if we use a learning rate instead of a line search parameter.

Steepest descent using learning rate

Write a function to run steepest descent for this problem.

- Store the solution after each iteration of steepest descent in a list called `city_loc_history`. To make plotting easier, we will reshape `city_loc` when we store it so that it is of shape $n \times 2$ instead of $2n$.
- Store the loss function after each iteration in a list called `loss_history`.
- Your algorithm should not exceed a given maximum number of iterations.
- In addition, you should add a convergence stopping criteria. Here assume that the change in the loss function from one iteration to the other should be smaller than a given tolerance `tol`.

```
def steepest_descent(city_loc, learning_rate, city_data, num_iterations, tol
):
    # compute num_iterations of steepest_descent
    city_loc_history = [city_loc.reshape(-1,2)]
    loss_history = []

    for i in range(num_iterations):

        # write step of steepest descent here

        loss_history.append( ... )

        city_loc_history.append(city_loc.reshape(-1,2))

        if (check if tolerance is reached):
            break

    return city_loc_history, loss_history
```

In [40]:
```
def steepest_descent(city_loc, learning_rate, city_data, num_iterations,
tol):
    city_history = [city_loc.reshape(-1,2)]
    loss_history = []
    for i in range(num_iterations):
        loss_history.append( loss(city_loc, city_data) )
        city_loc = city_loc - learning_rate * gradientLoss(city_loc, cit
y_data)
        city_history.append(city_loc.reshape(-1,2))
        if (i > 0):
            error = abs(loss_history[-2] - loss_history[-1])
            if ( error < tol ) :
                break
    return city_history, loss_history
```

Using your `steepest_descent` function, find the location of each city. Use a random initial guess for the location of each of the cities and use the following parameters.

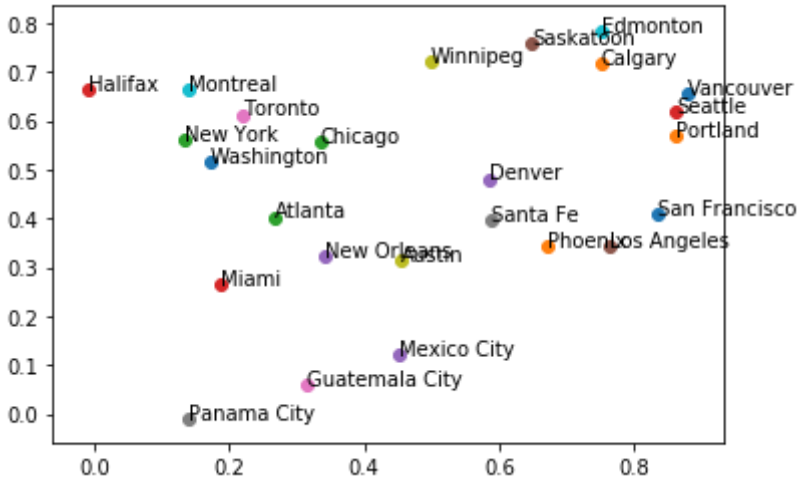
In [41]:
```
learning_rate = 0.005
num_iterations = 300
tol = 1e-8
city_loc = np.random.rand(2*city_data.shape[0])

city_loc_history, loss_history = steepest_descent(city_loc, learning_rat
e, city_data, num_iterations, tol)
```

We can use the `np.dstack` function to change the list of city locations into a numpy array. We will have a 3D numpy array with dimensions $n \times 2 \times num\_iterations$.

In [42]:
```
city_loc_history = np.dstack(city_loc_history)
```

Now let's display the location of each of the cities. Note, you can use `plt.text` to display the name of the city on the plot next to its location instead of using a legend. The following code snippet will plot the final location of the cities if we assume that we stored the result of steepest descent as `city_loc_history` .

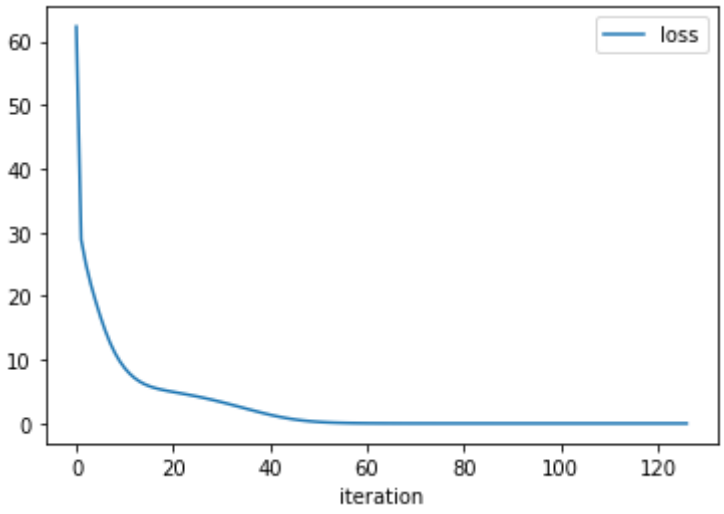
In [43]:
```
num_cities = city_loc_history.shape[0]

for i in range(num_cities):
    plt.scatter(city_loc_history[i,0,-1], city_loc_history[i,1,-1])
    plt.text(city_loc_history[i,0,-1], city_loc_history[i,1,-1], city_na
mes[i])
plt.show()
```



Does your plot make sense? Keep in mind that we aren't keeping track of orientation (we don't have a fixed point for the origin) so you may need to "rotate" or "invert" your plot (mentally) for it to make sense. Or if you want an extra challenge for after class, you can modify the optimization problem to include a fixed origin :-)

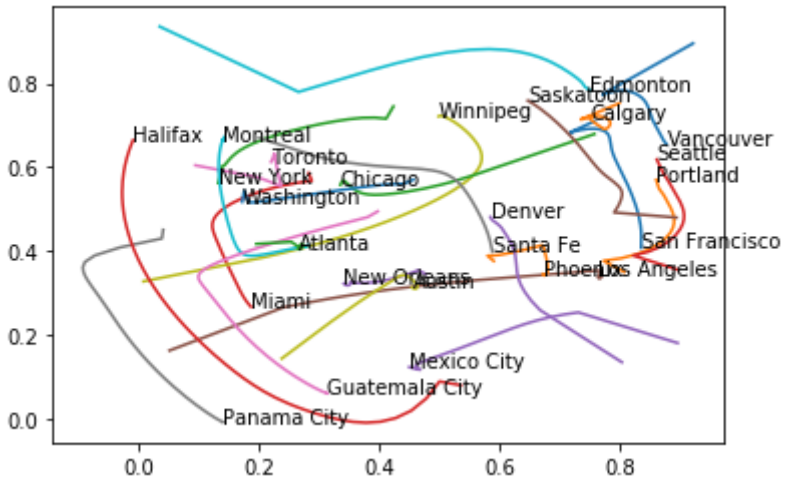
You can plot the history of the loss function, and observe the convergence of the optimization

In [44]:
```
plt.plot(loss_history, label = 'loss')
plt.xlabel('iteration')
plt.legend()
plt.show()
```



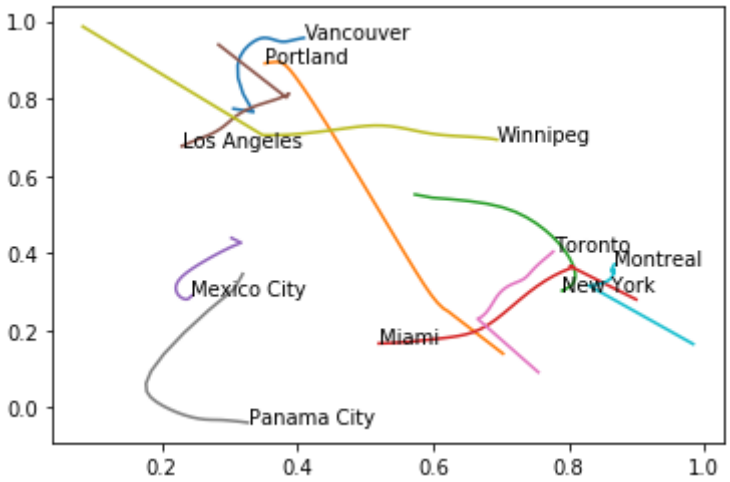
Let's also include how location of each of the cities changes throughout the optimization iterations. The following code snippet assumes that the output for steepest descent is stored in `city_loc_history` and is a numpy array of shape $n \times 2 \times num\_iterations$

In [45]:
```
num_cities = city_loc_history.shape[0]

for i in range(num_cities):
    plt.plot(city_loc_history[i,0,:], city_loc_history[i,1,:])
    plt.text(city_loc_history[i,0,-1], city_loc_history[i,1,-1], city_na
mes[i])
plt.show()
```



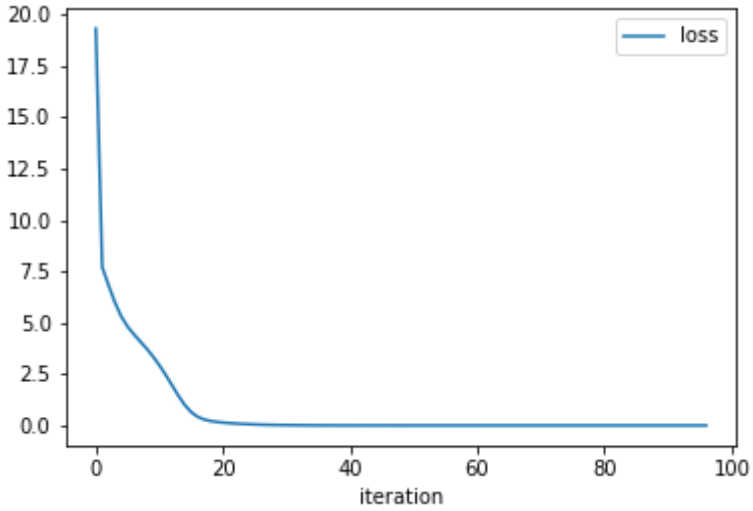
The plot looks too cluttered, right? Repeat the same experiment but use a smaller number of cities. But don't forget to normalize the smaller data set. Use 10 cities for this smaller experiment.

In [46]:
```
city_data2 = city_data[:10,:10]/np.max(city_data[:10,:10])
city_loc2 = np.random.rand(2*city_data2.shape[0])

learning_rate = 0.01
num_iterations = 300
tol = 1e-8

city_loc_hist2, loss_hist_2 = steepest_descent(city_loc2, learning_rate,
city_data2, num_iterations,tol)
city_loc_hist2 = np.dstack(city_loc_hist2)

num_cities = city_loc_hist2.shape[0]
for i in range(num_cities):
    plt.plot(city_loc_hist2[i,0,:], city_loc_hist2[i,1,:])
    plt.text(city_loc_hist2[i,0,-1], city_loc_hist2[i,1,-1], city_names[
i])
plt.show()
```



Let's see how the loss changes after each iteration. Plot the `loss_history` variable.

In [47]:
```
plt.plot(loss_hist_2, label = 'loss')
plt.xlabel('iteration')
plt.legend()
plt.show()
```



Repeat the same experiment but try different values for the learning rate. What happens when we decrease the learning rate? Do we need more iterations? What happens when we increase the learning rate? Do we need more or less iterations?

Steepest descent with golden-section search

Now, let's use golden-section search again for computing the line search parameter instead of a learning rate.

Wrap the function that we are minimzing so that α is a parameter.

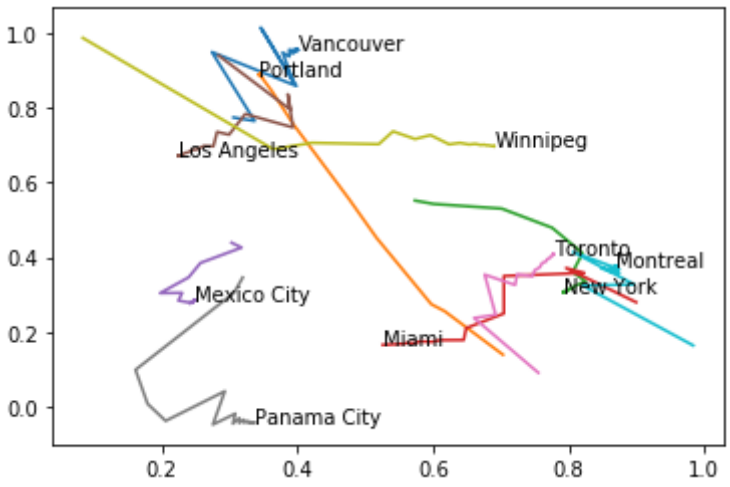
In [48]:
```
def objective_1d(alpha, city_loc, city_data):
    return loss(city_loc - alpha * gradientLoss(city_loc,city_data), cit
y_data)
```

Rewrite your steepest descent function so that it uses `scipy.optimize.golden` .

In [49]:
```
def sd_line(city_loc, city_data, num_iterations, tol):
    city_history = [city_loc.reshape(-1,2)]
    loss_history = []
    for i in range(num_iterations):

        alph = sopt.golden(objective_1d, args=(city_loc,city_data))

        loss_history.append( loss(city_loc, city_data) )

        city_loc = city_loc - alph * gradientLoss(city_loc, city_data)

        city_history.append(city_loc.reshape(-1,2))

        if (i > 0):
            error = abs(loss_history[-2] - loss_history[-1])
            if ( error < tol ) :
                break

    return city_history, loss_history
```

Use your new function for steepest descent with line search to find the location of the cities and plot the result. You can use the code snippets for plotting that we provided above.

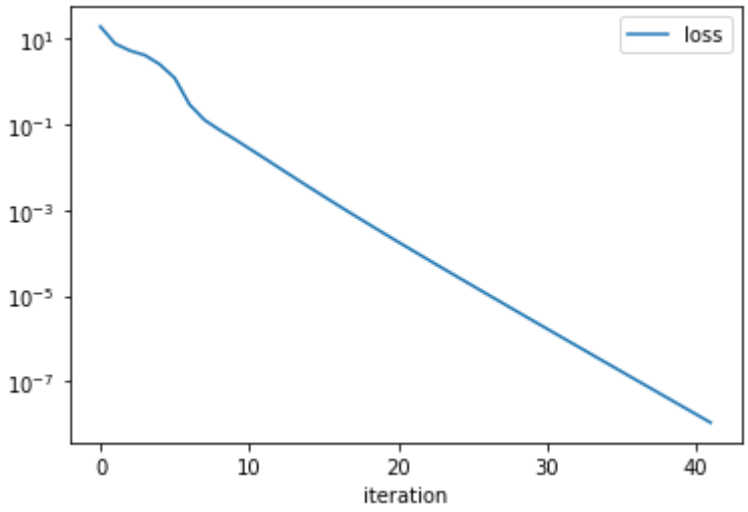
In [50]:
```
num_iterations = 300
tol = 1e-8

city_hist3, loss_hist3 = sd_line(city_loc2, city_data2, num_iterations,
tol)
city_hist3 = np.dstack(city_hist3)

num_cities = city_hist3.shape[0]

for i in range(num_cities):
    plt.plot(city_hist3[i,0,:], city_hist3[i,1,:])
    plt.text(city_hist3[i,0,-1], city_hist3[i,1,-1], city_names[i])
plt.show()
```



What do you notice about how the solution evolves? Do you expect using a line search method that the solution will converge faster or slower? Will using a line search method increase the cost per iteration of steepest descent?

Plot the loss function at each iteration to see if it converges in fewer number of iterations.

In [51]:
```
plt.semilogy(loss_hist3, label = 'loss')
plt.xlabel('iteration')
plt.legend()
plt.show()
```



Using scipy function

We can now compare the functions you defined above with the `minimize` function from `scipy`

In [52]:
```
import scipy.optimize as sopt

num_cities = 6
tol = 1e-5

city_data_small = city_data[:num_cities,:num_cities]/np.max(city_data[:n
um_cities,:num_cities])
x0 = np.random.rand(2*city_data_small.shape[0])
```

In [53]:
```
# Providing only the function to the optimization algorithm
# Note that it takes a lot more function evaluations for the optimizatio
n, since
# gradient and Hessians are approximated in the backend
res1 = sopt.minimize(loss, x0, args=(city_data_small), tol=tol )
xopt1 = res1.x.reshape(num_cities,2)
print(xopt1)
print('Optimized loss value is ', res1.fun)
print('converged in', res1.nit, 'iteration with', res1.nfev, 'function e
valuations' )
```

```
[[-0.22536158  0.43173626]
 [-0.1776976   0.53010238]
 [ 0.71500222  0.31302007]
 [ 0.74052802  0.69069069]
 [ 0.46333442  0.94884885]
 [ 0.01000607  0.77691277]]
Optimized loss value is  1.240295141398893e-13
converged in 26 iteration with 434 function evaluations
```

In [54]:
```
# Providing function and gradient to the optimization algorithm
# Note that the number of function evaluations are reduced, since only H
essian are now approximated
res2 = sopt.minimize(loss, x0, args=(city_data_small) , jac=gradientLoss
, tol = tol )
xopt2 = res2.x.reshape(num_cities,2)
print(xopt2)
print('Optimized loss value is ', res2.fun)
print('converged in', res2.nit, 'iteration with', res2.nfev, 'function e
valuations' )
```

```
[[-0.22535718  0.43174013]
 [-0.17769284  0.53010607]
 [ 0.71500618  0.3130205 ]
 [ 0.74053336  0.69069101]
 [ 0.4633407   0.9488502 ]
 [ 0.01001173  0.77691576]]
Optimized loss value is  1.1317397195054691e-13
converged in 26 iteration with 31 function evaluations
```

In [55]:
```
xhist,loss_hist = sd_line(x0, city_data_small, 400, tol)
xopt3 = xhist[-1]
print(xopt3)
print('Optimized loss value is ', loss_hist[-1])
print('convergend in ', len(loss_hist),'iterations')
```

```
[[ 0.76007754  0.70466142]
 [ 0.66809493  0.7656341 ]
 [ 0.01547857  0.11854334]
 [-0.2016352   0.43002325]
 [-0.09678166  0.79307366]
 [ 0.38060779  0.87938788]]
Optimized loss value is  8.940715307584716e-06
convergend in  28 iterations
```

In [56]:
```
xhist,loss_hist = steepest_descent(x0, 0.01,  city_data_small, 400, tol)
xopt3 = xhist[-1]
print(xopt3)
print('Optimized loss value is ', loss_hist[-1])
print('convergend in ', len(loss_hist),'iterations')
```

```
[[-2.10582072e-01  3.96242028e-01]
 [-1.71841942e-01  5.05115391e-01]
 [ 7.34482373e-01  3.44737807e-01]
 [ 7.34057664e-01  7.24814711e-01]
 [ 4.40381034e-01  9.61878321e-01]
 [-6.55099253e-04  7.58535399e-01]]
Optimized loss value is  9.110522246096597e-05
convergend in  83 iterations
```