Optimization

Computational Methods, Nov. 2017, Kenji Doya

Research involves lots of parmeter tuing. When there are just a few paramters, we often tune them by hand using our intuition or run a grid search. But when the number of parameters is large or it is difficult to get any intuition, we need a systematic method for optimization.

Optimization is in general considered as minimization or maximization of a certain objective function $f(x)$ where x is a parameter vector. There are different cases:

- If the mathematical form of the objective function $f(x)$ is known:
    - put derivatieves $\frac{\partial f(x)}{\partial x}=0$ and solve for x.
    - check the signs of second order derivatives $\frac{\partial^2 f(x)}{\partial x^2}$
        - if all positive, that is a minimum
        - if all negative, that is a maximum
        - if mixed, that is a saddle point
- If analytic solution of $\frac{\partial f(x)}{\partial x}=0$ is hard to derive:
    - gradient descent/ascent
    - Newton-Raphson method
    - conjugate gradient method
- If the derivatives of $f(x)$ is hard to derive:
    - genetic/evolutionary algorithms
    - sampling methods (next week)
- If $f(x)$ needs to be optimized under constraints, such as $g(x) \leq 0$ or $h(x)=0$:
    - penalty function
    - Lagrange multiplyer method
    - linear programming if $f(x)$ is linear
    - quadratic programming if $f(x)$ is quadratic

References:

- Jan A. Snyman: Practical Mathematial Optimization. Springer, 2005.
- SciPy Lecture Notes: 5.5 Optimization and fit
- SciPy Tutorial: 3.1.5 Optimization (scipy.optimize)

In [1]:
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 %matplotlib inline
4 from mpl_toolkits.mplot3d import Axes3D
```

Example

For the sake of visualization, consider a function in 2D space $x = (x_1, x_2)$

$$f(x) = x_1^4 - \frac{8}{3}x_1^3 - 6x_1^2 + x_2^4$$

The gradient is

$$\nabla f(x) = \frac{\partial f(x)}{\partial x} = \begin{pmatrix} 4x_1^3 - 8x_1^2 - 12x_1 \\ 4x_2^3 \end{pmatrix}.$$

By putting $\nabla f(x) = 0$, we have

$$x_1(x_1^2 - 2x_1 - 3) = x_1(x_1 + 1)(x_1 - 3) = 0$$
$$x_2^3 = 0,$$

so there are three points with zero gradient: $(-1, 0)$, $(0, 0)$, $(3, 0)$.

You can check the second-order derivative, or *Hessian*, to see if they are a minimum, a saddle point, or a maximum.

$$\nabla^2 f(x) = \frac{\partial^2 f(x)}{\partial x^2} = \begin{pmatrix} 12x_1^2 - 16x_1 - 12 & 0 \\ 0 & 12x_2^2 \end{pmatrix}.$$

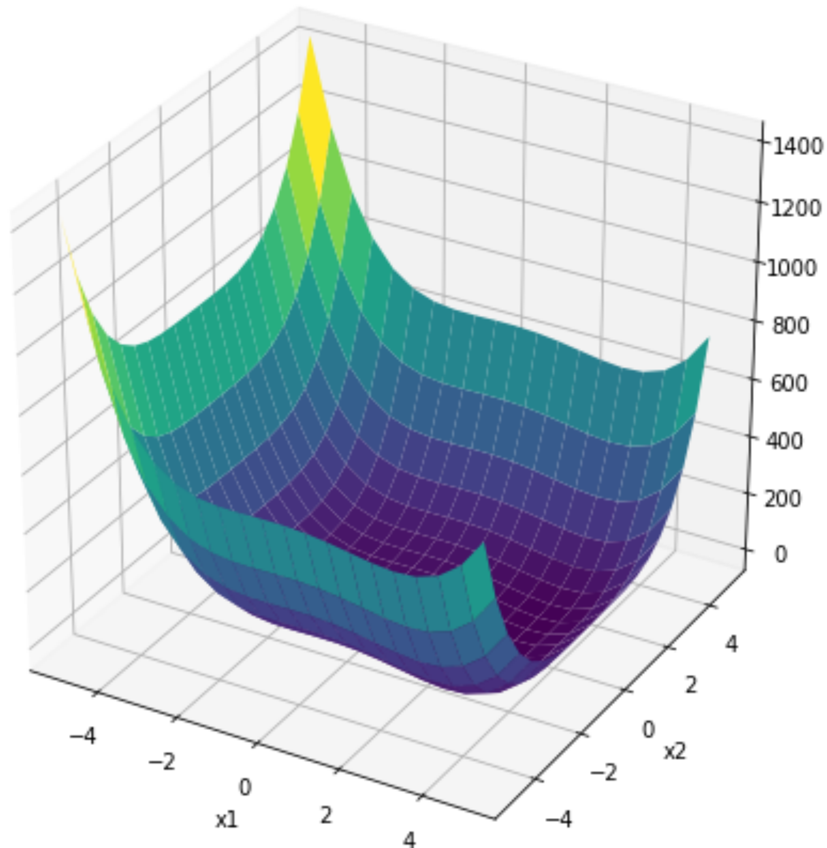
As $\frac{\partial^2 f(x)}{\partial x_1^2}$ is positive for $x_1 = -1$ and $x_1 = 3$ and negative for $x_1 = 0$, $(-1, 0)$ and $(3, 0)$ are minima and $(0, 0)$ is a saddle point.

Let us visualize this.

In [2]:
```
1 def dips(x):
2     """a function to minimize"""
3     f = x[0]**4 - 8/3*x[0]**3 - 6*x[0]**2 + x[1]**4
4     return(f)
5 
6 def dips_grad(x):
7     """gradient of dips(x)"""
8     df1 = 4*x[0]**3 - 8*x[0]**2 - 12*x[0]
9     df2 = 4*x[1]**3
10     return(np.array([df1, df2]))
11 
12 def dips_hess(x):
13     """hessian of dips(x)"""
14     df11 = 12*x[0]**2 - 16*x[0] - 12
15     df12 = 0
16     df22 = 12*x[1]**2
17     return(np.array([[df11, df12], [df12, df22]]))
```

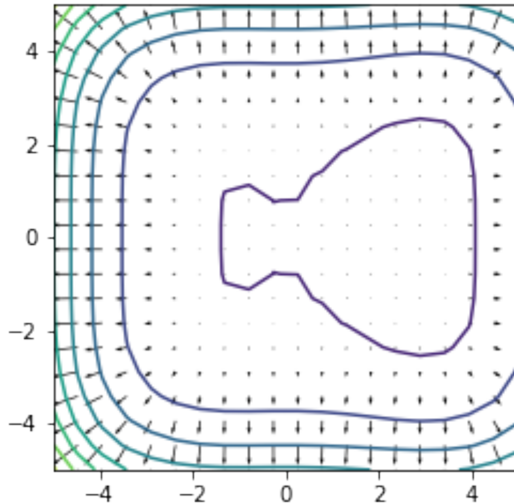
In [3]:
```
1 x1, x2 = np.meshgrid(np.linspace(-5, 5, 20), np.linspace(-5, 5, 20
2 fx = dips([x1, x2])
3 # 3D plot
4 fig = plt.figure(figsize=(8, 8))
5 ax = fig.add_subplot(111, projection='3d')
6 ax.plot_surface(x1, x2, fx, cmap='viridis')
7 ax.set_xlabel('x1')
8 ax.set_ylabel('x2')
```

Out[3]: Text(0.5, 0, 'x2')



In [4]:
```
1 plt.contour(x1, x2, fx)
2 dfx = dips_grad([x1, x2])
3 plt.quiver(x1, x2, dfx[0], dfx[1])
4 plt.axis('square')
```

Out[4]: (-5.0, 5.0, -5.0, 5.0)



In [5]:
```
1 plt.contour?
```

In [ ]:
```
1
```

Gradient Descent/Ascent

Gradient descent/ascent is the most basic method of min/maximization of a function using its gradient.

From an initial state x_0 and a coefficient $\eta > 0$, repeat

$$x_{i+1} = x_i - \eta \nabla f(x_i)$$

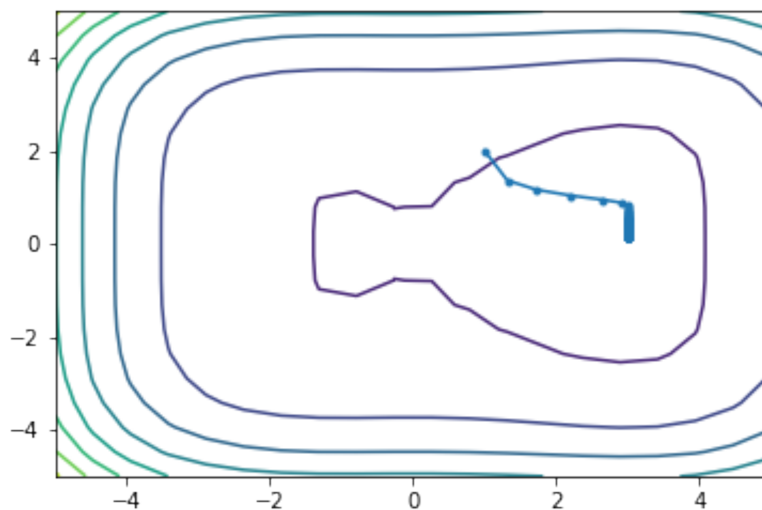
for minimization.

In [6]:
```
def grad_descent(f, df, x0, eta=0.01, eps=1e-6, imax=1000):
    """Gradient descent"""
    xh = np.zeros((imax+1, len(np.ravel([x0]))))  # history
    xh[0] = x0
    f0 = f(x0)  # initialtization
    for i in range(imax):
        x1 = x0 - eta*df(x0)
        f1 = f(x1)
        # print(x1, f1)
        xh[i+1] = x1
        if(f1 <= f0 and f1 > f0 - eps):  # small decrease
            return(x1, f1, xh[:i+2])
        x0 = x1
        f0 = f1
    print("Failed to converge in ", imax, " iterations.")
    return(x1, f1, xh)
```

In [7]:
```
xmin, fmin, xhist = grad_descent(dips, dips_grad, [1,2], 0.02)
print(xmin, fmin)
plt.contour(x1, x2, fx)
plt.plot(xhist[:,0], xhist[:,1], '.-')
#plt.axis([1, 4, -1, 3])
```

```
[3.         0.1206838] -44.99978787303231
```

Out[7]: `[<matplotlib.lines.Line2D at 0x10fe134a8>]`



In [ ]:
```

```

Newton-Raphson Method

A problem with the gradient descence/ascent is the choice of the coefficient η. If the second-order derivative, called the *Hessian*,

$$\nabla^2 f(x) = \frac{\partial^2 f}{\partial x^2}$$

is available, we can use the Newton method to find the solution for $\nabla f(x) = \frac{\partial f}{\partial x} = 0$ by repeating

$$x_{i+1} = x_i - \nabla^2 f(x_i)^{-1} \nabla f(x_i).$$

This is called Newton-Raphson method. It works efficiently when the Hessian is positive definite ($f(x)$ is like a parabolla), but can be unstable when the Hessian has a negative eigenvalue (near the saddle point).

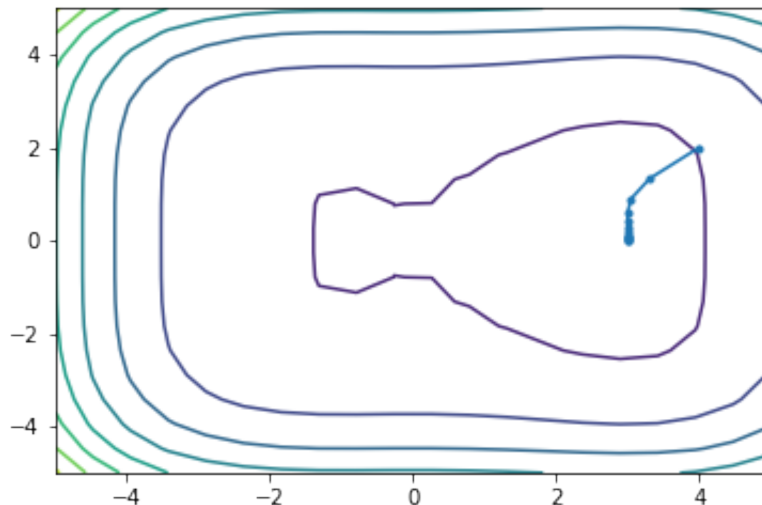
In [8]:

```
def newton_raphson(f, df, d2f, x0, eps=1e-6, imax=1000):
    """Newton-Raphson method"""
    xh = np.zeros((imax+1, len(np.ravel([x0]))))  # history
    xh[0] = x0
    f0 = f(x0)  # initialtization
    for i in range(imax):
        x1 = x0 - np.linalg.inv(d2f(x0)) @ df(x0)
        f1 = f(x1)
        #print(x1, f1)
        xh[i+1] = x1
        if( f1 <= f0 and f1 > f0 - eps):  # decreasing little
            return(x1, f1, xh[:i+2])
        x0 = x1
        f0 = f1
    print("Failed to converge in ", imax, " iterations.")
    return(x1, f1, xh)
```

In [9]:
```
1 xmin, fmin, xhist = newton_raphson(dips, dips_grad, dips_hess, [4,2
2 print(xmin, fmin)
3 plt.contour(x1, x2, fx)
4 plt.plot(xhist[:,0], xhist[:,1], '.-')
5 #plt.axis([1, 4, -1, 3])
```

```
[3.          0.01541469] -44.999999943540175
```

Out[9]:
```
[<matplotlib.lines.Line2D at 0x10fe67c50>]
```



In [ ]:
```
1
```

scipy.optimize

To address those issues, advanced optimization algorithms have been developed and implemented in `scipy.optimize` package.

In [10]:
```
1 from scipy.optimize import minimize
```

- The default method for unconstrained minimization is 'BFGS' (Broyden-Fletcher-Goldfarb-Shanno) method, a variant of gradient descent.

In [11]:
```
1 result = minimize(dips, [-1,2], jac=dips_grad, options={'disp': Tru
2 print( result.x, result.fun)
```

```
Optimization terminated successfully.
         Current function value: -2.333333
         Iterations: 17
         Function evaluations: 18
         Gradient evaluations: 18
[-1.          0.01205015] -2.3333333122484903
```

If the gradient function is not specified, it is estimated by finite difference method.

In [12]:
```
result = minimize(dips, [2,2], options={'disp': True})
print( result.x, result.fun)
```

```
Optimization terminated successfully.
         Current function value: -45.000000
         Iterations: 19
         Function evaluations: 84
         Gradient evaluations: 21
[2.99999999 0.01062343] -44.999999987263244
```

In [13]:
```
minimize?
```

- 'Newton-CG' (Newton-Conjugate-Gradient) is a variant of Newton-Raphson method using linear search in a *conjugate* direction.

In [14]:
```
result = minimize(dips, [2,2], method='Newton-CG',
          jac=dips_grad, hess=dips_hess, options={'disp': True})
print( result.x, result.fun)
```

```
Optimization terminated successfully.
         Current function value: -45.000000
         Iterations: 16
         Function evaluations: 18
         Gradient evaluations: 33
         Hessian evaluations: 16
[3.        0.0065082] -44.999999998205915
```

In [15]:
```
result
```

Out[15]:
```
     fun: -44.999999998205915
     jac: array([0.00000000e+00, 1.10266218e-06])
 message: 'Optimization terminated successfully.'
    nfev: 18
    nhev: 16
     nit: 16
    njev: 33
  status: 0
 success: True
       x: array([3.       , 0.0065082])
```

- 'Nelder-Mead' is a *simplex* method that uses a set of $n+1$ points to estimate the gradient and select a new point by flipping the simplex.
    - note that it is totally different from the *simplex* method for linear programming.

In [16]:
```
1 result = minimize(dips, [2,2], method='Nelder-Mead', options={'dis
2 print( result.x, result.fun)
```

```
Optimization terminated successfully.
         Current function value: -45.000000
         Iterations: 60
         Function evaluations: 119
[ 3.00000000e+00 -1.83332383e-05] -45.00000000000001
```

In [ ]:
```
1
```

Constrained Optimization

Often we want to minimize/maximize $f(x)$ under constraints on x, e.g.

- inequality constraints $g_j(x) \leq 0, (j = 1, \ldots, m)$
- equality constraints $h_j(x) = 0, (j = 1, \ldots, r)$

Penalty function

Define a function with penalty terms:

$$P(x, \rho) = f(x) + \sum_j \beta_j(x) g_j(x)^2 + \sum_j \rho h_j(x)^2$$

$$\beta_j(x) = \begin{cases} 0 & \text{if} \quad g_j(x) \leq 0 \\ \rho & \text{if} \quad g_j(x) > 0 \end{cases}$$

and increase ρ to a large value.

Lagrange multiplyer method

For minimization of $f(x)$ with equality constraints $h_j(x) = 0, (j = 1, \ldots, r)$, define a *Lagrangian function*

$$L(x, \lambda) = f(x) + \sum_j \lambda_j h_j(x).$$

The necessary condition for a minimum is:

$$\frac{\partial L(x, \lambda)}{\partial x_i} = 0 \qquad (i = 1, \ldots, n)$$

$$\frac{\partial L(x, \lambda)}{\partial \lambda_j} = 0 \qquad (j = 1, \ldots, r)$$

Scipy implements `SLSQP` (Sequential Least SQuares Programming) method. Constraints are defined in a sequence of dictionaries.

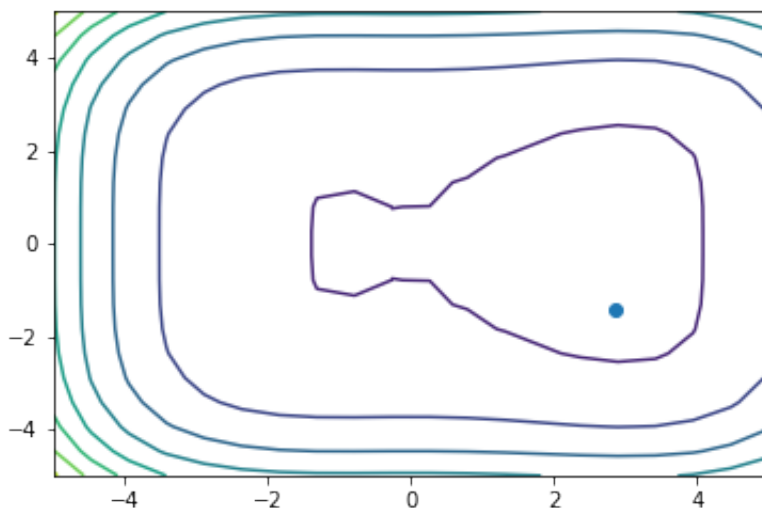
In [17]:
```
# h(x) = x[0] + 2*x[1] = 0
def h(x):
    return -(x[0] + 2*x[1])
def h_grad(x):
    return -(np.array([1, 2]))
cons = ({'type': 'ineq', 'fun': h, 'jac':h_grad})
```

In [18]:
```
result = minimize(dips, [-2,1], jac=dips_grad,
               method='SLSQP', constraints=cons, options={'disp': True
print( result.x, result.fun)
```

```
Optimization terminated successfully.    (Exit mode 0)
            Current function value: -40.37440406204255
            Iterations: 7
            Function evaluations: 12
            Gradient evaluations: 7
[ 2.86713506 -1.43356753] -40.37440406204255
```

In [19]:
```
plt.contour(x1, x2, fx)
plt.plot(result.x[0], result.x[1], 'o')
```

Out[19]: [<matplotlib.lines.Line2D at 0x151d55f128>]



Genetic/Evolutionaly Algorithms

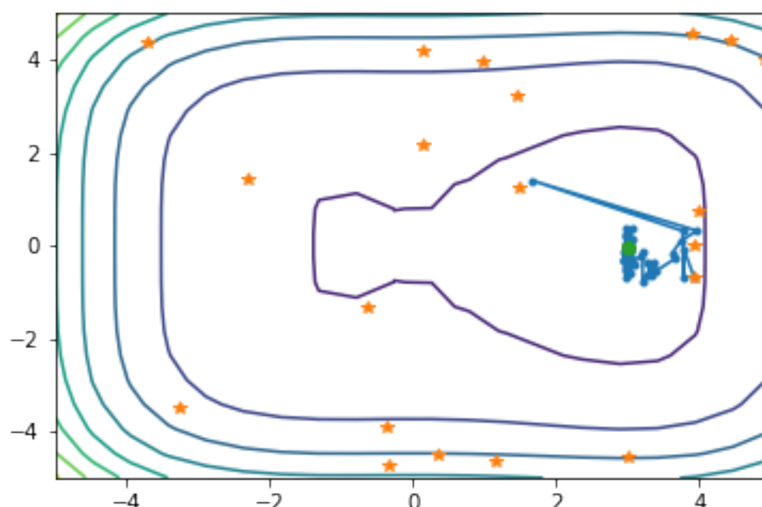
For objective functions with many local minima/maxima, stochastic search methods are preferred. They are called *genetic algorithm (GA)* or *evolutionay algorithm (EA)*, from an analogy with mutation and selection in genetic evolution.

In [20]:
```
def evol_min(f, x0, sigma=0.1, imax=100):
    """simple evolutionary algorithm
    f: function to be minimized
    x0: initial population (p*n)
    sigma: mutation size"""
    p, n = x0.shape  # population, dimension
    x1 = np.zeros((p, n))
    xh = np.zeros((imax, n))  # history
    for i in range(imax):
        f0 = f(x0.T)  # evaluate the current population
        fmin = min(f0)
        xmin = x0[np.argmin(f0)]
        #print(xmin, fmin)
        xh[i] = xmin  # record the best one
        # roulette selection
        fitness = max(f0) - f0  # how much better than the worst
        prob = fitness/sum(fitness)  # selection probability
        #print(prob)
        for j in range(p):  # pick a parent for j-th baby
            parent = np.searchsorted(np.cumsum(prob), np.random.ra
            x1[j] = x0[parent] + sigma*np.random.randn(n)
        x0 = x1
    return(xmin, fmin, xh)
```

In [21]:
```
x0 = np.random.rand(20, 2)*10 - 5
xmin, fmin, xhist = evol_min(dips, x0, 0.1)
print(xmin, fmin)
plt.contour(x1, x2, fx)
plt.plot(xhist[:,0], xhist[:,1], '.-')
plt.plot(x0[:,0], x0[:,1], '*')
plt.plot(xhist[-1,0], xhist[-1,1], 'o')
```

```
[ 2.81820241 -0.05245301] -44.999304660870536
```

Out[21]: [<matplotlib.lines.Line2D at 0x151d672ac8>]



For more advanced genetic/evolutionary algorithms, you can use `deap` package:
[https://github.com/DEAP (https://github.com/DEAP)](https://github.com/DEAP)

In [ ]:
```
1
```

Exercise

1. Try with you own function

1) Define a function of your interest (with two or more inputs) to be minimized or maximized. It can be an explicit mathematical form, or given implictly as a result of simulation.

In [ ]:
```
1 def fun(x):
2
3
```

2) Visualize the function, e.g., by surface plot or contour plot.

In [ ]:
```
1
```

3) Apply two or more optimization algorithms to the function and compare the results, with different starting points and parameters.

In [ ]:
```
1
```

Option) Set equality or inequality constraints and apply an algorithm for constrained optimization.

In [ ]:
```
1
```

In [ ]:
```
1
```