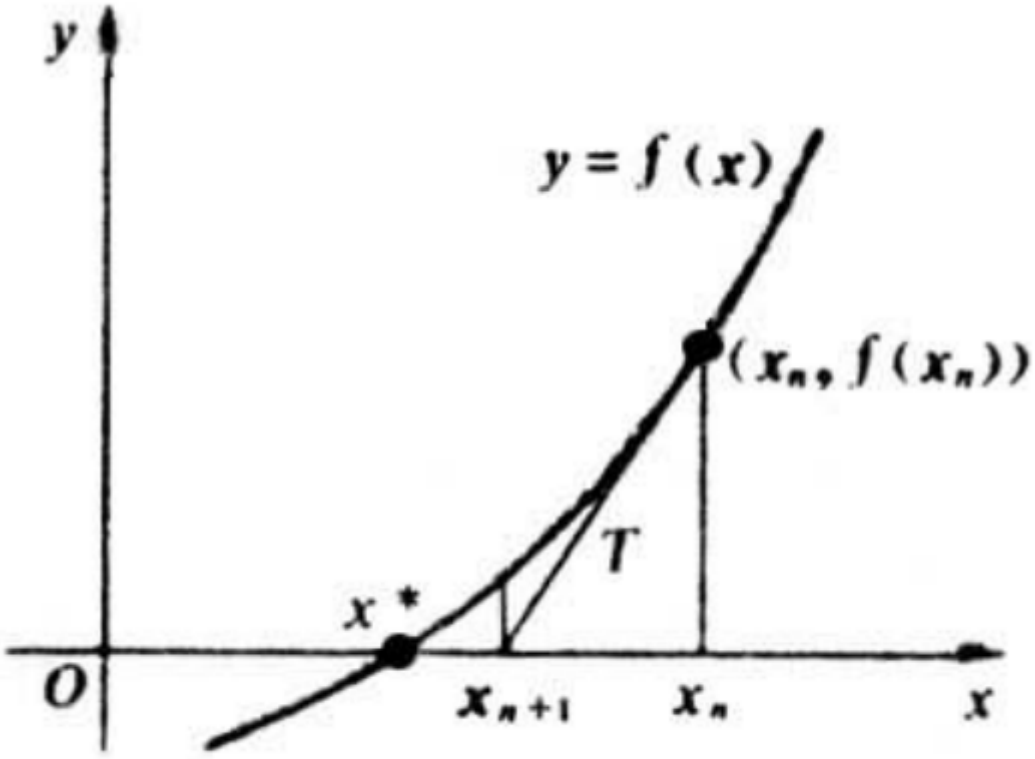
6.6 牛顿法

> 优点：相较于梯度下降法(一阶收敛)，收敛速度快
>
> 缺点：相较于拟牛顿法，计算海森矩阵的逆比较困难，消耗时间和计算资源。

一. 解方程

> 求解函数f(x)，即当f(x)=0时，x的值
>
> 用一阶泰勒公式展开得：f(x) = f(x0)+(x - x0)f'(x0)
>
> 综合上述得： f(x0)+(x - x0)f'(x0)=0
>
> 解得第一个解为：x = x1=x0 - f(x0)/f'(x0)



牛顿法求实根图示

https://images2017.cnblogs.com/blog/1022856/201709/1022856-20170916202719078-1588446775.gif

二.求最优解(二阶泰勒)

> 求目标函数的导函数等于零时的解

本思想是: 在现有极小点估计值的附近对 $f(x)$ 做二阶泰勒展开, 进而找到极小点的下一个估计值. 设 x_k 为当前的极小点估计值, 则

$$\varphi(x) = f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2}f''(x_k)(x - x_k)^2 \tag{1.2}$$

表示 $f(x)$ 在 x_k 附近的二阶泰勒展开式 (略去了关于 $x - x_k$ 的高阶项). 由于求的是最值, 由极值必要条件可知, $\varphi(x)$ 应满足

$$\varphi'(x) = 0, \tag{1.3}$$

即

$$f'(x_k) + f''(x_k)(x - x_k) = 0, \tag{1.4}$$

从而求得

$$x = x_k - \frac{f'(x_k)}{f''(x_k)}, \tag{1.5}$$

于是, 若给定初始值 x_0, 则可以构造如下的迭代格式

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}, \quad k = 0, 1, \cdots \tag{1.6}$$

三. 多维求解

$$\varphi(\mathbf{x}) = f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k) \cdot (\mathbf{x} - \mathbf{x}_k) + \frac{1}{2} \cdot (\mathbf{x} - \mathbf{x}_k)^T \cdot \nabla^2 f(\mathbf{x}_k) \cdot (\mathbf{x} - \mathbf{x}_k), \tag{1.7}$$

其中 ∇f 为 f 的**梯度向量**, $\nabla^2 f$ 为 f 的**海森矩阵** (Hessian matrix), 其定义分别为

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_N} \end{bmatrix}, \quad \nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_N} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_N} \\ & & \ddots & \\ \frac{\partial^2 f}{\partial x_N \partial x_1} & \frac{\partial^2 f}{\partial x_N \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_N^2} \end{bmatrix}_{N \times N}, \tag{1.8}$$

注意, ∇f 和 $\nabla^2 f$ 中的元素均为关于 $\mathbf{x}$ 的函数, 以下分别将其简记为 $\mathbf{g}$ 和 H. 特别地, 若 f 的混合偏导数可交换次序 (即对 $\forall\, i, j$, 成立 $\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}$), 则海森矩阵 H 为**对称矩阵**. 而 $\nabla f(\mathbf{x}_k)$ 和 $\nabla^2 f(\mathbf{x}_k)$ 则表示将 $\mathbf{x}$ 取为 $\mathbf{x}_k$ 后得到的实值向量和矩阵, 以下分别将其简记为 $\mathbf{g}_k$ 和 H_k (这里字母 g 表示 gradient, H 表示 Hessian).

同样地, 由于是求极小点, 极值必要条件要求它为 $\varphi(\mathbf{x})$ 的**驻点**, 即

$$\nabla\varphi(\mathbf{x}) = 0, \tag{1.9}$$

亦即 (通过在 (1.7) 两边作用一个梯度算子)

$$\mathbf{g}_k + H_k \cdot (\mathbf{x} - \mathbf{x}_k) = 0. \tag{1.10}$$

进一步, 若矩阵 H_k **非奇异**, 则可解得

$$\mathbf{x} = \mathbf{x}_k - H_k^{-1} \cdot \mathbf{g}_k, \tag{1.11}$$

于是, 若给定初始值 $\mathbf{x}_0$, 则同样可以构造出迭代格式

$$\mathbf{x}_{k+1} = \mathbf{x}_k - H_k^{-1} \cdot \mathbf{g}_k, \quad k = 0, 1, \cdots \tag{1.12}$$

这就是**原始的牛顿迭代法**, 其迭代格式中的搜索方向 $\mathbf{d}_k = -H_k^{-1} \cdot \mathbf{g}_k$ 称为**牛顿方向**. 下面给

算法 1.1 (牛顿法)

1. 给定初值 $\mathbf{x}_0$ 和精度阀值 ϵ, 并令 $k := 0$.
2. 计算 $\mathbf{g}_k$ 和 H_k.
3. 若 $\|\mathbf{g}_k\| < \epsilon$, 则停止迭代; 否则确定搜索方向 $\mathbf{d}_k = -H_k^{-1} \cdot \mathbf{g}_k$.
4. 计算新的迭代点 $\mathbf{x}_{k+1} := \mathbf{x}_k + \mathbf{d}_k$.
5. 令 $k := k + 1$, 转至步 2.

当目标函数是二次函数时, 由于二次泰勒展开函数与原目标函数不是近似而是完全相同的二次式, 海森矩阵退化成一个常数矩阵, 从任一初始点出发, 利用 (1.12) 只需一步迭代即可达到 $f(\mathbf{x})$ 的极小点 x^*, 因此牛顿法是一种具有**二次收敛性**的算法. 对于非二次函数, 若函数的二次性态较强, 或迭代点已进入极小点的邻域, 则其收敛速度也是很快的, 这是牛顿法的主要优点.

但原始牛顿法由于迭代公式中没有步长因子, 而是定步长迭代, 对于非二次型目标函数, 有时会使函数值上升, 即出现 $f(\mathbf{x}_{k+1}) > f(\mathbf{x}_k)$ 的情况, 这表明原始牛顿法**不能保证函数值稳定地下降**, 在严重的情况下甚至可能造成迭代点列 $\{\mathbf{x}_k\}$ 的发散而导致计算失败.

四. Python numpy

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

def dampnm(fun,gfun,hess,x0):
    # 用牛顿法求解无约束问题
    #x0是初始点，fun，gfun和hess分别是目标函数值，梯度，海森矩阵的函数
    maxk = 500
    rho = 0.55
    sigma = 0.4
    k = 0
    epsilon = 1e-5

    f=open("牛顿.txt","w")
    W = np.zeros((2, 20000))

    while k < maxk:
        W[:, k] = x0
        gk = gfun(x0)
        Gk = hess(x0)
        dk = -1.0*np.linalg.solve(Gk,gk)
        print(k, np.linalg.norm(dk))
        f.write(str(k)+'   '+str(np.linalg.norm(gk))+"\n")
        if np.linalg.norm(dk) < epsilon:
```

```
            break

        x0 += dk
        k += 1
    W = W[:, 0:k + 1]  # 记录迭代点
    return x0,fun(x0),k,W

# 函数表达式fun
fun = lambda x:100*(x[0]**2-x[1])**2 + (x[0]-1)**2

# 梯度向量 gfun
gfun = lambda x:np.array([400*x[0]*(x[0]**2-x[1])+2*(x[0]-1), -200*(x[0]**2-
x[1])])

# 海森矩阵 hess
hess = lambda x:np.array([[1200*x[0]**2-400*x[1]+2, -400*x[0]],[-400*x[0],200]])

if __name__=="__main__":
    X1 = np.arange(-1.5, 1.5 + 0.05, 0.05)
    X2 = np.arange(-3.5, 2 + 0.05, 0.05)
    [x1, x2] = np.meshgrid(X1, X2)
    f = 100 * (x2 - x1 ** 2) ** 2 + (1 - x1) ** 2  # 给定的函数
    plt.contour(x1, x2, f, 40)  # 画出函数的20条轮廓线

    x0 = np.array([-1.2, 1])
    out=dampnm(fun, gfun, hess, x0)
    print(out[2],out[0])

    W = out[3]
    print(W[:,:])

    plt.plot(W[0, :], W[1, :], 'g*-')
    plt.show()
```