```python
import numpy as np

import matplotlib.pyplot as plt

import scipy as sp
import math
import sympy as syp
```

对于一般的高阶函数或者超越方程,我们很难得到其解析解,在这种情况下,若$f(x) \in C_{[a,b]}$,且$f(a)f(b) < 0$,那么我们如何求解方程零点呢?\ 通常来讲,我们有如下的方法来解决这个问题:

1. 二分法
2. 不动点迭代
3. Newton法及其变形

在介绍这些方法之前我们需要有如下知识储备:

1. 零点存在性定理:\ if $f(x) \in C_{[a,b]}$ and when $f(a)f(b) < 0$ then there exists as least one root $x_0 \in (a,b)$ such that $f(x_0) = 0$.
2. 根分解定理:\ if exists x_0 such that $f(x_0) = 0$ then f can be decomposed into $f(x) = (x - x0)^m g(x)$ when $g(x0) \neq 0$.

我们首先来介绍 `二分法` 的基本原理:\ 二分法是解决一个方程的经典方法，其基本思想是每次将方程的解空间范围缩小一半，直到找到方程的解。

```python
##在计算过程中,我们将采用数值微分与符号微分同时进行的策略
```

```python
def f(x):
    return 100 * (3**x - 1) - 1200 * x
```

```python
def bisection_method(f,x_in,set,tol = 1e-6,max_iter = 150):
    """
    采用二分法计算零点
    (
        二分法有着本身固有的缺陷:
        1、无法向高维推广
        2、对于复杂函数可能很难找到全部零点
    )

    Args:
        f (_type_): _description_
        a (_type_): _description_
        b (_type_): _description_
        tol (_type_, optional): _description_. Defaults to 1e-6.
        max_iter (int, optional): _description_. Defaults to 150.
    """
    a,b = set

    x0 = x_in

    t = np.linspace(a,b,51)

    i = 0

    while i < max_iter:

        x1 = (a+b)/2

        if abs(f(x1))<tol:
            return x1

        if f(x1)*f(a) < 0:
            b = x1
        else:
            a = x1

        i += 1

    return x1
```

```python
set = [0,2]

x0 = bisection_method(f,0.5,set)


print(f(x0),x0)
```

```
-1600.0 2.0
```

我们再来介绍 `不动点迭代法` 的基本原理:\ 我们将$f(x) = 0$ 表达为等价形式$x = \varphi(x)$\ 若存在x^*满足上式,则称之为$\varphi(x)$的一个不动点.求解$f(x) = 0$即求解$\varphi(x) = x$的解

基于这一思想,我们可以构造迭代格式$x_{k+1} = \varphi(x_k)$,重复迭代,直至其近似收敛.

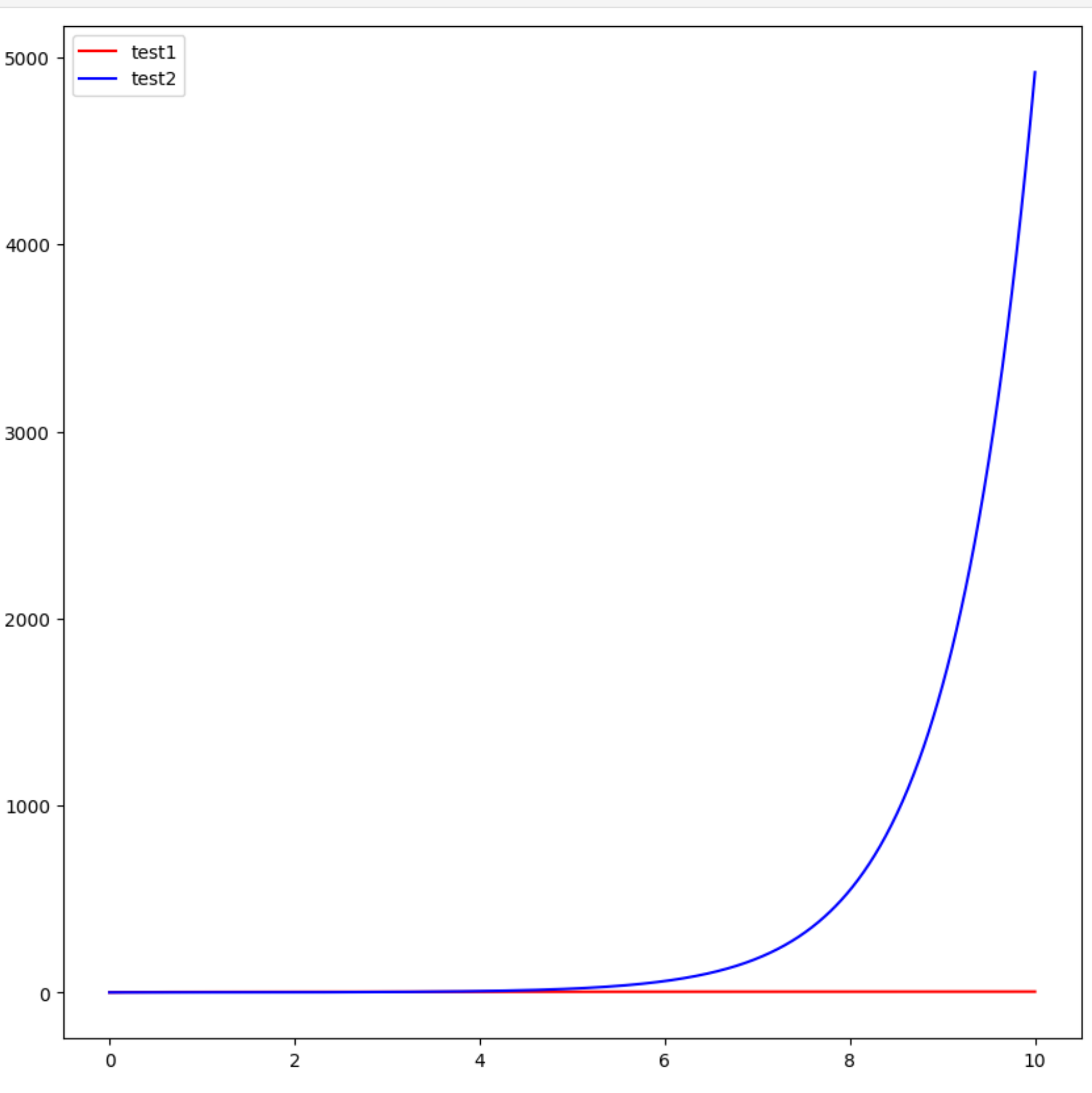
```python
##容易发现,不动点迭代具有非常多变的特点,但是他们的收敛效率差别巨大,我们可以通过以下这两个例子发现

def test1(x):
    return np.log(12 * x + 1) / np.log(3)

def test2(x):
    return (3**x - 1) / 12

x = np.linspace(0,10,1000)

plt.figure(figsize=(10,10))
plt.plot(x,test1(x),'r-',label='test1')
plt.plot(x,test2(x),'b-',label='test2')
plt.legend()
plt.show()
```



```python
iter_1,iter_2 = 0,0

max_iter = 150

tol = 1e-6

x0 = np.random.rand()

while True:
    x1 = test1(x0)
    if np.abs(x1-x0) < tol or iter_1 == max_iter:
        break
    x0 = x1
    iter_1 += 1

print(f"第一种迭代格式{test1.__name__}的迭代次数为{iter_1}次，迭代结果为{x1}")

x0 = np.random.rand()

while True:
    x1 = test2(x0)
    if np.abs(x1-x0) < tol or iter_2 == max_iter:
        break
    x0 = x1
    iter_2 += 1

print(f"第二种迭代格式{test2.__name__}的迭代次数为{iter_2}次，迭代结果为{x1}")
```

```
第一种迭代格式test1的迭代次数为13次，迭代结果为3.3970598739670415
第二种迭代格式test2的迭代次数为5次，迭代结果为1.6792183366032276e-08
```

下面我们来介绍牛顿法及其变形:\ 牛顿法(Newton Method)，也叫牛顿迭代法，是一种迭代方法，用于求解非线性方程组的根。\ 牛顿法通过迭代地计算函数的导数来逼近方程组的根。

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

由此,我们易知f(x) = 0,即:

$$x_{k+1} = x_k - f(x_k)/f'(x_k)。$$

因此,牛顿法通过迭代地计算函数的导数来逼近方程组的根。

```python
##Method1 采用数值微分的方式计算函数

def f(x):
    return 3 * x ** 2 - np.exp(x)

def NewtonIteration_Numeric(f,x_in,set, tol = 1e-6, max_iter = 150):
    """采用数值微分的形式计算牛顿迭代法

    Args:
        x0 (_double/float_): 初值
        set (_list/array_): 迭代区间
        tol (_type_, optional): 误差上限. Defaults to 1e-6.
        max_iter (int, optional): 最大迭代次数. Defaults to 150.
    """

    a,b = set
    x0 = x_in

    epsilon = 1e-6

    i = 0

    while True:

        if x0 < a or x0 > b:
            print('x0超出迭代区间')
            break

        df = (f(x0 + epsilon) - f(x0)) / epsilon

        x1 = x0 - f(x0) / df

        if abs(x1 - x0) < tol or i == max_iter:

            x = x1

            break

        x0 = x1

        i += 1

    if i == max_iter:
        print('迭代次数超过最大次数')

    return x
```

```python
set = [3,5]

x = NewtonIteration_Numeric(f,3.5,set)

print(f"{f.__name__}在区间{set}的零点为{x}")
```

```
f在区间[3, 5]的零点为3.7330790286328144
```

```python
##Method2 采用符号微分的方式计算函数零点

def NewtonIteration_Symbol(x_in,set,tol = 1e-6,max_iter = 150):
    """采用符号微分的方式计算函数零点

    Args:
        f (_Function_): 传入函数,一定要用sympy库定义
        x_in (_float_): 传入初始零点
        set (_list/array/tuple_): 零点存在区间
        tol (_type_, optional): 计算误差上限. Defaults to 1e-6.
        max_iter (int, optional): 最大迭代次数. Defaults to 150.

    """

    a,b = set

    x = syp.Symbol('x')

    ##f = eval(input("输入计算的函数:"))
    f = 3*x**2 + syp.exp(x)

    diff1 = syp.diff(f,x)

    x0 = x_in

    x_out = None

    i = 0

    while True:

        if x0 < a or x0 > b:
            print('x0超出迭代区间')
            break

        x1 = x0 - f.subs(x,x0) / diff1.subs(x,x0)

        if abs(x1 - x0) < tol or i == max_iter:

            x_out = x1

            break

        x0 = x1

        i += 1

    if i == max_iter:
        print('迭代次数超过最大次数')

    return x_out
```

```python
set = [-2,2]

x = NewtonIteration_Symbol(0.5,set)

print(f"{f.__name__}在区间{set}的零点为{x}")
```

```
x0超出迭代区间
f在区间[-2, 2]的零点为None
```

```python
def NewtonIteration_linear(f, x_in, set, tol = 1e-6, max_iter = 150):

    a,b = set
    x0,x1 = x_in

    i = 0
    x = None

    while i < max_iter:

        df = (f(x1) - f(x0)) / (x1 - x0)

        x2 = x1 - f(x1) / df

        if abs(x2-x1) < tol:

            x = x2

            break

        x1,x0 = x2,x1

        i += 1


    if i == max_iter:

        print(f'Iteration:{i+1},Maximum iteration reached')

    return x
```

```python
set = [3,5]
x_in = [3.5,3.6]
x = NewtonIteration_linear(f,x_in,set)

print(f"{f.__name__}在区间{set}的零点为{x}")
```

```
f在区间[3, 5]的零点为3.7330790285461832
```

在分析完一维的情况后,我们来考虑一下多维的情况:

1. 二分法:\ 这一方法在高维时将难以再做出推广.因为我们对于一个区间的二分总是容易分的,但是对于一个区域而言,我们很难知道应该如何划分.\
2. Newton法:\ 对于一个$X \in \R^n$,由Newton法的思想,我们仍然可以采用:

$$X_{k+1} = Xk - \frac{f(X_k)}{f'(X_k)}$$

此时$f'(X_k)$为f关于X各分量的偏导构成的矩阵,即 `Jacobii` 矩阵.

$$f'(X_k) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

,在实际求解时,我们非常不推荐使用矩阵求逆,而是更加推荐采用求解方程组的方法来解决这一问题,即:

$$a = \begin{cases} F'(X_k) * s = -F(X_k) \\ X_{k+1} = X_k + s \end{cases}$$

```python
## 采用数值微分的方式

def Newton_Iteration2D(f,x0,set = None,tol = 1e-6,max_iter = 150):
    """牛顿法求解非线性方程组的数值解

    Args:
        f (_type_): _description_
        x0 (_type_): _description_
        set (_type_, optional): _description_. Defaults to None.
        tol (_type_, optional): _description_. Defaults to 1e-6.
        max_iter (int, optional): _description_. Defaults to 150.

    Raises:
        ValueError: _description_
    """

    x0 = x0.copy()

    def diff(f,x,b,sigma = 1e-4):

        ## 采用 Krylov子空间方法 与 Arnoldi 方法求解线性方程组

        m = len(b)
        V = np.zeros((m,m+1))
        H = np.zeros((m+1,m))

        r0 = b
        bet = np.linalg.norm(r0)

        b0 = np.zeros(m)
        b0[0] = bet

        v1 = r0 / bet
        V[:,0] = v1

        for j in range(m):

            w = f(x + sigma*V[:,j]) - f(x)

            w = w / sigma

            for i in range(j+1):
                h = np.dot(w,V[:,i])
                w = w - h*V[:,i]
                H[i,j] = h

            H[j+1,j] = np.linalg.norm(w)
            V[:,j+1] = w / H[j+1,j]

        y = np.linalg.solve(H[:m,:],b0)
        x0 = np.dot(V[:,:m],y)

        return x0

    i = 0

    while i < max_iter:

        b = - f(x0)

        s = diff(f,x0,b)

        x0 = x0 + s

        if np.linalg.norm(s) < tol:
            break

        i += 1

    return [x0,i]
```

```python
def f(x):
    x1,x2 = x[0],x[1]

    f1 = x1 ** 2 - 10 * x1 + x2 ** 2 + 8
    f2 = x1 * x2**2 + x1 - 10 * x2 + 8

    return np.array([f1,f2]).astype(float)

X0 = [0,0]

X = Newton_Iteration2D(f,X0)

print(X)
```

```
[array([1., 1.]), 4]
```

上机练习:

ex1:\ 结合二分法和Newton法求解下列方程的实根:\ (1)$x^2 - 3x + 2 - e^x = 0$\ (2)$x^3 + 2x^2 + 10x - 20 = 0$

```python
def f_1(x):
    return x**2 - 3 * x + 2 - np.exp(x)

def f_2(x):
    return x**3 + 2 * x**2 + 10 * x - 20

set = [0,3]
x0 = 0.5

x_0_bise = bisection_method(f_1,x0,set)
x_0_newton = NewtonIteration_Numeric(f_1,x0,set)
print(f"第一个函数的二分法根为{x_0_bise},牛顿根为{x_0_newton}")

x_1_bise = bisection_method(f_2,x0,set)
x_1_newton = NewtonIteration_Numeric(f_2,x0,set)
print(f"第二个函数的根为{x_1_bise},牛顿根为{x_1_newton}")
```

```
第一个函数的二分法根为0.2575303316116333,牛顿根为0.2575302854398608
第二个函数的根为1.3688081502914429,牛顿根为1.3688081078213734
```

ex2:\ 分别用二分法,Newton法,割线法求解:

$$xe^x - 1 = 0$$

```python
def f(x):
    return x*np.exp(x) - 1

set = [0,1]

x_in_bise = 0.1
x_bise = bisection_method(f,x_in_bise,set)
print(f"采用二分法求解结果为:{x_bise}")

x_in_newton = 0.1
x_newton = NewtonIteration_Numeric(f,x_in_newton,set)
print(f"采用牛顿迭代法求解结果为:{x_newton}")

x_in_linear = [0.1,0.2]
x_linear = NewtonIteration_linear(f,x_in_linear,set)
print(f"采用线性插值法求解结果为:{x_linear}")
```

```
采用二分法求解结果为:0.567143440246582
采用牛顿迭代法求解结果为:0.5671432904097838
采用线性插值法求解结果为:0.5671432904097891
```

ex3:\ 利用Newton法或者Inexact Newton method计算:

$$\begin{cases} 3x_1 - \cos x_2x_3 - \frac{1}{2} = 0 & (3) \\ x_1^2 - 81(x_2 + 0.1)^2 + \sin x_3 + 1.06 = 0 & (4) \\ \exp -x_1x_2 + 20x_3 + \frac{10}{3}\pi - 1 = 0 & (5) \end{cases}$$

```python
def f(x):
    x1,x2,x3 = x[:3]

    f1 = 3*x1-np.cos(x2*x3) - 1/2
    f2 = x1 ** 2 - 81 * (x2 + 0.1) **2 + np.sin(x3) + 1.06
    f3 = np.exp(-x1 * x2) + 20 * x3 + 10 / 3 * np.pi - 1

    return np.array([f1,f2,f3]).astype(float)


X0 = np.random.random(3)

X = Newton_Iteration2D(f,X0)

X1 = sp.optimize.fsolve(f,X0)

print(f"经过Inexact Newton Iteration求解的根为:{X[0]}\n")
print(f"经过scipy fsolve求解的根为:{X1}")
```

```
经过Inexact Newton Iteration求解的根为:[ 5.00000000e-01  2.60183662e-11 -5.23598776e-01]

经过scipy fsolve求解的根为:[ 5.00000000e-01 -7.89773012e-12 -5.23598776e-01]
```

```python
## 尝试使用符号系统Sympy求解非线性方程组问题

x1,x2,x3 = syp.symbols('x1,x2,x3')

f1 = 3*x1-syp.cos(x2*x3) - 1/2

f2 = x1 ** 2 - 81 * (x2 + 0.1) **2 + syp.sin(x3) + 1.06

f3 = syp.exp(-x1 * x2) + 20 * x3 + 10 / 3 * syp.pi - 1

eq1 = syp.Eq(f1,0)
eq2 = syp.Eq(f2,0)
eq3 = syp.Eq(f3,0)

solution = syp.solve([eq1,eq2,eq3],[x1,x2,x3])

print(solution)
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_1320\2106127312.py in <module>
     13 eq3 = syp.Eq(f3,0)
     14 
---> 15 solution = syp.solve([eq1,eq2,eq3],[x1,x2,x3])
     16 
     17 print(solution)

e:\anaconda\lib\site-packages\sympy\solvers\solvers.py in solve(f, *symbols, **flags)
   1106         solution = _solve(f[0], *symbols, **flags)
   1107     else:
-> 1108         solution = _solve_system(f, symbols, **flags)
   1109 
   1110     #

e:\anaconda\lib\site-packages\sympy\solvers\solvers.py in _solve_system(exprs, symbols, **flags)
   1970                     got_s.add(s)
   1971                 if not hit:
-> 1972                     raise NotImplementedError('could not solve %s' % eq2)
   1973             else:
   1974                 result = newresult

NotImplementedError: could not solve (60*x3 - 3 + 10*pi)*exp(x2*(cos(x2*x3)/3 + 1/6)) + 3
```

可以发现,sympy库无法解出解,因而,这个方程组 `没有解析解` .\ 但是我们可以使用sympy库中提供的nsolve(func,variables,guess)来求出其相对来说更加精准的解.

```python
nsolution = syp.nsolve([f1,f2,f3],[x1,x2,x3],X0)

print(nsolution)
```

```
Matrix([[0.500000000000000], [-3.50929737102836e-18], [-0.523598775598299]])
```