ニュートン法

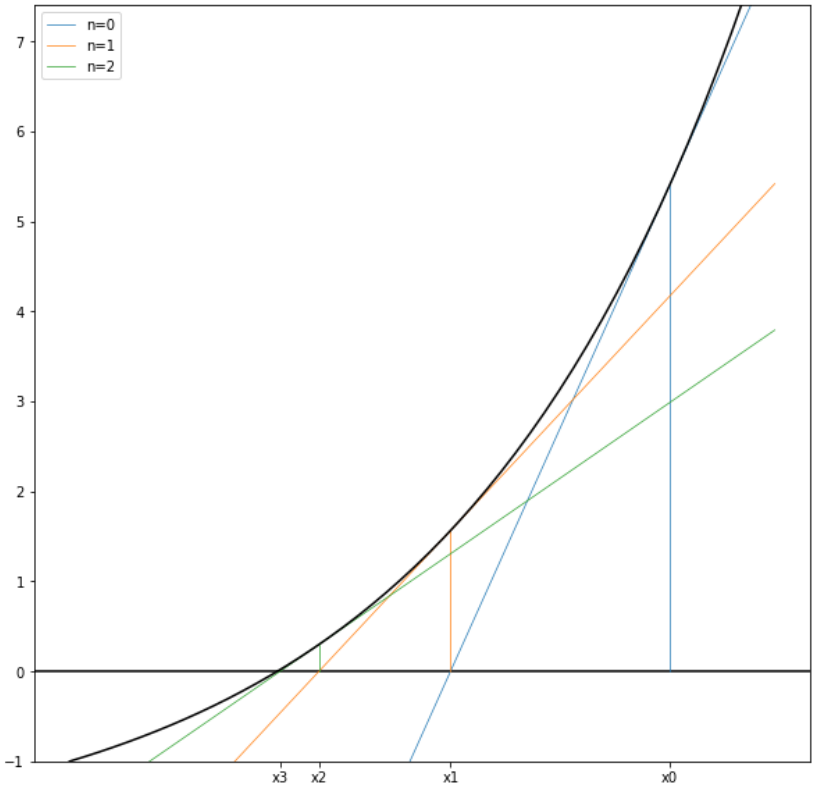
1 ニュートン法導入

今回は方程式の根を見つける方法である、ニュートン法について学習する。

ここで言う「根」とは、ある関数$f(x)$について、$f(x)=0$となるようなxのことである。

別の言い方をすると、関数fに関する$f(x)=0$の解のことである。

任意の関数の根を見つける解析解は式変形を要するためプログラムで行うのは難しい作業である。しかし、一次方程式の根は式が定まっているので簡単に求まる。そこで、適当な初期値x_0における$f(x)$の接線の根x_1を求め、さらにx_1における$f(x)$の根を求め…という反復操作を行う。こうして得られた数列$x_0,\ x_1,\cdots,x_n$が収束すれば、その値は$f(x)$の根であることが期待される。



では、このアルゴリズムを定式化してみよう。ある関数$f(x)$について、その微分を$f'(x)$とする。この時、$x = x_n$における$f(x)$の接線の式は

$$y - f(x_n) = f'(x_n)(x - x_n)$$

で表される。この接線の根は$y = 0$の時であり、接線の根をx_{n+1}とすると、

$$0 - f(x_n) = f'(x_n)(x_{n+1} - x_n)$$

となる。これを変形すると、

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

となる。こうして生成した数列を解くことで関数$f(x)$の根を求めるのがニュートン法である。ちなみに、接線の代わりに接二次関数を用いたものは Bailey 法と呼ばれているが、あまり使わないので名前を紹介するに留める。

今、方程式の根を見つける方法で少しご利益がある場合の例として、numpy の sqrt 関数を用いずに$\sqrt{2}$を計算してみよう。

$$f(x) = x^2 - 2$$

$$f'(x) = 2x$$

$$\frac{f(x)}{f'(x)} = \frac{x}{2} - \frac{1}{x}$$

これを用いれば、ニュートン法を用いて、初期値を 2 として$\sqrt{2}$を計算するプログラムは以下のように実装できる。

```
def sqrt2(max_n):
    xn = 2
    for n in range(max_n):
        xn = xn - xn / 2 + 1 / xn
    return xn
```

実際にこの関数を用い、np.sqrt の結果と比較してみて欲しい。

2 演習

任意の実数xに対してニュートン法を用いて平方根$\sqrt{x}$を計算する関数my_sqrtを実装せよ。ただし、ここで、my_sqrtは引数としてxと、ニュートン法の最大ステップ数max_nを受け取る。ただし、ここで、

$$f(x_n) = x_n^2 - x$$

に対して

$$\frac{f(x_n)}{f'(x_n)} = \frac{x_n}{x} - \frac{1}{x_n}$$

であることを利用して良い。また、実装した関数に引数として負の値を入れてみて、どうなるか試してみて欲しい。

3 一般的なニュートン法を実行する関数

先ほどまで、$\sqrt{\ }$専用のニュートン法を実装してきたが、もっと一般的に使える関数を作りたい。これは、fとf'も引数として受け取ることで実現する。

```
def simple_newton(fun, jac, x0, maxiter):
    xn = x0
    for n in range(maxiter):
        xn -= fun(xn) / jac(xn)
    return xn
```

今、$f(x) = x^3 - 10x^2 + x + 150$を例に挙げてニュートン法を実験してみよう。

```
def fun(x):
    return x**3 - 10*x**2 + x + 150


def jac(x):
    return 3 * x**2 - 20*x + 1
```

一応プロットしておこう。

```
x = np.linspace(-4, 10, endpoint=False)
plt.axhline(0, color='k', linewidth=0.7)
```

```
plt.plot(x, f(x))
```

グラフの様子からして、根は-3 付近にありそうだ。

```
simple_newton(fun, jac, -3, 10)
```

ところで、今、maxstep を適当に 10 にしたが、本当に 10 で良いだろうか？他の数値も試してみて欲しい。

実用的には maxstep がどれくらいあれば十分なのか分からないことがほとんどである。このような場合には絶対に収束するであろう十分大きな maxstep を取ればよい。しかし、逆に maxstep が大きすぎると計算に時間がかかりすぎてしまう。そこで、値が十分収束すればループを抜けるようにしよう。

```
def newton(fun, jac, x0, maxiter, tol):
    xn = x0
    xn_old = xn
    for n in range(maxiter):
        xn -= fun(xn) / jac(xn)
        if abs(xn - xn_old) < tol:
            break
        xn_old = xn
    return xn
```

使ってみよう。

```
newton(fun, jac, -3, 1000000, 1E-8)
```

4 ニュートン法の欠点

この章はではニュートン法の欠点を紹介しておく。欠点を可視化するために、以下の print する newton 法の関数を使う。

```
def newton_with_print(fun, jac, x0, maxiter, tol):
    xn = x0
    xn_old = xn
    for n in range(maxiter):
        xn -= fun(xn) / jac(xn)
```

```
        print("{}:¥t{}".format(n, xn))
        if abs(xn - xn_old) < tol:
            break
        xn_old = xn
    return xn
```

newton_with_print を使ってみよう。

```
newton_with_print(fun, jac, 3, 1000, 1E-8)
```

さて、ニュートン法の欠点が露わになるのは以下のような初期値を使った時だ。

```
newton_with_print(fun, jac, 1.8280502888150825, 1000, 1E-8)
```

なんと、この値に対してニュートン法は 126 回ものイテレーションをすることになる。結果の最初の方を見て欲しい。同じような値を反復していないだろうか？

このような現象を振動と呼ぶ。今回の関数は計算時間が短いので早く終わったが、もしも fun と jac が計算に 1 分かかるような関数だったとしたら、このニュートン法は 2 時間かかることになる。そこで、この欠点を補う減速ニュートン法という方法が存在する。ただし、ややマニアックなのでこの講義では名前を紹介するに留める。

5 ニュートン法を用いた関数の極値の求め方

関数の根を求めることももちろん実用的に興味のあることだが、関数の最小値や最大値等の極値もまた実用的に興味を持たれがちな量である。例えば、分子の安定構造を計算したければ、分子の座標を引数にエネルギーを返す関数を最小化する。このような計算は研究室では良く行われることである。

さて、このような最小化問題にもニュートン法を流用することが出来る。ある関数$f(x)$に対して、$\underset{x}{\mathrm{argmin}}\, f(x)$や$\underset{x}{\mathrm{argmax}}\, f(x)$を求めると言い換えても良い。この時、$f(x)$の極値において、$f'(x) = 0$であることを利用して、$f'(x)$に対してニュートン法を使うことで容易に極値を求めることが出来る。この時の更新式は以下である。

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

では、これを使って、$f(x) = x^3 - 10x^2 + x + 150$の 7 付近にある極小値を求めてみよう。

```
def fun(x):
    return x**3 - 10*x**2 + x + 150

def jac(x):
    return 3 * x**2 - 20*x + 1

def hes(x):
    return 6 * x - 20
```

```
newton_with_print(jac, hes, 7, 100, 1E-8)
```

6　おまけ・多変数関数の極値

最後に多変数関数を最小化するニュートン法を紹介しよう。

一変数の関数を最小化するニュートン法は関数のテーラー展開から導くことが出来る。ある意味、fの接二次関数の極値を求め続ける問題であるとも言える。fの二次のテーラー展開をf_2とすると、

$$f_2(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{1}{2}f''(x_n)(x - x_n)^2$$

となり、この近似された関数の極値を求めたい。この関数の微分は

$$f_2'(x) = f'(x_n) + f''(x_n)(x - x_n)$$

であり、接二次関数の極値をx_{n+1}とすると、

$$0 = f'(x_n) + f''(x_n)(x_{n+1} - x_n)$$

が得られ、式変形すると、

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

となる。

多変数の場合、テーラー展開は

$$f_2(x,y) = f(x_n,y_n) + f_x(x_n,y_n)(x-x_n) + f_y(x_n,y_n)(y-y_n) + \frac{1}{2}f_{xx}(x_n,y_n)(x-x_n)^2$$
$$+ \frac{1}{2}f_{xy}(x_n,y_n)(x-x_n)(y-y_n) + \frac{1}{2}f_{yx}(x_n,y_n)(y-y_n)(x-x_n)$$
$$+ \frac{1}{2}f_{yy}(x_n,y_n)(y-y_n)^2$$

であるが、式が長いので$\mathbf{r} \equiv (x,y),\ \nabla f(\mathbf{r}) = \left(f_x(\mathbf{r}),\ f_y(\mathbf{r})\right),\ \nabla\nabla f(\mathbf{r}) = \begin{pmatrix} f_{xx}(\mathbf{r}) & f_{xy}(\mathbf{r}) \\ f_{yx}(\mathbf{r}) & f_{yy}(\mathbf{r}) \end{pmatrix}$

とすると、式を簡潔にまとめることが出来る。

$$f_2(\mathbf{r}) = f(\mathbf{r}_n) + \nabla f(\mathbf{r}_n)\cdot(\mathbf{r}-\mathbf{r}_n) + \frac{1}{2}(\mathbf{r}-\mathbf{r}_n)^T\cdot\nabla\nabla f(\mathbf{r}_n)\cdot(\mathbf{r}-\mathbf{r}_n)$$

さて、$f_2(\mathbf{r})$の極値は x で微分しても y で微分しても 0 になるような点であるので、$\nabla f_2(\mathbf{r}) = \equiv (0,0)$を満たすような点である。従って、両辺のヤコビアンを取ると、

$$\nabla f_2(\mathbf{r}) = \nabla f(\mathbf{r}_n) + \nabla\nabla f(\mathbf{r}_n)\cdot(\mathbf{r}-\mathbf{r}_n)$$

となり、$\nabla f(\mathbf{r}_{n+1}) = 0$とすると、

$$0 = \nabla f(\mathbf{r}_n) + \nabla\nabla f(\mathbf{r}_n)\cdot(\mathbf{r}_{n+1}-\mathbf{r}_n)$$

が得られる。変形すると、

$$\mathbf{r}_{n+1} = \mathbf{r}_n - \left(\nabla\nabla f(\mathbf{r}_n)\right)^{-1}\nabla f(\mathbf{r}_n)$$

となる。ここで、$\left(\nabla\nabla f(\mathbf{r}_n)\right)^{-1}$は$\nabla\nabla f(\mathbf{r}_n)$の逆行列である。

なお、$\nabla f(\mathbf{r})$はヤコビアンと呼ばれ、$\nabla\nabla f(\mathbf{r})$はヘシアンと呼ばれる。

では実装してみよう。今、関数の例として、$f(x,y) = x^2 + y^2$を使おう。この関数は$(x,y) = 0$で極小値 0 を持つはずである。また、この関数に対して、

$$\nabla f(\mathbf{r}) = 2\mathbf{r} = (2x, 2y)$$

$$\nabla\nabla f(\mathbf{r}) = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

である。

```
def fun(r):
    x, y = r
    return x * x + y * y
def jac(r):
    return 2 * r

def hes(r):
    return np.array([[2.0, 0.0],
                     [0.0, 2.0]])
```

```
def newton_simple_multi(jac, hes, r0, maxiter):
    rn = np.copy(r0)
    for n in range(maxiter):
        rn -= np.linalg.solve(hes(rn), jac(rn))
    return rn
```

```
newton_simple_multi(jac, hes, np.array([3.0, 3.0]), 100, 1E-8)
```

実は、多次元の二次関数に対してはニュートン法は一回のステップで収束する。

6.1 演習

何か他の、三次以上の二変数関数を一つ自分で決め、に対してもニュートン法を使ってみよう。

7 さらなる学習に向けて

多変数関数に対してニュートン法を用いるにはヘシアンを計算する必要があることを見た。しかし、一般にヘシアンは量も多く、微分の次数も多いので計算が大変なことも多く、ヘシアン

を近似する準ニュートン法と呼ばれる手法群が実用的である。中でも、BFGS 法という方法が有用なので、名前だけでも知っておくと良いだろう。