数値微分と数値積分

1 関数の準備

今回はガウス関数

$$y = \exp(-x^2)$$

$$y' = -2x\exp(-x^2)$$

$$y'' = -2\exp(-x^2) + 4x^2\exp(-x^2)$$

を例に、数値微分や数値積分を実験してみよう。

まずはライブラリをインポートしよう。

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

次いでx軸とy, y', y''、軸の間隔hを作る。

```
start = -2
stop = 2
num = 50
x = np.linspace(start, stop, num, endpoint=True)
y = np.exp(-x * x)
dxdy_exact = -2 * x * np.exp(-x * x)
dx2d2y_exact = -2 * np.exp(-x * x) + 4 * x * x * np.exp(-x * x)
h = (stop - start) / (num - 1)
```

なお、ここで、h = (stop − start)/(num)ではなく、h = (stop − start)/(num − 1)となっているのはx軸の作成で endpoint=True にして最後の点を含めたことに起因する。（植木算）

h の定義がこれで良いことは一応確認しておこう。

```
x[1] - x[0]
```

```
h
```

プロットしておこう。

```
plt.plot(x, y, label="y")
plt.plot(x, dxdy_exact, label="y'")
plt.plot(x, dx2d2y_exact, label="y''")
plt.legend()
```

2 前進差分と後退差分

微分の定義は覚えているだろうか？微分可能な関数であれば正則な関数に関しては以下の二つの微分の定義は全く同じ値になるはずである。（ここで下付きの「f」や「b」は「forward」、「backward」を示し、前進と後退を意味する。）

$$\begin{cases} f_f'(x) = \lim_{h \to +0} \dfrac{f(x+h) - f(x)}{h} \\ f_b'(x) = \lim_{h \to +0} \dfrac{f(x) - f(x-h)}{h} \end{cases}$$

これらの定義をそのまま離散化した世界に持ち込むと、lim は使えず、有限の差分でもって微分を近似することになる。これを有限差分と言う。特に、以下の二つを前進差分、後退差分と言う。

$$\begin{cases} f_f'(x) = \dfrac{f(x+h) - f(x)}{h} \\ f_b'(x) = \dfrac{f(x) - f(x-h)}{h} \end{cases}$$

この時、h の値は小さい程微分の定義に近づくが、あまりに小さすぎると数値計算の誤差が出るので、適当な値を選ばなければならない。しかし、今回扱うような簡単な離散グリッドの場合は h は x 軸上の点と点の距離であると設定してしまえばよい。この時、i 番目の点での差分は以下で定義される。（ここで、(i)とせずに[i]と書くのは Python の配列アクセスをイメージしてのことである。）

$$\begin{cases} f_f'[i] = \dfrac{f[i+1] - f[i]}{h} \\ f_b'[i] = \dfrac{f[i] - f[i-1]}{h} \end{cases}$$

ただし、ここで、h はグリッド間の間隔であり、$\mathrm{x}[i+1] - \mathrm{x}[i] \approx h$となるはずである。

また、iの定義上、f_f'は最後のiに対しては計算出来ないし、f_b'は最初のiに対しては計算出来ないことに注意。従って、前進差分や後退差分は通常の関数より削った x 軸を使う

```
x_forward = np.array([x[i] for i in range(num-1)])
dxdy_forward = np.array([(y[i+1] - y[i]) / h for i in range(num-1)])
x_backward = np.array([x[i] for i in range(1, num)])
dxdy_backward = np.array([(y[i] - y[i-1]) / h for i in range(1, num)])
```

では、プロットしてみよう。

```
plt.plot(x, dxdy_exact, label="exact")
plt.plot(x_forward, dxdy_forward, label="forward")
plt.plot(x_backward, dxdy_backward, label="backward")
plt.legend()
```

3 中心差分

さて、前進差分と後退差分は微分の様子を大まかには表現出来ていたが、微妙にズレていたのではないだろうか？

num を大きくすることでズレは小さくなっていくが、それでも常にずれることになる。

そのため、もっと良い微分の方法を使うことが多い。中心差分である。先のプロットでは前進差分と後退差分はそれぞれ逆の方向にズレていたのではないだろうか？

そのことを考慮に入れると、前進差分と後退差分の平均がまあまあ良い微分の近似になりそうではなかろうか？これを中心差分と呼ぶ。以下、中心差分をf_c'で表す。

$$\begin{cases} f_f'(x) = \dfrac{f(x+h) - f(x)}{h} \\ f_b'(x) = \dfrac{f(x) - f(x-h)}{h} \end{cases}$$

$$f_c'(x) = \frac{f_f'(x) + f_b'(x)}{2} = \frac{f(x+h) - f(x-h)}{2h}$$

さて、これを今回の離散グリッドで表現すると、

$$f_c'[i] = \frac{f[i+1] - f[i-1]}{2h}$$

となる。では実際に中心差分を計算してみよう。中心差分ではx軸の最初も最後も計算出来ないことに注意しよう。

```
dxdy_central = np.array([(y[i+1] - y[i-1]) / (2 * h) for i in range(1, num - 1)])
x_central = np.array([x[i] for i in range(1, num - 1)])
```

プロットする。

```
plt.plot(x, dxdy_exact, label="exact")
plt.plot(x_central, dxdy_central, label="central")
plt.legend()
```

どうだろう？綺麗に一致したのではないだろうか？

4 二回微分

さて、中心差分の式は

$$f_c'(x) = \frac{f(x+h) - f(x-h)}{2h}$$

であった。この式を用いて二回の差分を計算することを考えよう。

$$f_c''(x) = \frac{f_c'(x+h) - f_c'(x-h)}{2h} = \frac{1}{2h}\left(\frac{f(x+2h) - f(x)}{2h} - \frac{f(x) - f(x-2h)}{2h}\right)$$

$$= \frac{f(x+2h) - 2f(x) + f(x-2h)}{4h^2}$$

となる。hは小さければ小さい程良いので、$2h = h'$とすると、

$$f_c''(x) = \frac{f(x+h') - 2f(x) + f(x-h')}{h'^2}$$

となる。これを等間隔グリッドを使って表すと、

$$f_c''[i] = \frac{f[i+1] - 2f[i] + f[i-1]}{h^2}$$

が得られる。

```
dx2d2y_central = np.array([(y[i+1] - 2 * y[i] + y[i-1]) / (h * h) for i in
range(1, num - 1)])
plt.plot(x, dx2d2y_exact, label="exact")
plt.plot(x_central, dx2d2y_central, label="central")
plt.legend()
```

5 数値積分

積分を数値的に行うにはどうすれば良いだろうか？このためには、一度区分求積に立ち戻ろう。区間[a, b]で関数$f(x)$を積分するとして、グリッド間隔をhとすると、

$$x_k = a + kh$$

となり、区分求積の式は

$$\int_a^b f(x)\mathrm{d}x = \lim_{n\to\infty} \sum_{k=0}^{n-1} f(x_k)h$$

である。この式から極限を外したものは数値積分できる

$$S \approx \sum_{k=0}^{n-1} f(x_k)h$$

今回は

$$y = \exp(-x^2) + 10$$

を題材として用いて数値積分を実行してみよう。

$$\mathrm{exact} = \int_{-a}^{a} [\exp(-x^2) + 10]\mathrm{d}x = \sqrt{\pi}\,\mathrm{erf}(a) + 20a$$

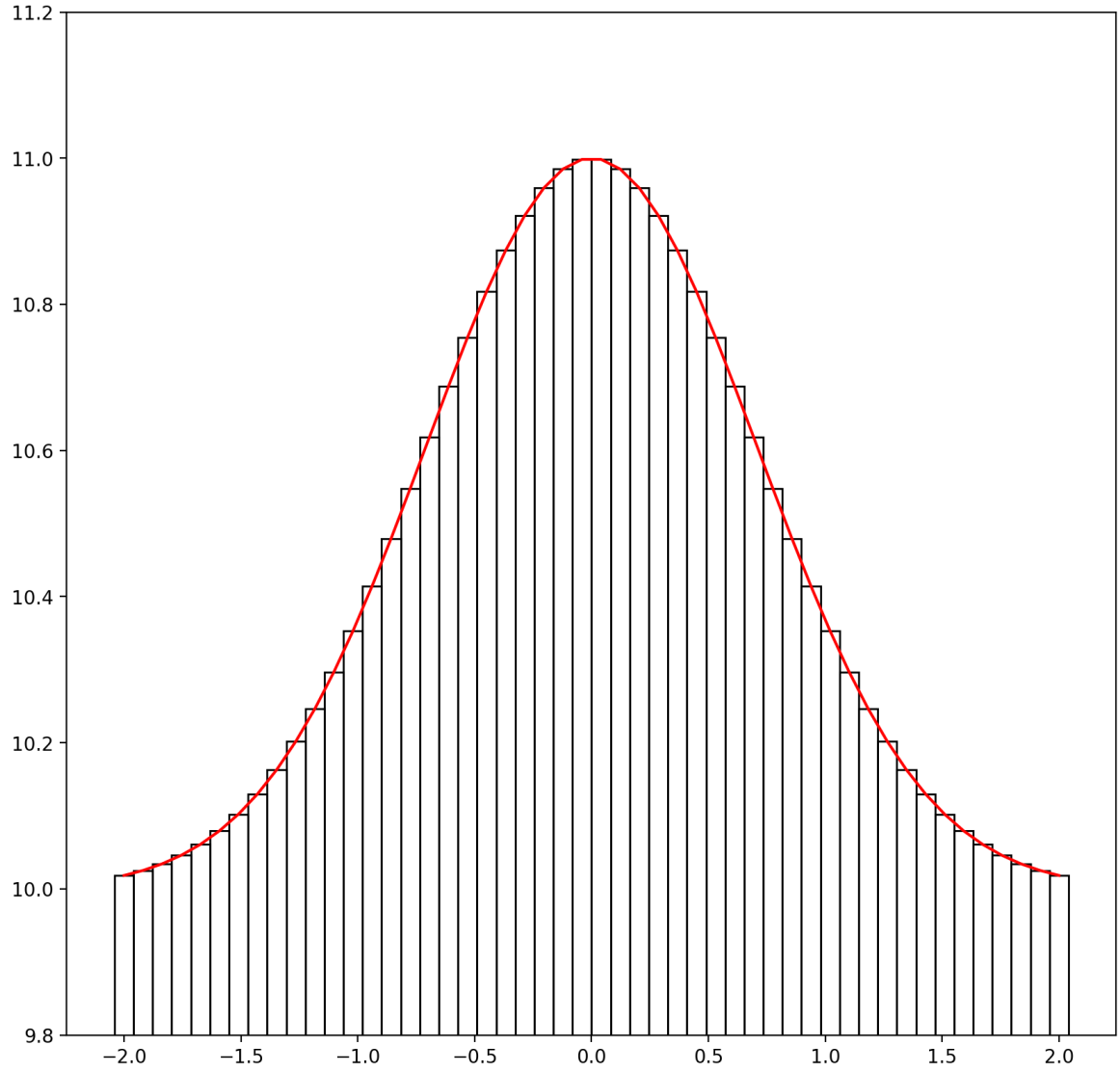
を考慮に入れつつ、積分の厳密値と関数を用意しよう。

```
a = 2
num = 50
x = np.linspace(-a, a, num, endpoint=True)
y = np.exp(-x * x) + 10
h = (stop - start) / (num - 1)
exact = np.sqrt(np.pi) * np.erf(2) + 20 * a
```

一応、y をプロットしておこう。

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

6　長方形近似



区分求積のイメージは離散化された各グリッドに高さ$f(x_k)$、幅hの長方形を並べたようなものとなる。

区分求積の式に従って面積を計算してみよう。

$$S \approx \sum_{k=0}^{n-1} f(x_k)h = h\sum_{k=0}^{n-1} f(x_k)$$

実装は非常に簡単な式となる。

```
s = np.sum(y) * h
```

厳密な積分値と比較してみよう。

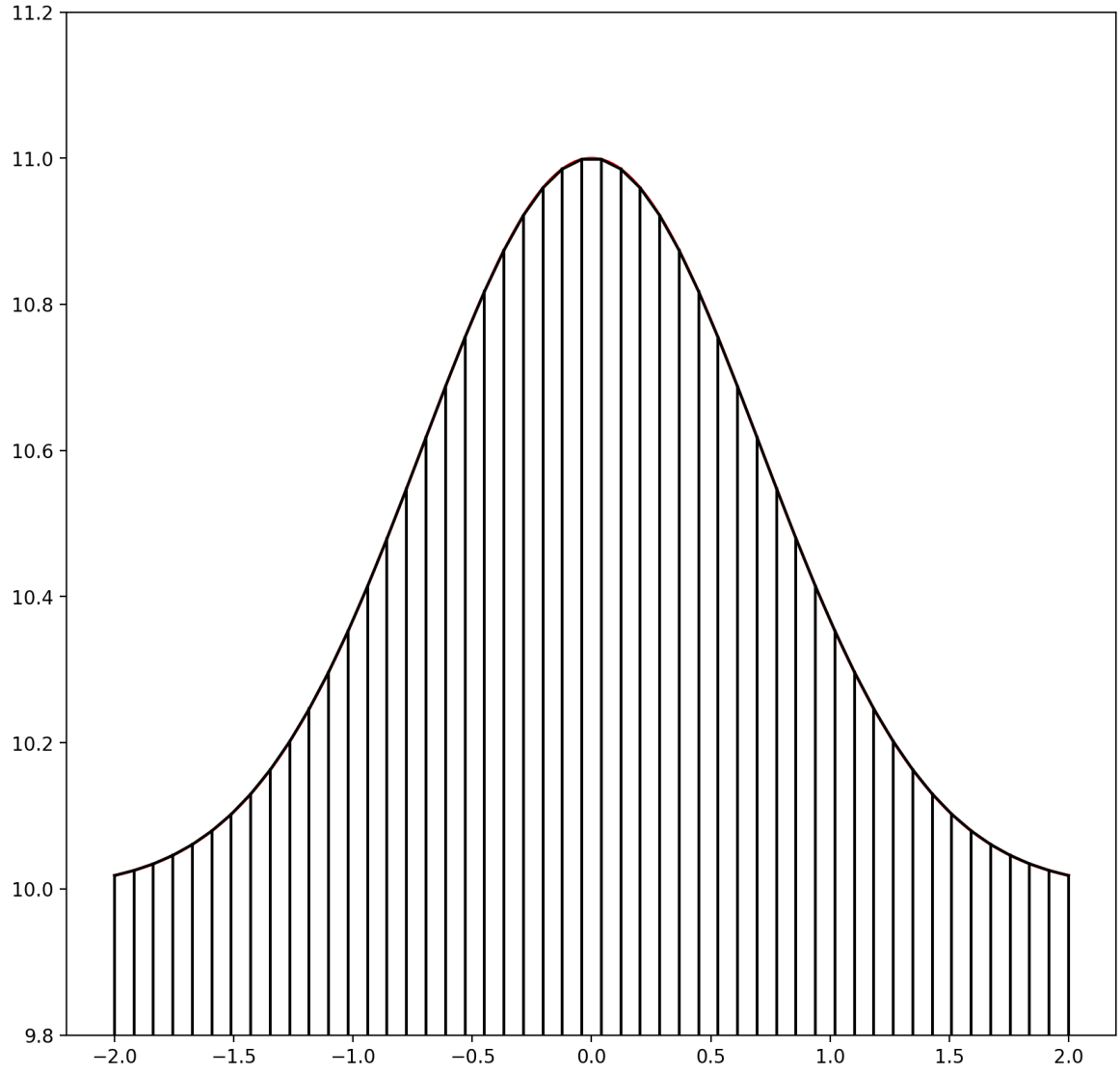
```
(s - exact) / (exact)
```

厳密値と 2%程のずれとなっていることがわかるだろう。

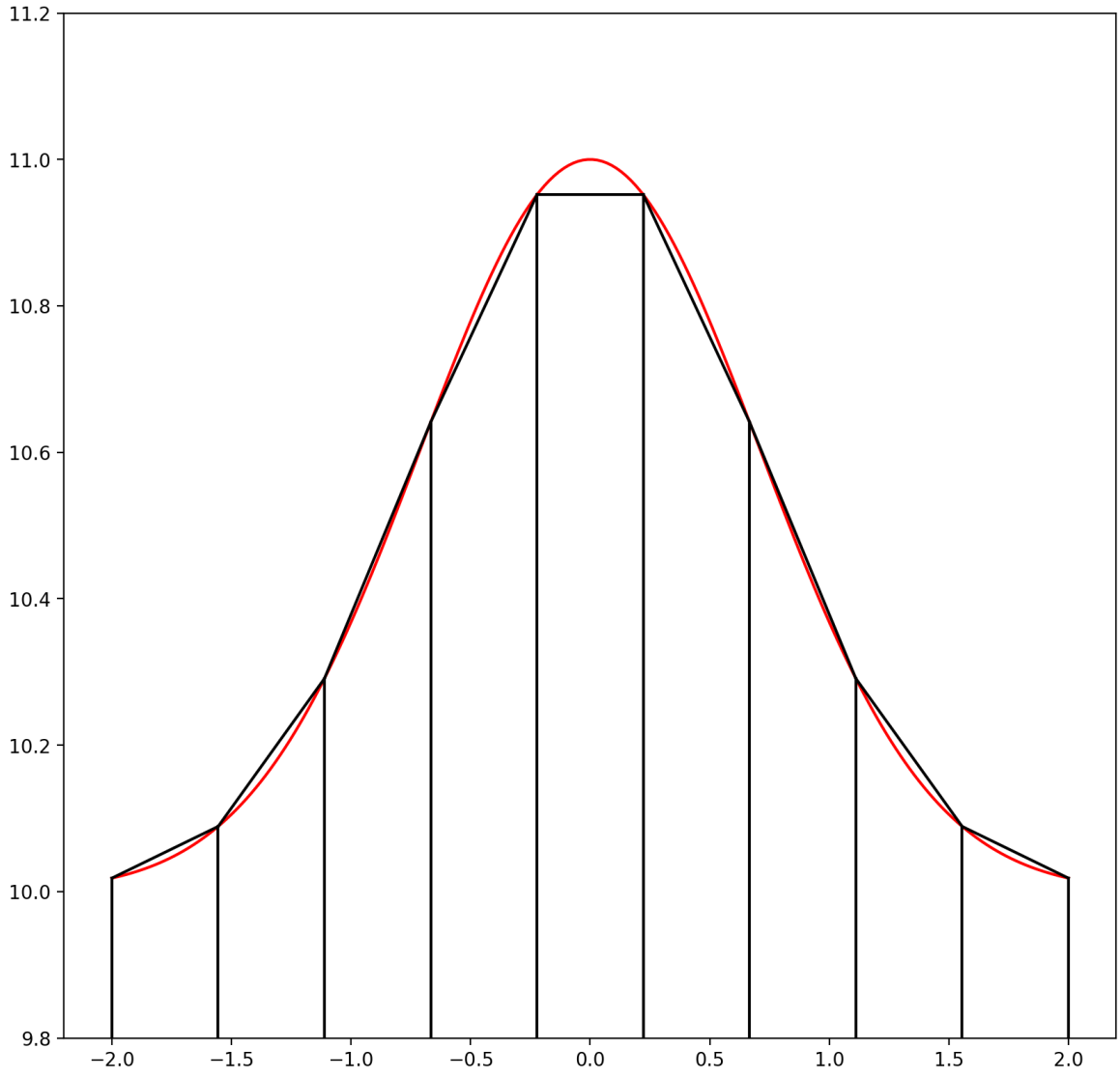
7 台形近似

長方形近似では 2%のずれが生じた。これも精度は悪くないが、もう少し良さそうな方法を考えよう。

各点グリッドの各点を用いてたくさんの台形を作り、たくさんの台形で関数を近似すると、長方形よりはよい近似で元の関数を表現出来そうだ。


11.2
11.0
10.8
10.6
10.4
10.2
10.0
9.8
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

これが実際にこの関数をたくさんの台形で近似した様子である。ただし、台形公式で 50 点取ると形が見えにくいので、10 点のバージョンも以下に示す。



少し見えづらいが、$\exp(-x^2)$を赤線で、台形による近似を黒線で描いている。一つ一つの台形の面積は$\frac{1}{2}h\big(f(x_k)+f(x_{k+1})\big)$で与えられるので、台形で近似された関数の面積は

$$S \approx \sum_{k=0}^{n-2} \frac{1}{2}h\big(f(x_k)+f(x_{k+1})\big)$$

で表される。$f(x_k)$は両端の$k=0,\ n-1$を除いて二回ずつ登場するので、結局この式は

$$S \approx \sum_{k=0}^{n-2} \frac{h}{2}\left(f(x_k) + f(x_{k+1})\right) = \frac{h}{2}\left(f(x_0) + f(x_{n-1})\right) + \sum_{k=1}^{n-2} f(x_k)h$$

となり、長方形近似の両端の寄与を半分になったバージョンと言うことが出来る。

実際にこの式を用いて計算してみよう。

```
s_trap = np.sum(y[1:-1]) * h + (y[0] + y[-1]) * 0.5 * h
```

厳密解との比も確認してみよう。

```
(s_trap - exact) / (exact)
```

このように、台形近似を用いた方法の方が精度が良くなることも良くある。

とはいえ、長方形近似と台形近似の違いは両端の寄与だけであり、分割数が多くなればなるほど両端の寄与は少なくなり、その差はなくなっていく。

他にも台形ではなく二次関数によって近似する「合成シンプソン公式」や、等間隔グリッドの代わりにルジャンドル多項式の零点をグリッドとして用いることで、n 点で 2n+1 次多項式まで厳密に積分できる「Gauss 求積」の名前を紹介しておく。

他に、実践的な量子化学計算パッケージで使われる Lebedev、Euler-Macraurin、MultiExp グリッド等の方法が存在する。

8 演習

10 点グリッドで台形公式を使って面積を求めて見よ。また、得られた結果を exact な解と比較せよ。

9 周期系の場合

周期系の場合、端っこが存在しないので、台形近似でも長方形近似でも全く同じ式になる。

例えば、代表的な周期関数である sin 関数について、[0,2π)の区間で積分してみよう。まずは関数を用意する。

```
start = 0
stop = 2 * np.pi
num = 50
```

```
x = np.linspace(start, stop, num, endpoint=False)
y = np.sin(x)
h = (stop - start) / num
```

一応プロットしておこう。

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

では、長方形近似に則って積分してみよう。

```
np.sum(y) * h
```

ほとんど 0 になっただろう。実際、sin 関数は一周期で積分すると 0 になる。このように、周期関数に対しては長方形近似は非常に精度が良い。