スクリプト言語プログラミング
Pythonによる数値解析

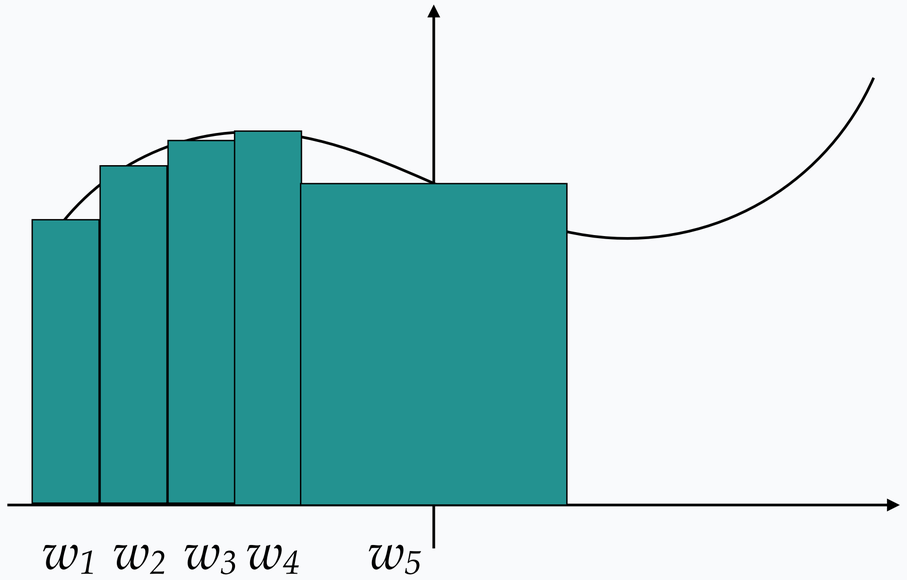
第10回講義資料

箕原辰夫

数値積分

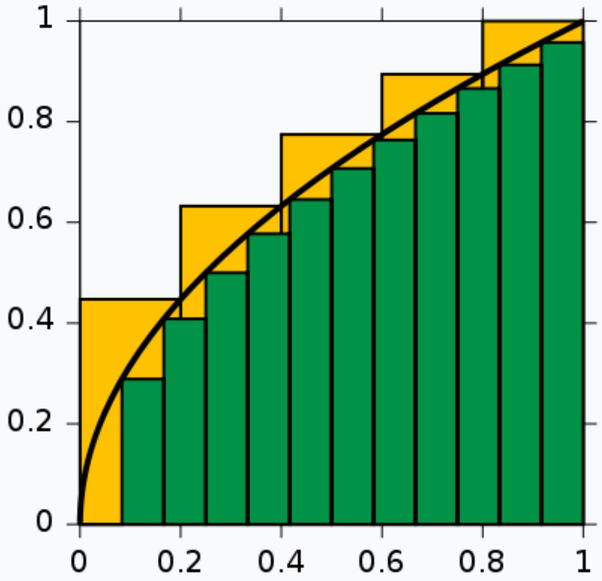
- ガウスの数値積分の公式

$$I = \int_{-1}^{1} f(x)\,dx = \sum_{i=1}^{n} w_i f(x_i)$$

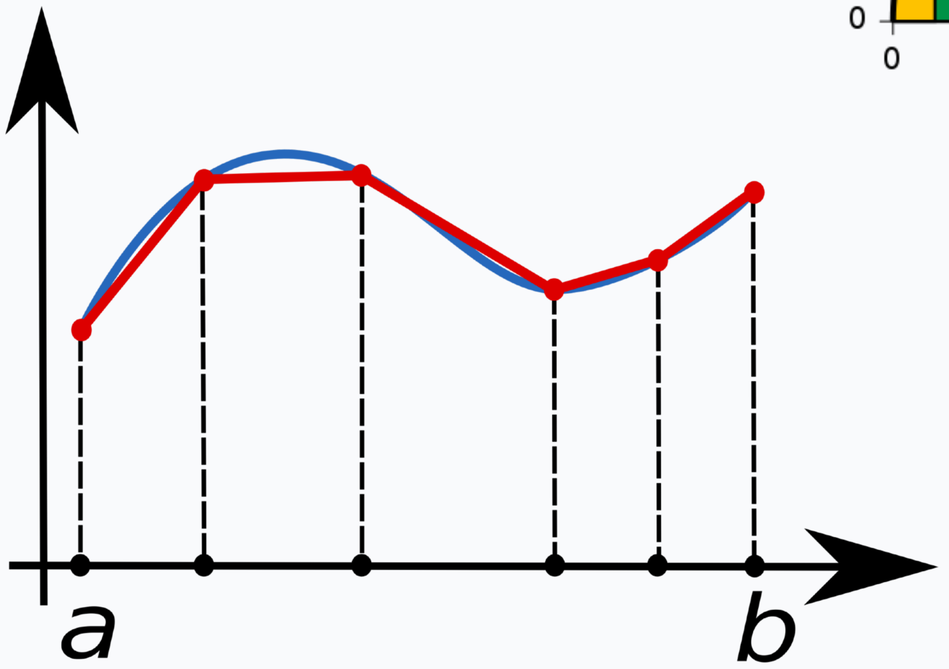


数値積分：矩形近似、台形公式

- 四角形で近似する
  - 内側・外側・中点

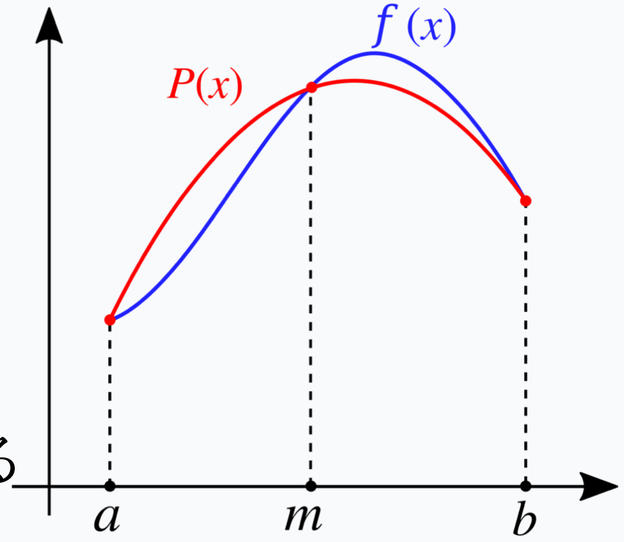


- 台形公式は台形で近似する



数値積分：シンプソンの公式

- シンプソンの公式
  - 2次曲線で近似する
  - f(x)をa, bの中点mで
    交差するP(x)で代替する
  - 積分値は、以下のような式で近似できる



$$\int_a^b f(x)\,dx \approx \int_a^b P(x)\,dx = \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

数値積分の各公式

- ニュートン・コーツの積分公式
  - 区間[a,b]内に(n+1)個の分点$a=x_0, x_1, x_2, \ldots x_n=b$をとり、その点でf(x)と値が一致する補間多項式$P_n(x)$を求めて近似するとき、これを数値積分の公式と呼ぶ。

$$\int_a^b f(x)dx \simeq \int_a^b P_n(x)dx = \sum_{j=0}^{n} a_j f(x_j)$$

- 台形公式
  - n=1のとき

$$\int_a^b f(x)dx \simeq \frac{h}{2}\big(f(a)+f(b)\big), h=b-a$$

- シンプソンの公式
  - n=2のとき

$$\int_a^b f(x)dx \simeq \frac{h}{3}\big(f(a)+4f(a+h)+f(b)\big), h=\frac{b-a}{2}$$

- 3/8シンプソンの公式
  - n=3のとき

$$\int_a^b f(x)dx \simeq \frac{3h}{8}\big(f(a)+3f(a+h)+3f(a+2h)+f(b)\big), h=\frac{b-a}{3}$$

数値積分：合成シンプソンの公式

- composite simpsonの公式では、各区間を分けて公式を適用し、その結果を足し合わせてを使って、積分値を求めていく

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(x_0) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(x_n)\right]$$

- 展開すると以下のようになる

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 4f(x_{n-1}) + f(x_n)\right]$$

- Pythonでの記述（lower〜upperの区間をn個に分割するとする）

```
delta = (upper-lower) / n
h = delta / 2
x = lower
ss = 0.0 # simpsonでの面積

for i in range( n-1 ):
    ss += 4 * f( x + h ) + 2 * f( x + delta )
    x += delta
ss += 4 * f( x + h )
result = h / 3.0 * ( f(lower) + ss + f(upper) )
```

数値積分：合成シンプソン3/8公式

- 合成シンプソン3/8公式は、3次曲線による補間を使い、各区間に適用して、その結果を足しあわせる

$$\int_a^b f(x)\,dx \approx \tfrac{3h}{8}\left[f(x_0) + 3f(x_1) + 3f(x_2) + 2f(x_3) + 3f(x_4) + 3f(x_5) + 2f(x_6) + \cdots + 3f(x_{n-2}) + 3f(x_{n-1}) + f(x_n)\right]$$

$$= \frac{3h}{8}\left[f(x_0) + 3\sum_{i \neq 3k}^{n-1} f(x_i) + 2\sum_{j=1}^{n/3-1} f(x_{3j}) + f(x_n)\right] \qquad \text{For: } k \in \mathbb{N}_0$$

- Pythonで記述する（lower〜upperの区間をn個に分割するとする）

```
n = n // 3 * 3  # 3の倍数にする
h = (upper-lower) / n
x = lower
ss = 0.0
for i in range( 0, n-3,  3 ):
    ss += 3 * f( x + h ) + 3 * f( x + 2*h ) + 2 * f( x+3*h)
    x += 3*h
ss +=  3 * f( x + h ) + 3 * f( x + 2*h )
result = 3.0 * h / 8.0 * ( f(lower) + ss + f(upper) )
```

数値積分：Romberg積分

- 再帰関数を使って求める方法
- 区間[a,b]の$f(x)$の積分値を求める

Using

$$h_n = \frac{1}{2^n}(b-a)$$

the method can be inductively defined by

$$R(0,0) = h_1(f(a)+f(b))$$

$$R(n,0) = \frac{1}{2}R(n-1,0) + h_n \sum_{k=1}^{2^{n-1}} f(a+(2k-1)h_n)$$

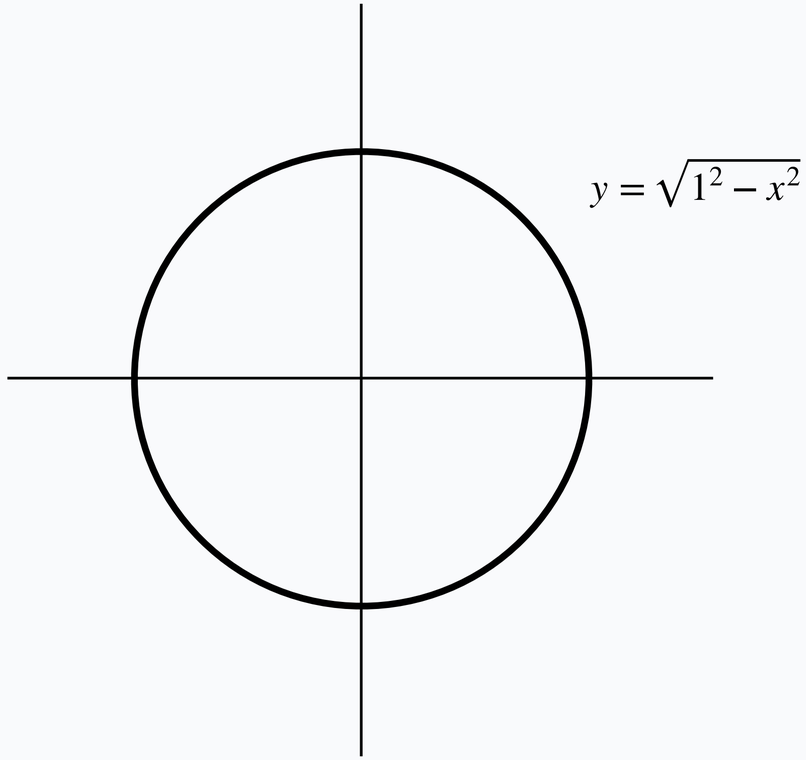
$$R(n,m) = R(n,m-1) + \frac{1}{4^m-1}(R(n,m-1)-R(n-1,m-1))$$

- Wikipedia英語版「Romberg's method」より

https://en.wikipedia.org/wiki/Romberg%27s_method

πの値を数値積分で求める

- f(x)=abs(1-x**2)**0.5と置いておく
  - absを掛けるのはマイナスにしない
  - x=0〜1.0の間で求める



精度（仮数部の桁数）の指定

- Decimalクラスのオブジェクトを利用する

  getcontext().prec = 仮数部の桁数を指定

  getcontext().rounding = 丸め方の指定

- **from** decimal **import** * でモジュールを使う指定をする
- Decimal(値)でその精度の変数を作る

- 例：

```
from decimal import *
getcontext().prec = 100
value2 = Decimal( 2 )
value3 = Decimal( 3 )
print( value2 / value3 )
print( value2.sqrt() )
```

Decimalあるある

- 実数からDecimalにするよりも、文字列からDecimalにした方が精度が高い（実数の場合は、元のIEEE 754上の誤差も入ってしまう）
- 例：

```
getcontext().prec = 28
Decimal('3.14')
    # Decimal('3.14')
Decimal(3.14)
    # Decimal('3.140000000000000124344978758017532527446746826171875')
```

- 丸め方を決められる（正確な四捨五入も可能）
  - ROUND_CEILING, ROUND_DOWN, ROUND_FLOOR, ROUND_HALF_DOWN, ROUND_HALF_EVEN, ROUND_HALF_UP, ROUND_UP, ROUND_05UP がある
- 例：

```
Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
    # Decimal('7.32')
Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
    # Decimal('8')
Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_HALF_UP)
    # Decimal('7.33')
```

Decimalで使えるインスタンスメソッド

- 演算子
  - +, -, *, **, /, //, % などの演算子が使える
  - 例：
    - Decimal("2.5") % Decimal("0.7") → Decimal('0.4')
- 数学関数
  - abs(), exp(), ln(), log10(), sqrt()
- 丸め関数
  - quantize()

decimalモジュールのContextクラスのオブジェクト

- Contextクラスのオブジェクト作成用のメソッド
  - Context(prec=桁数, rounding=小数の最下位の丸め方)
- Decimalクラスのオブジェクト作成用のメソッド
  - create_decimal(string_value), create_decimal_from_float(float_value)
- 演算用の関数
  - add(x, y), divide(x, y), divide_int(x, y), divmod(x, y), minus(x), multiply(x, y), plus(x), power(x, y, modulo=None), remainder(x, y), remainder_near(x, y), subtract(x, y)
- 比較用の関数
  - compare(x, y), compare_signal(x, y), compare_total(x, y), compare_total_mag(x, y), max(x, y), max_mag(x, y), min(x, y), min_mag(x, y), same_quantum(x, y)
  - is_canonical(x), is_finite(x), is_infinite(x), is_nan(x), is_normal(x), is_qnan(x), is_signed(x), is_snan(x), is_subnormal(x), is_zero(x)
- コピー用の関数
  - canonical(x), copy_abs(x), copy_negate(x), copy_sign(x, y)
- 数学関数
  - abs(x), exp(x), fma(x, y, z), ln(x), log10(x), logb(x), scaleb(x, y), sqrt(x)
- 丸め関数
  - next_minus(x), next_plus(x), next_toward(x, y), quantize(x, y), to_integral_exact(x)

decimalモジュールのContextを使う例

- 例：
  - stringPI = "3.14159265358979323846264338327795" # 32 digits in the fraction
  - context = Context(prec=len(stringPI)-2, rounding=ROUND_HALF_UP)
  - decimalPI = context.create_decimal(strPI)
  - print(decimalPI)
  - decimal2 = context.create_decimal_from_float(2)
  - decimal05 = context.create_decimal_from_float(0.5)
  - print(context.power(decimal2, decimal05))

mpmathライブラリ

- 桁数を指定することができる
  - **from** mpmath **import** mp
  - mp.dps = 30 # 10進数での仮数部の桁数（bit数指定ではmp.prec）
- 3つの型変換がある
  - mp.mpf(整数・実数・数字文字列) … 実数型のオブジェクト
  - mp.mpc(実部, 虚部) … 複素数型のオブジェクト
  - mp.matrix(リスト) … 行列型のオブジェクト
- 詳細は、https://mpmath.org/doc/current/を参照

πの値を求める

- 収束度の遅い方法
  - モンテカルロ法で求める（非常に遅い）
  - 数値積分で求める
  - オイラーのζ関数を使う
- 収束度の速い方法（Wikipedia「マチンの公式」参照）
  - オイラーの公式（1748年）
  - マチンの公式（1706年）
  - ガウスの公式（1863年）
  - ストーマーの公式（1896年）
  - 高野喜久雄の公式（1982年）
- 収束度が非常に速い方法
  - ガウス＝ルジャンドルのアルゴリズム
  - チェドノフスキーのアルゴリズム

πを求める式のいくつか

- 級数で求める（収束遅い）
  - ライプニッツの公式
  - ウォリスの公式
  - オイラーのζ(2)関数
- arctanで求める
  - オイラーの公式
  - マチンの公式
  - ガウスの公式
  - ストーマーの公式
  - 高野喜久雄の公式

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

$$\frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \frac{8}{7} \cdot \frac{8}{9} \cdots = \frac{\pi}{2}$$

$$\zeta(2) = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots = \frac{\pi^2}{6}$$

$$\frac{\pi}{4} = \arctan\frac{1}{2} + \arctan\frac{1}{3}$$

$$4\arctan\frac{1}{5} - \arctan\frac{1}{239} = \frac{\pi}{4}$$

$$\frac{\pi}{4} = 12\arctan\frac{1}{18} + 8\arctan\frac{1}{57} - 5\arctan\frac{1}{239}$$

$$\frac{\pi}{4} = 6\arctan\frac{1}{8} + 2\arctan\frac{1}{57} + \arctan\frac{1}{239}$$

$$\frac{\pi}{4} = 12\arctan\frac{1}{49} + 32\arctan\frac{1}{57} - 5\arctan\frac{1}{239} + 12\arctan\frac{1}{110443}$$

πを求めるときのarctan

- テイラー展開で求める
- xの絶対値が1未満のときにしか機能しないので注意

$$\tan^{-1} x = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1} \qquad \text{for } |x| < 1$$

- 複素数を使うと次のようにも求められる（mpmathでの求め方）

$$\tan^{-1}(x) = \frac{i}{2}(\log(1 - ix) - \log(1 + ix))$$

ガウス＝ルジャンドル（Gauss-Legendre）のアルゴリズム

- 円周率を計算する際に用いられる反復計算アルゴリズム
- 円周率を計算するものの中では非常に収束が速い
- 収束は、小数第n桁まで求めるのに、$\log_2 n$回の繰返しで求めることが可能
- アルゴリズムは、以下の通り
  - 初期値の設定

$$a_0 = 1 \quad b_0 = \frac{1}{\sqrt{2}} \quad t_0 = \frac{1}{4} \quad p_0 = 1$$

  - 漸化式

$$a_{n+1} = \frac{a_n + b_n}{2}$$

$$b_{n+1} = \sqrt{a_n b_n}$$

$$t_{n+1} = t_n - p_n \left(a_n - a_{n+1}\right)^2$$

$$p_{n+1} = 2 p_n$$

  - πの算出

$$\pi \approx \frac{(a+b)^2}{4t}$$

チュドノフスキー（Chudnovsky）のアルゴリズム

- Flex Kleinのj関数（j不変関数: j-invariant）に基づいている。

$$j\left(e^{2\pi i/3}\right) = 0, \quad j(i) = 1728 = 12^3.$$

- 負のヘーグナー数 d = -163にj関数を適用してできる数を利用する。

$$j\left(\frac{1+i\sqrt{163}}{2}\right) = -640320^3$$

- この数に急速に収束する一般化超幾何系列から、次の式が求められる。

$$\frac{1}{\pi} = 12\sum_{q=0}^{\infty}\frac{(-1)^q(6q)!(545140134q+13591409)}{(3q)!(q!)^3(640320)^{3q+\frac{3}{2}}}$$

- 高速反復実装の場合、次のように簡略化することができる。

$$\frac{(640320)^{\frac{3}{2}}}{12\pi} = \frac{426880\sqrt{10005}}{\pi} = \sum_{q=0}^{\infty}\frac{(6q)!(545140134q+13591409)}{(3q)!(q!)^3(-262537412640768000)^q}$$

- このアルゴリズムを用いて、2022年3月までに100兆桁のπの値が求められているが、桁数が少なければ、ガウス＝ルジャンドルのアルゴリズムの方が収束度は速い

numbaの高速関数

- Anacondaでは、標準的に用いることができる
- **from** numba **import** jit を記述する
- @jit修飾子を関数の定義の前に入れる
  - その関数は、実行前にコンパイルされるので、高速に動く

timeモジュールのnano second関数

- **import** timeが必要
- 以下の関数が、Python 3.8から追加された
  - time.clock_gettime_ns()
  - time.clock_settime_ns()
  - time.monotonic_ns()
  - time.perf_counter_ns()
  - time.process_time_ns()
  - time.time_ns()

実行速度（性能）測定

- timeitモジュールのtimeit関数での測定、repeat関数は何回か試行ししてみて、そのすべての結果をリストで表示する
  - **import** timeit
  - timeit.timeit('関数呼出しの文字列', globals=globals(), number=実行回数)
  - timeit.repeat(lambda: 関数呼出し, repeat=試行回数, number=実行回数))
  - 文字列としても関数呼出しができるし、引数なしのlambda関数（無名関数）として関数呼出しが記述ができる
- IPythonの%timeit
  - %timeit 関数名
  - %%timeit
Pythonスクリプト
Escapeキーで、スクリプトの入力終了