

Table of Contents

- 1 概要:補間と近似
- 2 多項式補間(polynomial interpolation)
- 3 Lagrange(ラグランジュ) の内挿公式
 - 3.0.1 python code
- 4 Newton(ニュートン) の差分商公式
 - 4.1 Newton補間と多項式補間の一一致の検証
- 5 数値積分 (Numerical integration)
 - 5.1 中点則 (midpoint rule)
 - 5.2 台形則 (trapezoidal rule)
 - 5.3 Simpson(シンプソン)則
- 6 数値積分のコード
 - 6.0.0.1 Midpoint rule(中点法)
 - 6.0.0.2 Trapezoidal rule(台形公式)
 - 6.0.0.3 Simpson's rule(シンプソンの公式)
- 7 課題
 - 7.1 補間法
 - 7.2 対数関数のニュートンの差分商補間(2014期末試験, 25点)
 - 7.2.1 差分商補間の表中の開いている箇所[XXX]を埋めよ.
 - 7.2.2 ニュートンの二次多項式
 - 7.2.3 ニュートンの三次多項式の値を求めよ.
 - 7.3 数値積分(I)
- 8 解答例[7.3]

外へ *extrapolation*

内へ

補間(interpolation)と数値積分(Integral)

file:/Users/bob/Github/TeamNishitani/jupyter_num_calc/interpolationintegral
https://github.com/daddygongon/jupyter_num_calc/tree/master/notebooks_p
cc by Shigeto R. Nishitani 2017

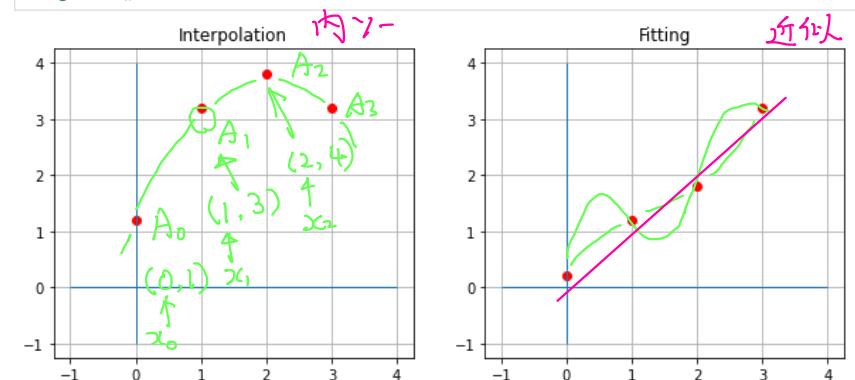
fiting

単純な2次元データについて補間と近似を考える。補間はたんに点を「滑らかに」つなぐことを、近似はある関数にできるだけ近くなるように「フィット」することを言う。補間は

Illustratorなどのドロー系ツールで曲線を引くときの、ベジエやスプライン補間の基本となる。本章では補間とそれに密接に関連した積分について述べる。

In [1]:

```
%matplotlib inline  
  
import matplotlib.pyplot as plt  
import numpy as np  
  
fig, (axL, axR) = plt.subplots(ncols=2, figsize=(10,4))  
  
x = np.array([0,1,2,3])  
y = np.array([1,2,3.2,3.8,3.2])  
for i in range(0,4):  
    axL.plot(x[i],y[i], 'o', color='r')  
    axL.hlines(0, -1, 4, linewidth=1)  
    axL.vlines(0, -1, 4, linewidth=1)  
    axL.set_title('Interpolation')  
    axL.grid(True)  
  
x = np.array([0,1,2,3])  
y = np.array([0.2,1.2,1.8,3.2])  
for i in range(0,4):  
    axR.plot(x[i],y[i], 'o', color='r')  
    axR.hlines(0, -1, 4, linewidth=1)  
    axR.vlines(0, -1, 4, linewidth=1)  
    axR.set_title('Fitting')  
    axR.grid(True)  
  
# fig.show()
```



多項式補間(polynomial interpolation)

データを単純に多項式で補間する方法を先ず示そう。 $\frac{N+1}{4}$ 点を N 次の多項式でつなぐ。この場合の補間関数は、 $\frac{3}{4}$

$$F(x) = \sum_{i=0}^N a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_N x^N$$

である。データの点を $(x_i, y_i), i = 0..N$ とすると

$$\underbrace{a_0 \dots a_N}_{N+1 \text{ 項}}$$

$$\begin{array}{l} \text{3.2} \\ \begin{array}{c} \xleftarrow{\text{N+1 2回}} \\ \begin{array}{l} a_0 + a_1 x_0 + a_2 x_0^2 + \cdots + a_N x_0^N = y_0 \\ a_0 + a_1 x_1 + a_2 x_1^2 + \cdots + a_N x_1^N = y_1 \\ \vdots \\ a_0 + a_1 x_N + a_2 x_N^2 + \cdots + a_N x_N^N = y_N \end{array} \end{array} \end{array}$$

が、係数 a_i を未知数と見なした線形の連立方程式となっている。係数行列は

$$A = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^N \\ 1 & x_1 & x_1^2 & \cdots & x_1^N \\ \vdots & & & & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^N \end{bmatrix} \quad \alpha = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_N \end{pmatrix} \quad y = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_N \end{pmatrix}$$

となる。 a_i と y_i をそれぞれベクトルとみなすと

$$\begin{aligned} A \alpha &= y \\ A^{-1} A \alpha &= A^{-1} y \\ \alpha &= A^{-1} y \end{aligned}$$

により未知数ベクトル α が求まる。これは単純に、前に紹介した Gauss の消去法や LU 分解で解ける。

Lagrange(ラグランジュ) の内挿公式

多項式補間は手続きが簡単であるため、計算間違いが少なく、プログラムとして組むのに適している。しかし、あまり“みとうし”的な方法とはいえない。その点、Lagrange(ラグランジュ) の内挿公式は見通しがよい。これは

$$\rightarrow F(x) = \sum_{k=0}^N \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)} y_k = \sum_{k=0}^N \frac{(x-x_0)(x-x_1)\cdots(x-x_N)}{(x_k-x_0)(x_k-x_1)\cdots(x_k-x_N)} y_k$$

と表わされる。数学的に 2つ目の表記は間違っているが、先に割り算を実行すると読み取って欲しい。これは一見複雑に見えるが、単純な発想から出発している。求めたい関数 $F(x)$ を

$$\rightarrow F(x) = y_0 L_0(x) + y_1 L_1(x) + y_2 L_2(x)$$

とすると

$$\begin{array}{lll} x=x_0 & \begin{array}{c} \text{○} \\ \text{○} \\ \text{○} \end{array} & \begin{array}{c} \text{○} \\ \text{○} \\ \text{○} \end{array} \\ x_1 & \text{○} & \text{○} \\ x_2 & \text{○} & \text{○} \end{array}$$

$$\begin{array}{lll} x_0 & L_0(x_0) = 1 & L_0(x_1) = 0 & L_0(x_2) = 0 \\ x_1 & L_1(x_0) = 0 & L_1(x_1) = 1 & L_1(x_2) = 0 \\ x_2 & L_2(x_0) = 0 & L_2(x_1) = 0 & L_2(x_2) = 1 \end{array}$$

となるように関数 $L_i(x)$ を決めればよい。これを以下のようにすれば Lagrange の内挿公式となる。

$L_0(x)$:

$$\rightarrow L_0(x) = \frac{(x-x_0)(x-x_1)(x-x_2)}{(x_0-x_0)(x_0-x_1)(x_0-x_2)}$$

$L_1(x)$:

$$\rightarrow L_1(x) = \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}$$

$L_2(x)$:

$$\rightarrow L_2(x) = \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}$$

Edge
2次
2点固定
1点固定
PW
力二重
サテ

$$\begin{aligned} & \frac{(x-1)(x-2)}{(0-1)(0-2)} = \frac{(x-1)(x-2)}{2} = \frac{1}{2}x^2 - \frac{3}{2}x + 1 \\ & x=1 \quad \frac{1}{2}-\frac{3}{2}+1=0 \end{aligned}$$

$$\rightarrow F(x) : L_0(x) \cdot y_0 + L_1(x) \cdot y_1$$

python code

python では Lagrange 補間は `interpolate.lagrange` で用意されている。

In [2]:

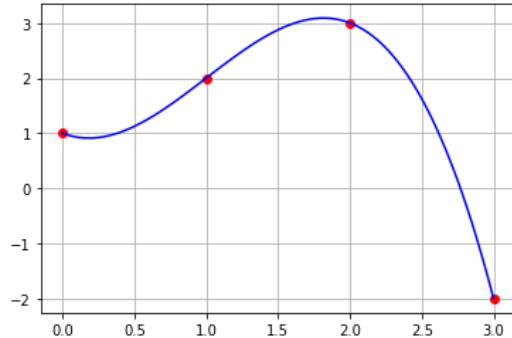
```
import numpy as np
from scipy import interpolate
import matplotlib.pyplot as plt

# もとの点
x = np.array([0, 1, 2, 3])
y = np.array([1, 2, 3, -2])
for i in range(0, 4):
    plt.plot(x[i], y[i], 'o', color='r')

# Lagrange 補間
f = interpolate.lagrange(x, y)
print(f)
x = np.linspace(0, 3, 100)
y = f(x)
plt.plot(x, y, color='b')

plt.grid()
plt.show()
```

$$-1 x^3 + 3 x^2 - 1 x + 1$$



Newton(ニュートン) の差分商公式

もう一つ有名なNewton(ニュートン) の内挿公式は、

$$F(x) = F(x_0) + (x - x_0)f_1[x_0, x_1] + (x - x_0)(x - x_1)f_2[x_0, x_1, x_2] + \dots + \prod_{i=0}^{n-1} (x - x_i) f_n[x_0, x_1, \dots, x_n]$$

となる。ここで $f_i[x]$ は次のような関数を意味していて、

$$\begin{aligned} f_1[x_0, x_1] &= \frac{y_1 - y_0}{x_1 - x_0} \\ f_2[x_0, x_1, x_2] &= \frac{f_1[x_1, x_2] - f_1[x_0, x_1]}{x_2 - x_0} \\ &\vdots \\ f_n[x_0, x_1, \dots, x_n] &= \frac{f_{n-1}[x_1, x_2, \dots, x_n] - f_{n-1}[x_0, x_1, \dots, x_{n-1}]}{x_n - x_0} \end{aligned}$$

差分商と呼ばれる。得られた多項式は、Lagrange の内挿公式で得られたものと当然一致する。Newtonの内挿公式の利点は、新たなデータ点が増えたときに、新たな項を加えるだけで、内挿式が得られる点である。

```
In [3]: # https://stackoverflow.com/questions/14823891/newton-s-interpolating-polynomial-python
# by Khalil Al Hooti (stackoverflow)
```

```
def _poly_newton_coefficient(x,y):
    """
    x: list or np array containing x data points
    y: list or np array containing y data points
    """

    m = len(x)

    x = np.copy(x)
    a = np.copy(y)
    for k in range(1,m):
        a[k:m] = (a[k:m] - a[k-1]) / (x[k:m] - x[k-1])

    return a

def newton_polynomial(x_data, y_data, x):
    """
    x_data: data points at x
```

y_data: data points at y

x: evaluation point(s)

"""

```
a = _poly_newton_coefficient(x_data, y_data)
n = len(x_data) - 1 # Degree of polynomial
p = a[n]
for k in range(1,n+1):
    p = a[n-k] + (x - x_data[n-k])*p
return p
```

In [4]:

```
import numpy as np
from scipy import interpolate
import matplotlib.pyplot as plt

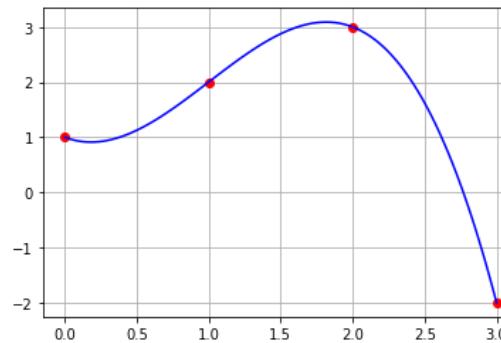
# もとの点
x = np.array([0,1,2,3])
y = np.array([1,2,3,-2])
for i in range(0,4):
    plt.plot(x[i],y[i],'o',color='r')

print(_poly_newton_coefficient(x,y))

xx = np.linspace(0,3, 100)
yy = newton_polynomial(x, y, xx)
plt.plot(xx, yy, color = 'b')

plt.grid()
plt.show()
```

[1 1 0 -1]



⇒ Newton補間と多項式補間の一一致の検証

関数 $F(x)$ を x の多項式として展開。その時の、係数の取るべき値と、差分商で得られる値が一致。

```
maple
> restart: F:=x->f0+(x-x0)*f1p+(x-x0)*(x-x1)*f2p;
F := x → f0 + (x - x0)f1p + (x - x0)(x - x1)f2p

maple
> F(x1);
sf1p:=solve(F(x1)=f1,f1p);
```

$$f0 + (x1 - x0)f1p$$

$$sf1p := \frac{f0 - f1}{-x1 + x0}$$

f20の取るべき値の導出

```
maple
> sf2p:=solve(F(x2)=f2,f2p);
fac_f2p:=factor(subs(f1p=sf1p,sf2p));
sf2p := -\frac{f0+f1p\,x2-f1p\,x0-f2}{(-x2+x0)(-x2+x1)}
fac_f2p := \frac{f0\,x1-x2\,f0+x2\,f1-x0\,f1-f2\,x1+f2\,x0}{(-x1+x0)(-x2+x0)(-x2+x1)}
```

ニュートンの差分商公式を変形

```
maple
> ff11:=(f0-f1)/(x0-x1);
ff12:=(f1-f2)/(x1-x2);
ff2:=(ff11-ff12)/(x0-x2);
fac_newton:=factor(ff2);
```

$$ff11 := \frac{f0 - f1}{-x1 + x0}$$

$$ff12 := \frac{f1 - f2}{-x2 + x1}$$

$$ff2 := \frac{\frac{f0 - f1}{-x1 + x0} - \frac{f1 - f2}{-x2 + x1}}{-x2 + x0}$$

$$fac_newton := \frac{f0\,x1 - x2\,f0 + x2\,f1 - x0\,f1 - f2\,x1 + f2\,x0}{(-x1 + x0)(-x2 + x0)(-x2 + x1)}$$

二式が等しいかどうかをevalbで判定

```
maple
> evalb(fac_f2p=fac_newton);
true
```

数値積分 (Numerical integration)

In [5]:

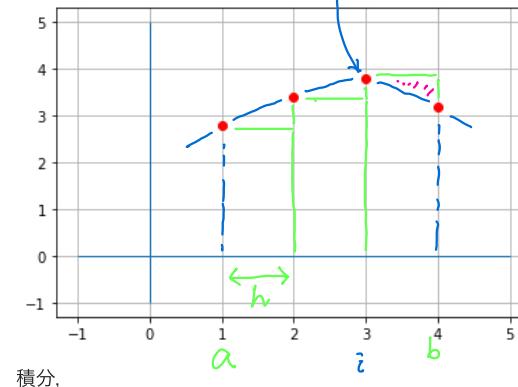
```
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

x = np.array([1,2,3,4])
y = np.array([2.8,3.4,3.8,3.2])
for i in range(0,4):
    plt.plot(x[i],y[i],'o',color='r')
plt.hlines(0,-1, 5, linewidth=1)
plt.vlines(0,-1, 5, linewidth=1)
```

plt.grid(True)

数値積分の模式図



を求める。1次元の数値積分法では連続した領域を細かい短冊に分けて、それぞれの面積を寄せ集めることに相当する。分点の数を N とすると、

$$x_i = a + \frac{b-a}{N}i = a + h \times i$$

$$h = \frac{b-a}{N}$$

となる。そうすると、もっとも単純には、

$$I_N = \left\{ \sum_{i=0}^{N-1} f(x_i) \right\} h = \left\{ \sum_{i=0}^{N-1} f(a + i \times h) \right\} h$$

となる。

中点則 (midpoint rule)

中点法 (midpoint rule) は、短冊を左端から書くのではなく、真ん中から書くことに対応し、

$$I_N = \left\{ \sum_{i=0}^{N-1} f \left(a + \left(i + \frac{1}{2} \right) \times h \right) \right\} h$$

となる。

台形則 (trapezoidal rule)

さらに短冊の上側を斜めにして、短冊を台形にすれば精度が上がりそうに思う。その場合は、短冊一枚の面積 S_i は、

$$S_i = \frac{f(x_i) + f(x_{i+1})}{2} h$$

で求まる。これを端から端まで加えあわせると、

$$I_N = \sum_{i=0}^{N-1} S_i = h \left\{ \frac{1}{2} f(x_0) + \sum_{i=1}^{N-1} f(x_i) + \frac{1}{2} f(x_N) \right\}$$

が得られる。

Simpson(シンプソン)則

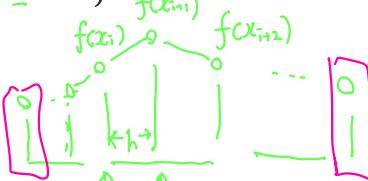
Simpson(シンプソン)則では、短冊を2次関数、

$$f(x) = ax^2 + bx + c$$

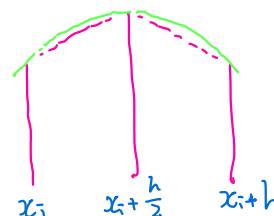
で近似することに対応する。こうすると、

$$S_i = \int_{x_i}^{x_{i+1}} f(x) dx = \int_{x_i}^{x_{i+1}} (ax^2 + bx + c) dx$$

$$= \frac{1}{3} (f(x_{i-1}) + 4f(x_i) + f(x_{i+1})) h$$



Simpson則の導出(数式変形) :



$$\frac{h}{6} \left\{ f(x_i) + 4f\left(x_i + \frac{h}{2}\right) + f(x_{i+1}) \right\}$$

となる。これより、

$$\rightarrow I_N = \frac{h}{6} \left\{ f(x_0) + 4 \sum_{i=0}^{N-1} f\left(x_i + \frac{h}{2}\right) + 2 \sum_{i=1}^{N-1} f(x_i) + f(x_N) \right\}$$

として計算できる。ただし、関数値を計算する点の数は台形則などの倍となっている。

教科書によつては、分割数Nを偶数にして、点を偶数番目(even)と奇数番目(odd)に分けて、

$$\rightarrow I_N = \frac{h}{3} \left\{ f(x_0) + 4 \sum_{i=even}^{N-2} f\left(x_i + \frac{h}{2}\right) + 2 \sum_{i=odd}^{N-1} f(x_i) + f(x_N) \right\}$$

としている記述があるが、同じ計算になるので誤解せぬよう。

数値積分のコード

次の積分を例に、pythonのコードを示す。

$$\int_0^1 \frac{4}{1+x^2} dx$$

*Newton
cotes*

*Open
formula*

まずは問題が与えられたらできるだけお任せで解いてしまう。答えをあらかじめ知っておくと間違いを見つけるのが容易、プロットしてみる。

scipyで積分計算をお任せしてくれる関数はquadで、これはFortran libraryのQUADPACKを利用している。自動で色々してくれるが、精度は1.49e-08以下。

In [6]:

```
import matplotlib.pyplot as plt
from scipy import integrate
```

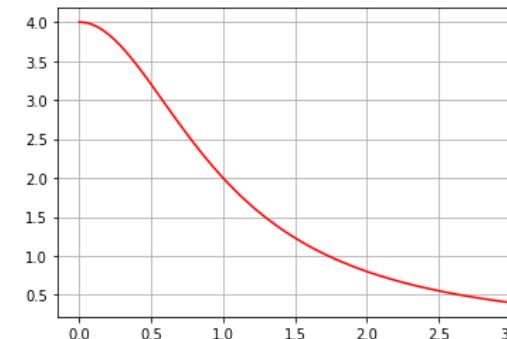
```
def func(x):
    return 4.0/(1.0+x**2)
```

```
x = np.linspace(0,3,100)
y = func(x)
```

```
plt.plot(x,y,color='r')
```

```
plt.grid()
plt.show()
```

```
print(integrate.quad(func,0,1))
```



(3.1415926535897936, 3.4878684980086326e-14)

なんでと思うかもしれないが、

maple
>int(1/(1+x^2),x);

arctan(x)

となるので、納得できるでしょう。

Midpoint rule(中点法)

In [7]:

```
def func(x):
    return 4.0/(1.0+x**2)
```

```
N, x0, xn = 8, 0.0, 1.0
```

```
h = (xn-x0)/N
```

```
S = 0.0
```

```
for i in range(0,N):
```

```
    xi = x0 + (i+0.5)*h
```

```
    dS = h * func(xi)
```

```
    S = S + dS
```

```
print(S)
```

3.142894729591689

Trapezoidal rule(台形公式)

```
In [8]: def func(x):
    return 4.0/(1.0+x**2)
```

N, x0, xn = 8, 0.0, 1.0

h = (xn-x0)/N

S = func(x0)/2.0

for i in range(1, N):

xi = x0 + i*h

dS = func(xi)

S = S + dS

print("{0}".format(i))

S = S + func(xn)/2.0
print(h*S)

```
1
2
3
4
5
6
7
3.138988494491089
```

Simpson's rule(シンプソンの公式)

```
In [9]: def func(x):
    return 4.0/(1.0+x**2)
```

N, x0, xn = 8, 0.0, 1.0

M = int(N/2)

h = (xn-x0)/N

Seven, Sodd = 0.0, 0.0

for i in range(1, 2*M, 2): #rangeの終わりに注意

xi = x0 + i*h

Sodd += func(xi)

print("{0}".format(i))

for i in range(2, 2*M, 2):

xi = x0 + i*h

Seven += func(xi)

print("{0}".format(i))

print(h*(func(x0)+4*Sodd+2*Seven+func(xn))/3)

```
1
2
3
4
5
6
3.141592502458707
```

課題

補間法

次の4点

maple

x y

0 1

1 2

2 3

3 1

を通る多項式を逆行列で求めよ。

対数関数のニュートンの差分商補間(2014期末試験, 25点)

2を底とする対数関数(Mapleでは $\log[2](x)$)の $F(9.2) = 2.219203$ をニュートンの差分商補間を用いて求める。ニュートンの内挿公式は、

$$F(x) = F(x_0) + (x - x_0)f_1[x_0, x_1] + (x - x_0)(x - x_1)f_2[x_0, x_1, x_2] + \cdots + \prod_{i=0}^{n-1}(x - x_i)f_n[x_0, x_1, \dots, x_n]$$

である。ここで $f_i[x]$ は次のような関数を意味していて、

対数値

$$\begin{aligned} f_1[x_0, x_1] &= \frac{y_1 - y_0}{x_1 - x_0} \\ f_2[x_0, x_1, x_2] &= \frac{f_1[x_1, x_2] - f_1[x_0, x_1]}{x_2 - x_0} \\ &\vdots \\ f_n[x_0, x_1, \dots, x_n] &= \frac{f_{n-1}[x_1, x_2, \dots, x_n] - f_{n-1}[x_0, x_1, \dots, x_{n-1}]}{x_n - x_0} \end{aligned}$$

差分商と呼ばれる。 $x_k = 8.0, 9.0, 10.0, 11.0$ をそれぞれ選ぶと、差分商補間のそれぞれの項は以下の通りとなる。

k	x_k	$y_k = F_0(x_k)$	$f_1[x_k, x_{k+1}]$	$f_2[x_k, x_{k+1}, x_{k+2}]$	$f_3[x_k, x_{k+1}, x_{k+2}, x_{k+3}]$
0	8.0	2.079442			
1	9.0	2.197225	0.117783		[XXX]
2	10.0	2.302585	0.105360	-0.0050250	0.0003955000
3	11.0	2.397895	0.095310		

それぞれの項は、例えば、

$$f_2[x_1, x_2, x_3] = \frac{0.095310 - 0.105360}{11.0 - 9.0} = -0.0050250$$

で求められる。ニュートンの差分商の一次多項式の値は $x=9.2$ で

$$F(x) = F_0(8.0) + (x - x_0)f_1[x_1, x_0] = 2.079442 + 0.117783(9.2 - 8.0) = 2.220782$$

となる。

差分商補間の表中の開いている箇所[XXX]を埋めよ。

ニュートンの二次多項式

$$F(x) = F(x_0) + (x - x_0)f_1[x_0, x_1] + (x - x_0)(x - x_1)f_2[x_0, x_1, x_2]$$

の値を求めよ。

ニュートンの三次多項式の値を求めよ。

ただし、ここでは有効数字7桁程度はとるように。(E.クライツィグ著「数値解析」(培風館,2003), p.31, 例4改)

数値積分(I)

次の関数

$$f(x) = \frac{4}{1+x^2}$$

を $x = 0..1$ で数値積分する。

1. N を2,4,8,...256ととり、 N 個の等間隔な区間にわけて中点法と台形則で求めよ。(15)
2. 小数点以下10桁まで求めた値3.141592654との差を dX とする。 dX と分割数 N とを両対数プロット(loglogplot)して比較せよ(10) (2008年度期末試験)

解答例[7.3]

以下には、中点則の結果を示した。課題では、台形則を加えて、両者を比較せよ。予測とどう違うか。

In [10]:

```
import numpy as np

def func(x):
    return 4.0/(1.0+x**2)

def mid(N):
    x0, xn = 0.0, 1.0

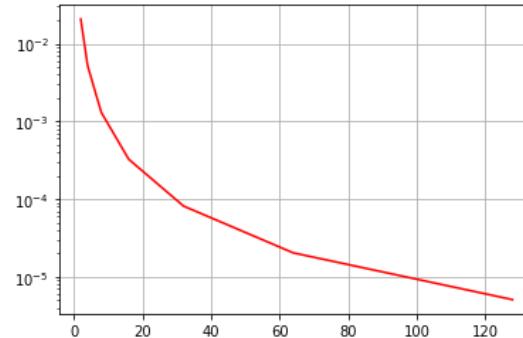
    h = (xn-x0)/N
    S = 0.0
    for i in range(0, N):
        xi = x0 + (i+0.5)*h
        dS = h * func(xi)
        S = S + dS
    return S
```

x, y = [], []

絶対値
absolute value

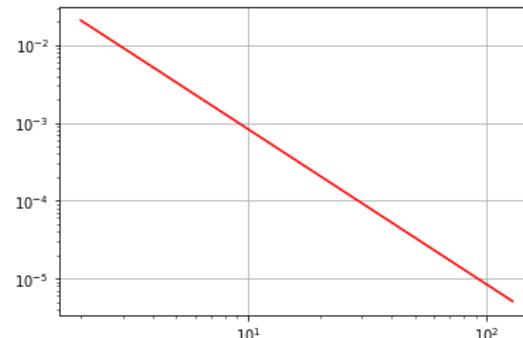
```
for i in range(1,8):
    x.append(2**i)
    y.append(abs(mid(2**i)-np.pi))

In [11]: plt.plot(x, y, color = 'r')
plt.yscale('log')
plt.grid()
plt.show()
```



In [12]:

```
plt.plot(x, y, color = 'r')
plt.yscale('log')
plt.xscale('log')
plt.grid()
plt.show()
```



In []: