2.10.3　台形則, 中点則, Simpson 則の比較

例 2.10.2 (滑らかな関数の積分) やはり

$$I = \int_1^2 \frac{dx}{x} = \log 2 = 0.69314718\cdots$$

を、3 つの方法で計算して、結果を比較する。

誤差の表

# N	台形則の誤差	中点則の誤差	Simpson 則の誤差
1	-5.685282e-02	2.648051e-02	-1.297264e-03
2	-1.518615e-02	7.432895e-03	-1.067877e-04
4	-3.876629e-03	1.927289e-03	-7.350095e-06
8	-9.746698e-04	4.866265e-04	-4.722595e-07
16	-2.440216e-04	1.219662e-04	-2.972988e-08
32	-6.102771e-05	3.051106e-05	-1.861510e-09
64	-1.525832e-05	7.628987e-06	-1.163973e-10
128	-3.814668e-06	1.907323e-06	-7.275514e-12
256	-9.536725e-07	4.768356e-07	-4.551914e-13
512	-2.384185e-07	1.192092e-07	-2.797762e-14
1024	-5.960464e-08	2.980232e-08	-2.220446e-15
2048	-1.490116e-08	7.450581e-09	-2.220446e-16
4096	-3.725290e-09	1.862645e-09	2.220446e-16
8192	-9.313247e-10	4.656625e-10	2.220446e-16
16384	-2.328292e-10	1.164144e-10	-1.110223e-16
32768	-5.820444e-11	2.910883e-11	4.329870e-15
65536	-1.455958e-11	7.274403e-12	-3.663736e-15

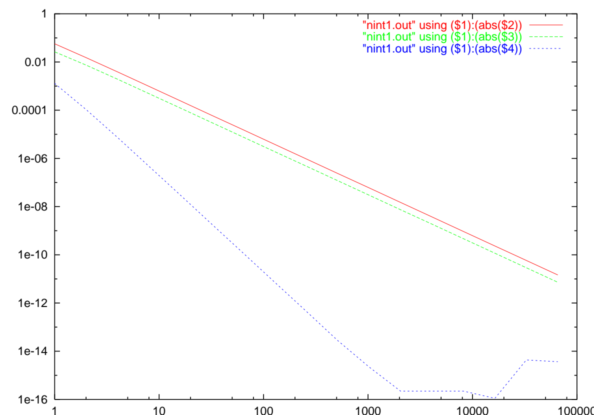
数表では今一つ分かりにくいでの、両側対数グラフにプロットしてみる。ここで用いているグラフ描画ソフト gnuplot については、

http://nalab.mind.meiji.ac.jp/~mk/labo/howto/index.html#gnuplot

特に

『gnuplot 入門』 by 桂田祐史
http://nalab.mind.meiji.ac.jp/~mk/labo/howto/intro-gnuplot/

を参照せよ。



gnuplot 用プログラム show_error1.gp

```
# show_nint1.gp --- gnuplot 用プログラム
#   ./nint1 > nint1.out として作ったデータを読んでグラフを描く
#   結果を nint1.eps に出力
set logscale xy
plot "nint1.out" using ($1):(abs($2)) with lines, \
     "nint1.out" using ($1):(abs($3)) with lines, \
     "nint1.out" using ($1):(abs($4)) with lines
pause -1 "終了するにはリターンを押してください"
set term postscript eps color
set output "nint1.eps"
replot
```

nint1.c

```
/*
 * nint1.c --- 1/x の [1,2] における定積分を
 *           複合台形則, 複合中点則, 複合 Simpson 則で計算して誤差を比較
 *
 *  コンパイル: gcc -o nint1 nint1.c nint.o -lm
 *    ただし nint.o は gcc -c nint.c として準備しておく。
 */

#include <stdio.h>
#include <math.h>
#include "nint.h"

rrfunction f;

int main()
{
  int N;
  double I = log(2.0), Tm, Mm, S2m;

  printf("#   N   台形則の誤差  中点則の誤差  Simpson 則の誤差\n");
  for (N = 1; N <= (1 << 16); N *= 2) {
    /* Tm: 台形則, Mm: 中点則, S2m: Simpson 則 */
    Tm = trapezoidal(f, 1.0, 2.0, N);
    Mm = midpoint(f, 1.0, 2.0, N);
    S2m = (Tm + 2 * Mm) / 3;
    printf("%5d\t%e\t%e\t%e\n", N, I - Tm, I - Mm, I - S2m);
  }
  return 0;
}

/* 被積分関数 */
double f(double x)
{
  return 1 / x;
}
```

これから

$$|I - S_{2m}| \ll |I - T_m|, |I - M_m|.$$

より詳しくは

$$I - T_m = O\left(\frac{1}{m^2}\right), \quad I - M_m = O\left(\frac{1}{m^2}\right), \quad I - S_{2m} = O\left(\frac{1}{m^4}\right)$$

という挙動を示していることが分る。また

$$(I - T_m) : (I - M_m) \fallingdotseq 2 : (-1)$$

となっていることも分かる。■

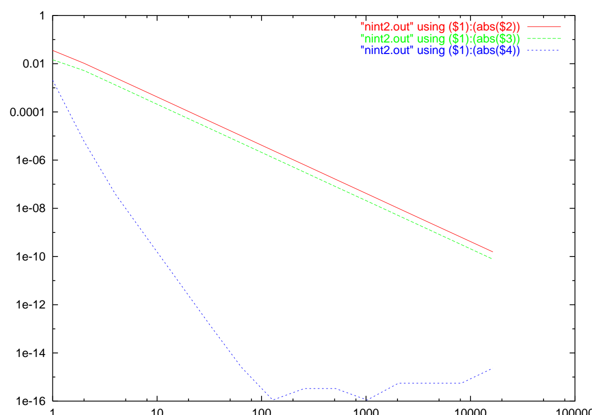
例 2.10.3 (Simpson 則がやけに高精度な例)

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

に対して $|I-S_n|$ は異常に小さくなることが知られている。

誤差の表

# N	台形則の誤差	中点則の誤差	Simpson 則の誤差
1	3.539816e-02	-1.460184e-02	2.064830e-03
2	1.039816e-02	-5.190072e-03	6.006535e-06
4	2.604046e-03	-1.301966e-03	3.778277e-08
8	6.510398e-04	-3.255190e-04	5.912427e-10
16	1.627604e-04	-8.138018e-05	9.239165e-12
32	4.069010e-05	-2.034505e-05	1.442180e-13
64	1.017253e-05	-5.086263e-06	2.553513e-15
128	2.543132e-06	-1.271566e-06	-1.110223e-16
256	6.357829e-07	-3.178914e-07	3.330669e-16
512	1.589457e-07	-7.947286e-08	-3.330669e-16
1024	3.973643e-08	-1.986821e-08	1.110223e-16
2048	9.934108e-09	-4.967053e-09	5.551115e-16
4096	2.483527e-09	-1.241763e-09	5.551115e-16
8192	6.208802e-10	-3.104410e-10	-5.551115e-16
16384	1.552228e-10	-7.760781e-11	2.331468e-15



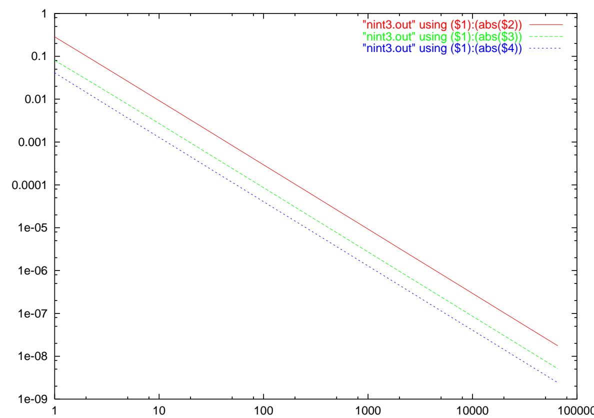
例 2.10.4 (積分区間に特異点が含まれる例) 積分区間に特異点を含む定積分

$$I=\int_0^1 \sqrt{1-x^2}\,dx$$

に対しては、複合台形則、複合中点則、複合 Simpson 則のいずれも低い精度しかでない。

誤差の表

# N	台形則の誤差	中点則の誤差	Simpson 則の誤差
1	2.853982e-01	-8.062724e-02	4.138123e-02
2	1.023855e-01	-2.944367e-02	1.449937e-02
4	3.647090e-02	-1.058414e-02	5.100871e-03
8	1.294338e-02	-3.773569e-03	1.798746e-03
16	4.584904e-03	-1.339789e-03	6.351089e-04
32	1.622558e-03	-4.746873e-04	2.243944e-04
64	5.739352e-04	-1.680046e-04	7.930866e-05
128	2.029653e-04	-5.942998e-05	2.803511e-05
256	7.176766e-05	-2.101722e-05	9.911071e-06
512	2.537522e-05	-7.431692e-06	3.503944e-06
1024	8.971763e-06	-2.627674e-06	1.238805e-06
2048	3.172045e-06	-9.290536e-07	4.379792e-07
4096	1.121496e-06	-3.284755e-07	1.548482e-07
8192	3.965100e-07	-1.161346e-07	5.474696e-08
16384	1.401877e-07	-4.105994e-08	1.935595e-08
32768	4.956389e-08	-1.451691e-08	6.843354e-09
65536	1.752349e-08	-5.132508e-09	2.419491e-09



例 2.10.5 解析的周期関数の数値積分を見てみよう。n 次の第 1 種 Bessel 関数 $J_n(x)$ は

$$J_n(x) = \frac{1}{2\pi}\int_{-\pi}^{\pi} \cos(nt - x\sin t)\,dt$$

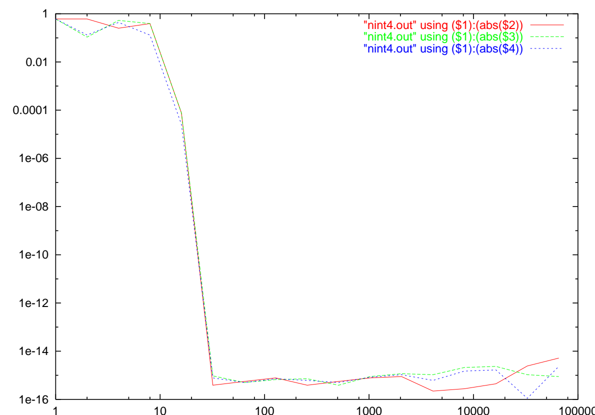
と積分表示できるが、これは解析的周期関数の一周期に渡る積分だから、台形則で非常に精密に計算できるはずである。$n=4$, $x=5$ のときの値、すなわち $J_4(5)$ の値を見てみよう。

nint4.c

```
/*
 * nint4.c --- 解析的周期関数の数値積分
 *
 *  n 次の第一種 Bessel 関数 J_n(x) は
 *
 *            1    π
 *    J_n(x)=---∫   cos (n t - x sin t) dt
 *           2π  -π
 *
 * と積分表示できるが、これは解析的周期関数だから、台形則で非常に
 * 精密に計算できるはずである。
 *
 * UNIX の数学関数ライブラリィには jn(int,double) という名前で関数が
 * 用意されているので、これと比較してみる。
 *
 *  コンパイル: gcc -o nint4 nint4.c nint.o -lm
 *    ただし nint.o は gcc -c nint.c として準備しておく。
 */

#include <stdio.h>
#include <math.h>
#include "nint.h"

rrfunction f;

/* n 次の Bessel 関数の x での値に注目 */
int n;
double x;

/* 被積分関数 */
double f(double t)
{
  return cos(n * t - x * sin(t));
}

int main()
{
  int m;
  double pi = 4.0 * atan(1.0), I, T, M, S;

  /* J4(5) */
  n = 4; x = 5.0;

  /* ライブラリィ関数 */
  I = jn(n, x);

  printf("# 解析的周期関数 cos(%d t - %g sin t) の [-π, π] における積分\n",
 n, x);
  printf("# 複合台形則 T_m, 複合中点則 M_m, 複合 Simpson 則 S_{2m} の誤差\n");
  printf("    m         I-T_m          I-M_m         I-S_{2m}\n");
  for (m = 1; m <= (1 << 16); m *= 2) {
    /* T: 台形則, M: 中点則, S: Simpson 則 */
    T = trapezoidal(f, - pi, pi, m) / (2 * pi);
    M = midpoint(f, - pi, pi, m) / (2 * pi);
    S = (T + 2 * M) / 3;
    printf("  %5d\t%14e\t%14e\t%14e\n", m, I - T, I - M, I - S);
  }
  return 0;
}
```

誤差の表

```
# 解析的周期関数 cos(4 t - 5 sin t) の [-π, π] における積分
# 複合台形則 T_m, 複合中点則 M_m, 複合 Simpson 則 S_{2m} の誤差
    m         I-T_m          I-M_m          I-S_{2m}
    1  -6.087676e-01   -6.087676e-01   -6.087676e-01
    2  -6.087676e-01    1.075702e-01   -1.312091e-01
    4  -2.505987e-01   -5.321711e-01   -4.383136e-01
    8  -3.913849e-01    3.912324e-01    1.303599e-01
   16  -7.627816e-05    7.627816e-05    2.542605e-05
   32  -3.885781e-16   -9.436896e-16   -7.771561e-16
   64  -5.551115e-16   -4.996004e-16   -4.996004e-16
  128  -7.771561e-16   -6.661338e-16   -7.216450e-16
  256  -3.885781e-16   -7.216450e-16   -6.106227e-16
  512  -5.551115e-16   -3.885781e-16   -4.996004e-16
 1024  -7.771561e-16   -8.881784e-16   -8.326673e-16
 2048  -8.881784e-16   -1.165734e-15   -1.054712e-15
 4096   2.220446e-16   -1.054712e-15   -6.106227e-16
 8192  -2.775558e-16   -2.109424e-15   -1.498801e-15
16384  -4.440892e-16   -2.331468e-15   -1.665335e-15
32768  -2.442491e-15    1.054712e-15   -1.110223e-16
65536  -5.162537e-15   -8.881784e-16   -2.331468e-15
```



かなり小さな分割数 m に対して非常に高精度の値が得られている。$m = 32$ のとき、複合台形則 T_{32}, 複合中点則 M_{32} ともに 10^{-16} 程度と double 型の計算精度一杯の値が得られている (実は丸め誤差がなければ 3.7×10^{-19} 程度の値であるとか)。一方で複合 Simpson 則 S_{32} については $I - S_{32} \fallingdotseq 2.5 \times 10^{-5}$ であまり精度が出ていない。(グラフでは同じ m について T_m, M_m, S_{2m} を比較しているので同程度の精度に見えるが、本来は被積分関数の計算回数を揃えて比較すべきであろう。そうすると Simpson 則は他の二つの方法に比べて見劣りすることが分る。)

2.10.4 数直線上の解析関数の数値積分

例 2.10.6 確率積分

$$\int_{-\infty}^{\infty} e^{-x^2}\,dx = \sqrt{\pi}$$

は数直線上の解析関数だから台形則で高精度に計算できるはず。

nint5.c

```
/*
 * nint5.c --- 確率積分
 *          ∞
 *    I =∫    exp(-x^2)dx = √π
 *        -∞
 * は R 上の解析関数の積分だから、台形則
 *              ∞
 *    T_h = h Σ    f(n h)            (ただし f は被積分関数)
 *            n=-∞
 * あるいは、その打ち切り
 *                 N
 *    T_{h,N} = h Σ    f(n h)
 *               n=-N
 * で非常に精密に計算できるはずである。
 *
 *  コンパイル: gcc -o nint5 nint5.c -lm
 */

#include <stdio.h>
#include <math.h>

typedef double rrfunction(double);
rrfunction f;
double trapezoidal2(rrfunction, double, int);

/* 被積分関数 */
double f(double x)
{
  return exp(- x * x);
}

int main()
{
  int m, N;
  double pi, I, h, T;

  /* 円周率, 確率積分の真値 */
  pi = 4.0 * atan(1.0); I = sqrt(pi);

  printf(" m      h              I-T\n");
  for (m = 1; m <= (1 << 2); m *= 2) {
    h = 1.0 / m;
    /* [-6,6] で打ち切る */
    N = m * 6;
    T = trapezoidal2(f, h, N);
    printf("%2d\t%g\t%14e\n", m, h, I - T);
  }
  return 0;
}

double trapezoidal2(rrfunction f, double h, int N)
{
  int j;
  double T = 0.0;
  for (j = - N; j <= N; j++) T += f(j * h);
  T *= h;
  return T;
}
```

誤差の表

```
m      h            I-T
1      1        -1.833539e-04
2      0.5      -2.220446e-16
4      0.25     -4.440892e-16
```

2.10.5 DE 公式

例 2.10.7 (端点の特異性くらい何でもない)

$$I=\int_{-1}^{1}\sqrt{1-x^2}\,dx=\frac{\pi}{2},\quad J=\int_{-1}^{1}\frac{1}{\sqrt{1-x^2}}\,dx=\pi.$$

誤差の表

```
test1 (sqrt(1-x^2) の積分)
 h=1.000000, I_h=       1.7125198292703636, I_h-I=1.417235e-01
 h=0.500000, I_h=       1.5709101233831164, I_h-I=1.137966e-04
 h=0.250000, I_h=       1.5707963267997540, I_h-I=4.857448e-12
 h=0.125000, I_h=       1.5707963267948970, I_h-I=4.440892e-16
 h=0.062500, I_h=       1.5707963267948970, I_h-I=4.440892e-16
 h=0.031250, I_h=       1.5707963267948954, I_h-I=-1.110223e-15
 h=0.015625, I_h=       1.5707963267948979, I_h-I=1.332268e-15
 h=0.007812, I_h=       1.5707963267948957, I_h-I=-8.881784e-16
 h=0.003906, I_h=       1.5707963267948959, I_h-I=-6.661338e-16
 h=0.001953, I_h=       1.5707963267948954, I_h-I=-1.110223e-15
test2 (1/sqrt(1-x^2) の積分)
 h=1.000000, I_h=       3.1435079763395435, I_h-I=1.915323e-03
 h=0.500000, I_h=       3.1415926717394895, I_h-I=1.814970e-08
 h=0.250000, I_h=       3.1415926194518016, I_h-I=-3.413799e-08
 h=0.125000, I_h=       3.1415926318228000, I_h-I=-2.176699e-08
 h=0.062500, I_h=       3.1415926343278695, I_h-I=-1.926192e-08
 h=0.031250, I_h=       3.1415926326210664, I_h-I=-2.096873e-08
 h=0.015625, I_h=       3.1415926323669550, I_h-I=-2.122284e-08
 h=0.007812, I_h=       3.1415926327540102, I_h-I=-2.083578e-08
 h=0.003906, I_h=       3.1415926312582481, I_h-I=-2.233155e-08
 h=0.001953, I_h=       3.1415926319069580, I_h-I=-2.168284e-08
```

nint6.c

```
/*
 * nint6.c --- DE 公式
 *
 *          1      2 (1/2)
 *  I1 = ∫    (1-x )     dx = π/2
 *        -1
 *
 *          1      2 (-1/2)
 *  I2 = ∫    (1-x )      dx = π
 *        -1
 *
 *  いずれも端点に特異性があって、古典的な数値積分公式はうまく行かない。
 *  double exponential formula (DE 公式) ならばうまく計算できる。
 *
 *  コンパイル: gcc -o nint6 nint6.c -lm
 *
 *  学生 (横山和正君) の指摘により少し修正を加える (2004/1/15)。
 */
```

```
#include <stdio.h>
#include <math.h>

typedef double rrfunction(double);

double debug = 0;
double pi, halfpi;

/* φ */
double phi(double t)
{
  return tanh(halfpi * sinh(t));
}

/* 2乗 */
double sqr(double x) { return x * x; }

/* φ' */
double dphi(double t)
{
  return halfpi * cosh(t) / sqr(cosh(halfpi * sinh(t)));
}

/* DE 公式による (-1,1) における定積分の計算 */
double de(rrfunction f, double h, double N)
{
  int n;
  double t, S, dS;
  S = f(phi(0.0)) * dphi(0.0);
  for (n = 1; n <= N; n++) {
    t = n * h;
    dS = f(phi(t)) * dphi(t) + f(phi(-t)) * dphi(- t);
    S += dS;
    if (fabs(dS) < 1.0e-16) {
      if (debug)
        printf("\tde(): n=%d, |t|=%g, fabs(dS)=%g\n", n, t, fabs(dS));
      break;
    }
  }
  return h * S;
}

/* テスト用の被積分関数 その1 */
double f1(double x)
{
  return sqrt(1 - x * x);
}

/* テスト用の被積分関数 その2 */
double f2(double x)
{
  if (x >= 1.0 || x <= -1.0) return 0; else return 1 / sqrt(1 - x * x);
}

void test(rrfunction f, double exact)
{
  int m, N;
  double h, IhN;

  /* |t|≦10 まで計算することにする */
  h = 1.0; N = 10;
  /* h を半分, N を倍にして double exponential formula で計算してゆく */
  for (m = 1; m <= 10; m++) {
```

```
    IhN = de(f, h, N);
    printf("h=%f, I_h=%25.16f, I_h-I=%e\n", h, IhN, IhN - exact);
    h /= 2; N *= 2;
  }
}

int main(int argc, char **argv)
{
  if (argc >= 2 && strcmp(argv[1], "-d") == 0)
      debug = 1;
  pi = 4.0 * atan(1.0); halfpi = pi / 2;

  printf("test1 (sqrt(1-x^2) の積分)\n");
  test(f1, halfpi);

  printf("test2 (1/sqrt(1-x^2) の積分)\n");
  test(f2, pi);

  return 0;
}
```

第3章　山本第7章「数値積分」から抜き書き

3.1　数値積分公式

積分

$$I(f) = \int_a^b f(x)dx$$

を求めるための近似公式[1] は、通常次の形である。

$$I_n(f) = \alpha_1 f(x_1) + \cdots + \alpha_n f(x_n) \quad (n \text{ 点公式}). \tag{3.1}$$

ここで α_j は定数, x_j は区間 $[a,b]$ における相異なる分点である。この公式による誤差を $E_n(f) \overset{\text{def.}}{=} I(f) - I_n(f)$ で表わし、条件

$$E_n(x^k) = 0 \quad (k = 0, 1, 2, \cdots, m)$$

が成り立つとき、公式 (3.1) は少なくとも m 次の精度を持つという。また

$$E_n(x^k) = 0 \quad (k = 0, 1, 2, \cdots, m), \quad E_n(x^{m+1}) \neq 0$$

が成り立つとき、公式 (3.1) は (ちょうど) m 次の精度を持つ、(ちょうど) m 次の積分公式であるという。

明らかに、ちょうど m 次の積分公式を用いるとき、高々 m 次の多項式 $f(x)$ について $E_n(f) = 0$, また $m+1$ 次の任意の多項式 $g(x)$ に対して、$E_n(g) \neq 0$ である。

n 点近似公式の作り方

$[a,b]$ 上に n 個の分点 $a \leq x_1 < x_2 < \cdots < x_n \leq b$ を取り、それらに関する $n-1$ 次補間多項式を $p_{n-1}(x)$ とする。Lagrange の公式を用いて表わせば、

$$p_{n-1}(x) = \sum_{j=1}^n \ell_j^*(x) f(x_j), \quad \ell_j^*(x) = \prod_{i=1, i\neq j}^n \left(\frac{x - x_i}{x_j - x_i}\right).$$

多項式であることから、$I(p_{n-1})$ は簡単に表現できる。

$$I(p_{n-1}) = \int_a^b p_{n-1}(x)\,dx = \sum_{j=1}^n \alpha_j f(x_j), \quad \alpha_j = \int_a^b \ell_j^*(x)\,dx.$$

そこで、$I(f)$ の近似として $I(p_{n-1})$ を採用してみると、これは確かに (3.1) の形をしていて、

$$\begin{aligned} F_n(f) &= I(f) - I_n(f) = \int_a^b \left(f(x) - p_{n-1}(x)\right)\,dx \\ &= \int_a^b (x - x_1) \cdots (x - x_n) f[x_1, \cdots, x_n, x] dx. \end{aligned}$$

[1]ある程度一般の f が与えられるものであると仮定している。

さらに f が $[a,b]$ において C^n-級ならば、

$$\exists\xi = \xi(x) \in (a,b) \quad \text{s.t.} \quad E_n(f) = \frac{1}{n!}\int_a^b (x-x_1)\cdots(x-x_n) f^{(n)}(\xi(x))\,dx.$$

特に $f(x) = x^k$ $(0 \le k \le n-1)$ ならば $f^{(n)}(x) \equiv 0$ であるから、$E_n(f) = 0$. つまり、これは少なくとも $n-1$ 次の公式である。

第4章　TO DO LIST

- 直交多項式の理論
- Gauss 型公式
- 高橋・森の誤差解析理論
- 杉原の DE 公式の最適性定理
- 多次元数値積分

森 [5]
Davis-Rabinowitz [6]
森・名取・鳥居 [7]
Davis-Rabinowitz [6]
山本 [3]
荒川・伊吹山・金子 [4]

関連図書

[1] 杉原正顯(まさあき), 室田(むろた)一雄：数値計算法の数理, 岩波書店 (1994).

[2] 森正武(まさたけ)：数値解析, 共立出版 (1973), 第 2 版が 2002 年に出版された。

[3] 山本哲朗(てつろう)：数値解析入門 [新訂版], サイエンス社 (2003), 1976 年初版発行の定番本の待望の改訂版.

[4] 荒川恒男, 伊吹山知義, 金子昌信：ベルヌーイ数とゼータ関数, 牧野書店 (2001).

[5] 森正武：FORTRAN 77 数値計算プログラミング, 岩波書店 (1990).

[6] Davis, P. J. and Rabinowitz, P.: *Methods of numerical integration*, Academic Press (1975, 1984), 初版の邦訳: 森正武訳, 計算機による数値積分法, 日本コンピュータ協会, 1980.

[7] 森 正武・名取 亮・鳥居達三：数値計算, 岩波講座・情報科学, 岩波書店 (1982), 4 章が森正武「数値積分」.