Ooura intde パッケージ について

桂田 祐史

2025 年 5 月 13 日, 2025 年 5 月 17 日

1 これは何か？

二重指数関数型数値積分公式 (以下 DE 公式と略記) の研究と、数値計算ライブラリィ (intde 以外に FFT のライブラリィを配布している) で有名な大浦拓哉氏による DE 公式のパッケージである。

次の WWW サイトで配布されている。

> 「二重指数関数型数値積分公式」
> https://www.kurims.kyoto-u.ac.jp/~ooura/intde-j.html
> (あ、エンコーディングが日本語 EUC で、ブラウザによっては誤判定するね。charset くらい書いておけばいいのに。)

Fortran と C 言語で同じ仕様の 3 つのサブルーチン/関数 intde(), intdei(), intdeo() が提供されている。

また簡単に使える Easy Version と、最初に表を初期化してそれを使って計算する Fast Version が用意されている。

パッケージに含まれるドキュメンテーション (`intde1.doc`, `intde2.doc`) 以外に、次の FAQ があって (その URL はパッケージに入れておとけばいいと思う)、それを読むと色々な疑問が氷解する。

> 「DE 積分パッケージ FAQ」
> https://www.kurims.kyoto-u.ac.jp/~ooura/intdefaq-j.html
> (あ、エン…)

その FAQ の参考文献表には載っているけれど、Ooura-Mori [1] という論文の内容を知らないと、`intdeo()` が理解できないのでは？と思う。

そういえば、このパッケージの公開の後に出版された Ooura-Mori [2], Ooura [3] と言う論文もある。和文の大浦 [4], [5] もチェックすべきか (特に [5] の内容は英文の報告が見当たらない)。講演のスライド大浦 [6] も分かりやすい説明なので、ぜひ読むことを勧めたい。

大浦先生は、このパッケージを公開した後に、たくさん論文を書いているわけで、それらの成果を取り入れたパッケージって作れないものなのだろうか…(勝手な期待)

2 試してみる

2.1 入手

```
curl -O https://www.kurims.kyoto-u.ac.jp/~ooura/intde.tar.gz
```

2.2 展開

```
mkdir intde
cd intde
tar xzf ../intde.tar.gz
```

2.3 C言語 Easy Version を試す

intde1.c は “Quadrature Package in C - Easy Version” とのこと。そして intde1t.c がテスト・プログラムである。

intde1t.c は intde1.c を #include しているので、単体でコンパイル&リンクできる。main() の型が書いてない。最近のコンパイラー (うるさい) でコンパイルするには、int main() と書き換えれば良い。

3つの関数が含まれている。関数の仕様が知りたければ、ソース・プログラム intde1.c の注釈を読め、だそうだ。

(i) intde() — 有限区間 (a, b) 上の積分 $\int_a^b f(x)\,dx$, f は (a, b) 上の解析関数

```
intde(double (*f)(double), double a, double b, double eps, double *i, double
*err);
```

f, a, b は被積分関数と積分区間

eps は許容される相対誤差 (絶対誤差を $\int_a^b |f(x)|dx$ で割ったもの)

*i は積分の評価値 (結果)

*err は絶対誤差を評価したもの (正常に終了した場合は)。異常が起こった場合は負の値を返す。

(ii) intdei() — 半無限区間 (a, ∞) 上の積分 $\int_a^\infty f(x)\,dx$, ただし f は “振動しない”。

```
intdei(double (*f)(double), double a, double eps, double *i, double *err);
```

各引数の意味は intde() のそれと同じ (当然 b がない)

(iii) intdeo() — 半無限区間 (a, ∞) 上の積分 $\int_a^\infty f(x)\,dx$, f は振動する。

```
intdeo(double (*f)(double), double a, double omega, double eps, double *i,
double *err);
```

b の代わりに入っている omega は何か？ $f(x) = f_1(x)\cos\omega x$ や $f(x) = f_1(x)\sin\omega x$ (Ooura-Mori [1] には、“$f_1(x)$ is an algebraic function” と書いてあったりするけれど、そういうことかな？？) という形の被積分関数を想定していて、その ω ということで良いのか？三角多項式とかでも大丈夫なのかな？？

コンパイル&実行

```
cc -O -o intde1t intde1t.c
./intde1t
```

実行結果を見る

```
% ./intde1t
I_1=int_0^1 1/sqrt(x) dx
 I_1= 2 , err= 4e-15 , N= 71
I_2=int_0^2 sqrt(4-x*x) dx
 I_2= 3.14159 , err= 6.28319e-15 , N= 57
I_3=int_0^infty 1/(1+x*x) dx
 I_3= 1.5708 , err= 3.14159e-15 , N= 55
I_4=int_0^infty exp(-x)/sqrt(x) dx
 I_4= 1.77245 , err= 7.08986e-15 , N= 127
I_5=int_0^infty sin(x)/x dx
 I_5= 1.5708 , err= 6.6389e-15 , N= 119
I_6=int_0^infty cos(x)/sqrt(x) dx
 I_6= 1.25331 , err= 2.87472e-14 , N= 128
```

(誤差小さいのだから、もっとたくさんの桁数表示すればいいのに。)

2.4 C 言語 Fast Version を試す

Easy Version は “表を使わない”、言い換えると中で表を生成している。初期化ルーチンで表 (引数の名前は、表が aw, その長さ (length of aw) が lenaw) を作っておいて、以下その表を使って積分することで、全体の効率をあげる。そうしたのが Fast Version ということみたいだ。

関数の名前は intde(), intdei(), intdeo() となっていて、Easy Version と同じなんだ (混用は考えていないわけだな)。それぞれ初期化関数 intdeini(), intdeiini(), intdeoini() がついている。

使い方については、Easy Version と Fast Version を並べて見せると分かりやすいかな。

Easy version での intde() の使い方

```
intde(f, a, b, 1.0e-15, &i, &err);
```

Fast version での intde() の使い方

```
double tiny; // minimum value that 1/tiny does not overflow (double)
double aw[8000];
...
lenaw = 8000;
tiny = 1.0e-307; // IEEE P754 の倍精度を想定しているのだろう
intdeini(lenaw, tiny, 1.0e-15, aw); // 1.0e-15 は許容する相対誤差
intde(f, a, b, aw, &i, &err);
```

表を作るのに被積分関数 f は要らない。許容する相対誤差は、表を作るときに与えるが、intde() には与えない。その代わりに表 aw を与える。ふむ。

intdei() の使い方

```
// 変数 aw[], lenaw, tiny の準備は intde() と同じ
intdeiini(lenaw, tiny, 1.0e-15, aw);
intdei(f, a, aw, &i, &err);
```

intde() との違いは、intdei() の方に b がないだけか。初期化する関数 intdeiini() の引数の並びは、intdeini() のそれと同じだけど、生成される表 aw[] の内容は違うのだろうね。

intdeo() の使い方

```
// 変数 aw[], lenaw, tiny の準備は intde() と同じ
double omega = 何か;
intdeoini(lenaw, tiny, 1.0e-15, aw);
intdeo(f, a, omega, aw, &i, &err);
```

intde() との違いは、intdeo() の方に、b の代わりに omega を指定すること。初期化する関数 intdeoini() の引数の並びは、intdeini() のそれと同じ。

コンパイル&実行

```
cc -O -o intde1t intde2t.c
./intde2t
```

実行結果を見る

```
% ./intde2t
I_1=int_0^1 1/sqrt(x) dx
 I_1= 2 , err= 3.9999e-15 , N= 64
I_2=int_0^2 sqrt(4-x*x) dx
 I_2= 3.14159 , err= 6.28319e-15 , N= 54
I_3=int_0^infty 1/(1+x*x) dx
 I_3= 1.5708 , err= 3.14159e-15 , N= 55
I_4=int_0^infty exp(-x)/sqrt(x) dx
 I_4= 1.77245 , err= 7.08978e-15 , N= 108
I_5=int_0^infty sin(x)/x dx
 I_5= 1.5708 , err= 6.6389e-15 , N= 119
I_6=int_0^infty cos(x)/sqrt(x) dx
 I_6= 1.25331 , err= 2.87472e-14 , N= 128
```

A　端点に特異性がある場合を試す

ソース・プログラムを見ると、tanh は見当たらず、exp で済ませている。$\tanh x \pm 1$ の桁落ち対策などは当然やっているようだ。試しに

$$\int_0^1 \frac{\sqrt{x}}{e^x - 1} dx$$

を積分してみると ($x = 0$ の近傍で、被積分関数 $\sim \dfrac{1}{\sqrt{x}}$ となっている。素朴なコードでは分母が桁落ちする。)

mytest1.c

```
// mytest1.c
```

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

#include "intde1.c"

double f(double x)
{
  if (x <= 0.0)
    return 0.0;
  else
    return sqrt(x) / expm1(x);
}

int main()
{
    double i, err, iexact = 1.69969635021554408316298900841791864 97;

    intde(f, 0.0, 1.0, 1.0e-15, &i, &err);
    printf("I_1=int_0^1 sqrt(x)/expm1(x) dx\n");
    printf(" I_1= %lg\t, err= %g, diff=%g\n", i, err, i - iexact);
    return 0;
}
```

プログラム中の `iexact = 1.69969···` は、Mathematica でカンニングした積分の値である。その値と `intde()` の結果との差を表示するようになっている。その結果は次のようになる。

コンパイル&実行結果

```
% cc -o mytest1 mytest1.c
% ./mytest1
I_1=int_0^1 sqrt(x)/expm1(x) dx
 I_1= 1.6997 , err= 3.39939e-15, diff=-4.44089e-16
```

満足の行く結果が得られている (誤差の見積もりは 4.0×10^{-15} で、実際の差は 4.4×10^{-16})。端点に特異性があっても、被積分関数の値が精度良く計算できるならば大丈夫のようだ。

それでは

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

はどうだろう？これは $x = 1$ に特異性があり、素朴なコードでは被積分関数を精度良く計算できない。

試しに

```
double f1(double x)
{
  if (x >= 1.0)
    return 0.0;
  else
    return 1.0 / sqrt(1.0 - x * x);
}
```

とすると

```
% ./mytest2
I_1=int_0^1 dx/sqrt(1-x^2)
 I_1=    1.570796315173100 , err= 1.25664e-14, diff=-1.16218e-08
```

誤差の見積もりは 1.3×10^{-14} だけど、真の値との差は実際には 1.2×10^{-8} だ。これはあまり満足できない。特異性の影響が出ている。

どうすれば良いか？特異性のある点が原点に来るように平行移動すれば良いのだろう。$x = y + 1$ と置換して

$$\int_0^1 \frac{dx}{\sqrt{1-x^2}} = \int_{-1}^0 \frac{dy}{\sqrt{1-(y+1)^2}} = \int_{-1}^0 \frac{dy}{\sqrt{-y(2+y)}}$$

と変形すると、特異性があるのは $y = 0$ の部分でこれなら $\dfrac{1}{\sqrt{-y(2+y)}}$ を素朴に計算して精度が保てる。実際

```
double f2(double y)
{
  if (y >= 0.0)
    return 0.0;
  else
    return 1.0 / sqrt(-y * (2.0 + y));
}
...
intde(f2, -1.0, 0.0, 1.0e-15, &i, &err);
```

としたところ

```
I_1=int_0^1 dx/sqrt(1-x^2)
 I_1=    1.570796326794896 , err= 3.14159e-15, diff=-2.22045e-16
```

これはうまく計算できた。

次に

$$\int_{-2}^2 \frac{dx}{\sqrt{4-x^2}}$$

のように区間の両端に特異性がある場合は、積分区間を 2 つに分けて、それから $x = -2, 2$ がそれぞれ 0 に来るように変数変換して

$$\begin{aligned}\int_{-2}^2 \frac{dx}{\sqrt{4-x^2}} &= \int_{-2}^0 \frac{dx}{\sqrt{4-x^2}} + \int_0^2 \frac{dx}{\sqrt{4-x^2}} \\ &= \int_0^2 \frac{dy}{\sqrt{4-(y-2)^2}} + \int_{-2}^0 \frac{dy}{\sqrt{4-(y+2)^2}} \\ &= \int_0^2 \frac{dy}{\sqrt{y(4-y)}} + \int_{-2}^0 \frac{dy}{\sqrt{-y(y+4)}}.\end{aligned}$$

この 2 つの積分を別々に計算して加えれば良いだろう。

mytest3.c

```
// mytest1.c

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

#include "intde1.c"

double f2(double x)
{
    if (fabs(x) >= 2.0)
      return 0.0;
    else
      return 1.0 / sqrt(4.0 - x * x);
```

```
}
double f2L(double y)
{
    return 1.0 / sqrt(y * (4.0 - y));
}

double f2R(double y)
{
    return 1.0 / sqrt(- y * (4.0 + y));
}

int main()
{
    double i, iL, iR, err, errL, errR, iexact = 4.0 * atan(1.0);

    intde(f2, -2.0, 2.0, 1.0e-15, &i, &err);
    printf("I_1=int_0^1 dx/sqrt(1-x^2)\n");
    printf(" I_1= %20.15f, err= %g, diff=%g\n", i, err, i-iexact);

    intde(f2L,  0.0, 2.0, 1.0e-15, &iL, &errL);
    intde(f2R, -2.0, 0.0, 1.0e-15, &iR, &errR);
    i = iL + iR;
    printf(" I_1= %20.15f, err= %g,%g diff=%g\n", i, errL, errR, i-iexact);
    return 0;
}
```

コンパイルと実行結果

```
% cc -o mytest3 mytest3.c
% ./mytest3
I_1=int_0^1 dx/sqrt(1-x^2)
 I_1=    3.141592634878369, err= 6.28319e-15, diff=-1.87114e-08
 I_1=    3.141592653589793, err= 3.14159e-15,3.14159e-15 diff=-4.44089e-16
```

対策をした方では、真の値との差の絶対値が 4.4×10^{-16}. 誤差の見積もりも小さい。満足の行く結果が得られた。

B　個人的な改造 — 複素数値関数の数値積分

私は、複素数を用いる数値計算をすることが多い。複素数値関数 (変数は実数) の数値積分が必要になったので、intde を使えないか考えてみた。

`intde()`, `indtdei()`, `intdeo()` のうち前の 2 つは、簡単な修正で複素数値関数対応にできるようだ。`intdeo()` はどうするものか…

B.1　C でやる場合

- `#include <complex.h>` をする。
- 関数の戻り値やそれを使った演算を結果を記憶する変数・引数 (`i`, `ir`, `iback`, `irback`, `fa`, `fb`, `fp`, `fm` など) は、型の名前を `double` から `double complex` にする。
- 被積分関数の値から求まる量についての `fabs()` は、`cabs()` に置き換える。
- `printf()` では `double complex` の式の値を直接表示してくれないので、実部・虚部をそれぞれ `creal()`, `cimag()` で取り出して表示する。例えば

```
printf("z=%f+%f i\n", creal(z), cimag(z));
```

のようにする。

B.2 C++ でやる場合

- `#include <complex>` をする。
- 関数の戻り値やそれを使った演算を結果を記憶する変数・引数 (`i`, `ir`, `iback`, `irback`, `fa`, `fb`, `fp`, `fm` など) は、型の名前を `double` から `complex<double>` にする。
- `fabs()` は `abs()` に置き換える。
- C++ では、int 型のデータと `complex<double>` 型のデータは直接演算できない。`2` をかけるところは `2.0` をかけるように直す。具体的には

  ```
  errd = h * (abs(*i - 2 * iback) + 4 * abs(ir - 2 * irback));
  ```

  を次のようにする。

  ```
  errd = h * (abs(*i - 2.0 * iback) + 4 * abs(ir - 2.0 * irback));
  ```

参考文献

[1] Ooura, T. and Mori, M.: The double exponential formula for oscillatory functions over the half infinite interval, *Journal of Computational and Applied Mathematics*, Vol. 38, pp. 353–360 (1991).

[2] Ooura, T. and Mori, M.: A robust double exponential formula for Fourier-type integrals, *Journal of Computational and Applied Mathematics*, Vol. 112, pp. 229–241 (1999).

[3] Ooura, T.: A Double Exponential Formula for the Fourier Transforms, *Publication of the Research Institute for Mathematical Sciences, Kyoto University*, Vol. 41, No. 4, pp. 971–977 (2005).

[4] 大浦拓哉：二重指数関数型数値積分公式の収束判定法の改良, 日本応用数理学会論文誌, Vol. 13, No. 2, pp. 225–230 (2003).

[5] 大浦拓哉：二重指数関数型変換を用いた様々な積分変換の計算法, 日本応用数理学会論文誌, Vol. 19, No. 1, pp. 73–79 (2009).

[6] 大浦拓哉：二重指数関数型積分公式について, 第 46 回数値解析シンポジウム, グリーンパーク想い出の森 滋賀県高島市, (森正武先生を偲ぶ講演), 2017/6/29, https://www.kurims.kyoto-u.ac.jp/~ooura/papers/mori_ohp.pdf (2017/6/29).