# 問 1

### 問題

次の整数型 (char 型,short 型,int 型,long 型,long long 型) で記憶可能である最大値と最小値について調べなさい. また, 各整数型の unsigned での最小値と最大値の範囲についても答えよ. 各整数型変数の最大値に 1 を加えると何が起こるか. また, 最小値から 1 を引くと何が起こるか.

### 方法

まず,sizeof 関数により, 各変数型のメモリサイズを調べる.

続いて, 符号付き変数型の場合, 最上位 bit が 0, それ以外の bit が 1 である値を代入し, その値をインクリメントし, 最大値かどうか調べる. また, 最上位 bit が 1, それ以外の bit が 0 である値を代入し, その値をデクリメントし, 最小値かどうか調べる. 符号無し変数型の場合, すべての bit が 1 である値を代入し, その値をインクリメントし, 最大値かどうか調べる. また, すべての bit が 0 である値を代入し, その値をデクリメントし, 最小値かどうか調べる.

### 結果

各変数型のメモリサイズは表 1 である. なお,unsigned の場合もメモリサイズは同じである.

表 1　各変数型のメモリサイズ

| 変数型 | メモリサイズ (byte) |
|---|---|
| char | 1 |
| short | 2 |
| int | 4 |
| long | 8 |
| long long | 8 |

符号付き変数型のときの結果は表 2, 表 3 となる.

表 2　最大値だと推測した値とインクリメント後の値 (符号付き)

| 変数型 | 初めに代入した値 | インクリメントした後の値 |
|---|---|---|
| char | 127 | -128 |
| short | 32767 | -32678 |
| int | 2147483647 | -2147483648 |
| long | 9223372036854775807 | -9223372036854775808 |
| long long | 9223372036854775807 | -9223372036854775808 |

表 3　最小値だと推測した値とデクリメント後の値 (符号付き)

| 変数型 | 初めに代入した値 | デクリメントした後の値 |
|---|---|---|
| char | -128 | 127 |
| short | -32768 | 32677 |
| int | -2147483648 | 2147483647 |
| long | -9223372036854775808 | 9223372036854775807 |
| long long | -9223372036854775808 | 9223372036854775807 |

また, 符号無し変数型のときの結果は表 4, 表 5 となる.

表 4　最大値だと推測した値とインクリメント後の値 (符号無し)

| 変数型 | 初めに代入した値 | インクリメントした後の値 |
|---|---|---|
| unsigned char | 255 | 0 |
| unsigned short | 65535 | 0 |
| unsigned int | 4294967295 | 0 |
| unsigned long | 18446744073709551615 | 0 |
| unsigned long long | 18446744073709551615 | 0 |

表 5　最小値だと推測した値とデクリメント後の値 (符号無し)

| 変数型 | 初めに代入した値 | デクリメントした後の値 |
|---|---|---|
| unsigned char | 0 | 255 |
| unsigned short | 0 | 65535 |
| unsigned int | 0 | 4294967295 |
| unsigned long | 0 | 18446744073709551615 |
| unsigned long long | 0 | 18446744073709551615 |

**考察**

まず,2 進数表現について解説する.[1] 2 進数表現には符号無しと符号付きの 2 つがある. 符号無し表現ではそのまま 2 進数表現を 10 進数に変換すればよい. 例えば 2 進数の 0110 は $0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$ と計算し,10 進数では 6 となる. 一方, 符号付き表現の場合は最上位 bit は符号を表現しており. 最上位 bit が 0 のときその 2 進数は正,1 のときは負となる. 最上位 bit 以外の bit で数の大きさを表現する. 10 進数に変換する場合はまず最上位 bit から正負を決定し, 正の場合は残りの bit を 10 進数変換すればよいが, 負の場合は補数を取る操作の逆の操作をしてから 10 進数変換しなければならない.(補数を取るとは bit を反転させて 1 を加えることであり, 逆の操作では 1 を引いてから bit を反転させることになる) 例えば,2 進数で 0110 は, まず最上位 bit が 0 であることから正とわかり,110 を 10 進数に変換すると 6 であるので,0110 は +6 となる. また,2 進数で 1110 は, まず最上位 bit が 1 であ

ることから負とわかり, 補数を取る操作の逆の操作をすると 0010 となり, これは 10 進数で 2 であるので 1110 は-2 となる.

以上を踏まえて, 本題に入る. まず sizeof 関数により各整数型の bit 数は,1byte = 8bit の関係から,char 型は 8bit,short 型は 16bit,int 型は 32bit,long 型と long long 型は 64bit であることがわかる. 前述した 2 進数表現方法により, 符号付き整数型では, 最大値は最上位 bit が 0, それ以外の bit が 1 となっているときであると推測できる. つまり, 最大値は,char 型は 127,short 型は 32767,int 型は 2147483647,long 型と long long 型は 9223372036854775807 であると推測できる. また, 最小値は最上位 bit が 1, それ以外の bit が 0 となっているときであると推測できる. つまり, 最小値は,char 型-128,short 型は-32768,int 型は-2147483648,long 型と long long 型は-9223372036854775808 であると推測できる. 続いて, 符号無し整数型では. 最大値はすべての bit が 1 になっているときであると推測できる. つまり, 最大値は,char 型は 255,short 型は 65535,int 型は 4294967295,long 型と long long 型は 18446744073709551615 であると推測できる. また, 最小値はすべての bit が 0 となっているときであると推測できる. つまり, 最小値は, どの整数型も 0 であると推測できる.

結果より, 符号付き整数型の最大値, 最小値は表 6, 符号無し整数型の最大値, 最小値は表 7 となる.

表 6　各整数型の最大値, 最小値 (符号付き)

| 変数型 | 最小値 | 最大値 |
|---|---|---|
| char | -128 | 127 |
| short | -32768 | 32677 |
| int | -2147483648 | 2147483647 |
| long | -9223372036854775808 | 9223372036854775807 |
| long long | -9223372036854775808 | 9223372036854775807 |

表 7　各整数型の最大値, 最小値 (符号無し)

| 変数型 | 初めに代入した値 | デクリメントした後の値 |
|---|---|---|
| unsigned char | 0 | 255 |
| unsigned short | 0 | 65535 |
| unsigned int | 0 | 4294967295 |
| unsigned long | 0 | 18446744073709551615 |
| unsigned long long | 0 | 18446744073709551615 |

また, 各整数型変数の最大値に 1 を加えると最小値になり, 逆に各整数型変数の最小値から 1 を引くと最大値になる. このようになるのは, 前述したように, 符号付きでは最大値は最上位 bit が 0, それ以外の bit が 1 で, 最小値は最上位 bit が 1, それ以外の bit が 0 であることと, 符号無しでは最大値はすべての bit が 1 で, 最小値はすべての bit が 0 であることが理由である.

# 問 2

### 問題

コンピュータの実数計算で使用される浮動小数点演算の仕組みを調べ, 具体例を用いて, 丸め誤差, 情報落ち誤差, 桁落ち誤差が発生する仕組みを論じよ.

### 方法

(1) 0.1 + 0.2 == 0.3 の値を出力するプログラムを用いて, 丸め誤差について調べる.

(2)1.0 に $10^{-16}$を $10^8$回足すプログラムを用いて, 情報落ち誤差について調べる.また, 足す順を変えたときの違いについて調べる.

(3)2 次方程式 $ax^2 + bx + c$ の解$\alpha, \beta$を公式で計算するプログラムを用いて, 桁落ち誤差について調べる.なお, 2 次方程式の 2 つの解$\alpha, \beta$は解の公式より,

$$\alpha = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \beta = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{1}$$

である. また, $\alpha$の分子を有理化し,

$$\alpha = \frac{-2c}{b + \sqrt{b^2 - 4ac}}, \beta = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{2}$$

とも書ける.

### 結果

(1) prob2_ marume.c を実行すると, 出力された値は 0 であった.

(2) prob2_ jouhou.c を実行すると, 結果は表 8 となる.

表 8　$1 + 10^{-16} \times 10^8$の計算結果

| 真の値 | 大きい順に足したとき | 小さい順に足したとき |
|---|---|---|
| 1.00000001 | 1.00000000000000000000 | 1.00000000999999993923 |

(3) $a = 1, b = 1000.001, c = 1$ の 2 次方程式について,prob2_ ketaoti.c を実行すると, 結果は表 9 となる.

表 9　$x^2 + 1000.001x + 1$ の真の解とプログラムにより計算した解

| | 真の解 | $\alpha = \frac{-b+\sqrt{b^2-4ac}}{2a}, \beta = \frac{-b-\sqrt{b^2-4ac}}{2a}$ | $\alpha = \frac{-2c}{b+\sqrt{b^2-4ac}}, \beta = \frac{-b-\sqrt{b^2-4ac}}{2a}$ |
|---|---|---|---|
| $\alpha$ | -0.001 | -0.00100708 | -0.00100000 |
| $\beta$ | -1000 | -1000 | -1000 |

**考察**

浮動小数点について説明する.[2] 浮動小数点とは, 小数点の位置を変えて表現された数である. 例えば $-16.25$ に対して $-1.625 \times 10^1$が浮動小数点である. この例において, $-$は符号部, 1.025 は仮数部, 10 の肩にある 1 は指数部と呼ばれている. しかし, このままでは 2 進数で数を表現するコンピュータでは取り扱えない.

そこで,IEEE754 という標準規格が採用されている. これは浮動小数点を 32bit(倍精度の場合 64bit) で表す方法であり, 最上位 bit から順番に符号部が 1bit, 指数部が 8bit(倍精度の場合 11bit), 仮数部が 23bit(倍精度の場合 52bit) を占める. 符号部が S, 指数部が E, 仮数部が M のとき, この浮動小数点で表現している数字は $(-1)^S \times 1.M \times 2^{E-127}$ である. -16.25 の 2 進数の浮動小数点表示を通じて,IEEE754 の表現方法について具体的に解説する. まず, 符号は問 1 で前述したことと同様に正なら 0, 負なら 1 であり, 今回は負なので符号の bit は 1 である. 続いて 16.25 を 2 進数表示すると 10000.01 であり, 整数部分が 1 になるように小数点を動かし,$1.000001 \times 2^4$となる.つまり.指数は 4, 仮数は 1.000001 となる. しかし, 実際に指数部や符号部には別の値が入る. まず, 指数部にはバイアス値として 127 を足した 131, つまり 2 進数で 1000 0011 が入る. 仮数部は整数部分を除いた 0000 0010 0000 000 が入る. 以上をまとめて,-16.25 の浮動小数点表示は 1 1000 0011 0000 0010 0000 000 となる.IEEE754 ではこのような過程により 2 進数の浮動小数点を求める.

コンピュータ上ではこのような浮動小数点を用いた浮動小数点演算を行っている. このため, 様々な誤差が生じる. 誤差には丸め誤差, 情報落ち誤差, 桁落ち誤差がある.

まず, 丸め誤差について考察する. 具体例として用いたプログラムでは $0.1 + 0.2 == 0.3$ の値は実数で考えれば 1, つまり True になるはずだが, プログラムを実行すると 0, つまり False であった. この理由は, 浮動小数点の表現方法からわかるようにすべて 2 進数の有限小数で表現するため,2 進数の無限小数はある桁で打ち切られてしまうからである. これにより, 打ち切られた分だけ誤差が生じてしまう. この誤差を丸め誤差という. 丸め誤差は元の数に比べてかなり小さいものであるが, プログラムに様々な影響を及ぼす. 具体例では 0.1 や 0.2, また,0.3 が 2 進数では無限小数となり, 浮動小数点表示にする際に丸め誤差が生じてしまったため,False が返されてしまった.

続いて. 情報落ち誤差について考察する. 具体例として用いたプログラムでは, 真の値は 1.00000001 であるが, 実際にこのプログラムを実行すると, 大きい順に足した場合, 結果は 1.00000000000000000000 となる. この理由は, 浮動小数点での加減算は指数部を揃えてから計算を行うが, 仮数のビット数が有限であることから, 絶対値の小さい数の仮数部が 0 になってしまうからである. このように絶対値の大きい数と絶対値の小さい数の加減算を行った際に, 絶対値の小さい数が無視されてしまうことで生じる誤差を情報落ち誤差という. 1 回の演算で生じる誤差は非常に小さいが, 多くの演算を行うにつれて, 誤差が無視できないほど大きくなってしまう. 情報落ち誤差は小さい数から計算をしていくことである程度回避することができる. 具体例において, 初めに $10^{-16}$ どうしで加算を行ってから最後に 1.0 を加えると,1.0000000999999993923 となり, 先ほどよりも誤差が小さくなっている.

続いて, 桁落ち誤差について考察する. 具体例として用いたプログラムでは, $\alpha = -0.00100708, \beta = -1000.000$ となり, $\alpha$が真の解と異なることがわかる. これは, $b$ と$\sqrt{b^2 - 4ac}$が非常に近い値であり, これらの数の引き算において, 有効桁数が落ちてしまったからである. 10 進数の例を用いると, 有効桁数 7 桁で $1.414213 \times 10^{-2} - 1.414135 \times 10^{-2}$を計算すると, $0.000078 \times 10^{-2}$で小数点を移動して $7.800000 \times 10^{-7}$となるが, 追加した 0 は形式的なものであり, 計算によって現れたものではない. つ

まり, この 0 は正確ではないため, 実質の有効桁数は 2 桁であり, 有効桁数が減少してしまっている. これと同じことが 2 進数の浮動小数点表示でも起こってしまい, 仮数部に不足した桁数分だけ 0 が追加され, 有効桁数が減少する.有効桁数が小さい数字を用いて計算を行うと, 計算結果も同様の有効桁数になってしまう. つまり, 計算結果の精度が非常に悪くなってしまう. このような誤差を桁落ち誤差という. 具体例では, $b \approx \sqrt{b^2 - 4ac}$となり, $\alpha$の計算で桁落ち誤差が起こり, $\alpha$の値が真の解と異なってしまっている. 桁落ち誤差は非常に近い値どうしの減算を避けることで回避することができる. 具体例では, $\alpha = (-b + \sqrt{b^2 - 4ac})/2a$ ではなく, $\alpha = -2c/(b + \sqrt{b^2 - 4ac})$ で計算すると, $\alpha = -0.001000000$ となり, 桁落ち誤差が生じず, 精度がよくなった. しかし, 桁落ち誤差は原理的に回避できない場合もある.

## 問 3

### 問題

無限和$\zeta(8) = S = \sum_{k=1}^{\infty} \frac{1}{k^8}$の近似値を足し上げる $k$ の値の上限値をある大きな整数 $K$ として $S_K = \sum_{k=1}^{K} \frac{1}{k^8}$として求めよ. $K$ を $100, 1000, 10000, 100000, 1000000$ と変化させた場合について $k$ を大きい値から順に加算した場合と, 小さい値から順に加算した場合とで近似値 $S_K$にどのような違いが生じるか確認せよ. また,float型とdouble型とで例で作成したプログラムを用いるとどのような違いが生じるか.またどうしてこれらの違いが生じたのであろうか?

### 方法

$\zeta(8)$ をプログラムを用いて計算する. 加算順を変えたり, 変数型を変えたときに生じる変化を調べる.

### 結果

prob3.c を実行し, 計算した結果は表 10 となる. 小さい順に足した場合と大きい順に足した場合で値が異なる桁から下の部分は太字にしている. なお,$\zeta(8)$ の真の値は 1.00407735619794433 9379 である.

表 10　変数型・加算順に対する計算結果

| 変数型 | K | k が小さい順 | k が大きい順 |
|---|---|---|---|
| float | 100 | 1.00407719612121582031250**0000** | 1.00407731533050537109375**0000** |
| | 1000 | 1.00407719612121582031250**0000** | 1.00407731533050537109375**0000** |
| | 10000 | 1.00407719612121582031250**0000** | 1.00407731533050537109375**0000** |
| | 100000 | 1.00407719612121582031250**0000** | 1.00407731533050537109375**0000** |
| | 1000000 | 1.00407719612121582031250**0000** | 1.00407731533050537109375**0000** |
| double | 100 | 1.00407735619794302728280399 6059 | 1.00407735619794302728280399 6059 |
| | 1000 | 1.00407735619794**3027282803996059** | 1.00407735619794**4359550433546246** |
| | 10000 | 1.00407735619794**3027282803996059** | 1.00407735619794**4359550433546246** |
| | 100000 | 1.00407735619794**3027282803996059** | 1.00407735619794**4359550433546246** |
| | 1000000 | 1.00407735619794**3027282803996059** | 1.00407735619794**4359550433546246** |

### 考察

まず, 足した順を変えたときに生じる違いについて考察する. double 型で K=100 のときは足す順を変えても計算結果は変わらなかったが, それ以外の場合では計算結果が変わり, 大きい順に足したほうが真の値に近づいた. double 型は 64bit の浮動小数点で有効桁数が 10 進数で 15 桁であるので k が小さい順に足すと情報落ち誤差が k=100 から生じ始めたのに対して,float 型は 32bit の浮動小数点で有効桁数が 10 進数で 7 桁であるので k=100 になる前に情報落ち誤差が生じ始めている. よって,double 型で $K = 100$ の場合は $k$ が小さい順に足しても情報落ち誤差が生じなかったため, 足す順を変えても計算結果は変わらなかった. しかし, それ以外の場合では $k$ が小さい順に足すと情報落ち誤差が生じ, $k$ がある値を超えた後はそれ以降の和はすべて無視されてしまうが, $k$ が大きい順に足すことで先ほど無視された部分も計算結果に含まれる.よって, 足す順を変えると計算結果が変化し, 真の値へ近づいた.

続いて, 変数型を変えたときに生じる違いについて考察する. double 型は K が 100 のときの 1000 以上のときで値が異なるが,float 型ではどの場合も値は同じになった. これは, 前述したように情報落ち誤差が生じ始める k の値が double 型の場合のほうが大きいからである. また,float 型に比べて,double 型で計算した場合のほうが真の値に近い計算結果となった. これは,double 型のほうが有効桁数が多く, 情報落ち誤差が生じにくく, 精度の高い計算ができるからである.

## 問 4

### 問題

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{x}{1+e^t}, x(0) = 1 \tag{3}$$

を Euler 法, 二次の Runge-Kutta 法, 四次の Runge-Kutta 法により数値的に解くことを考える.$t = 100$ における解析解 $x_{theo}(100)$ と数値解 $x(100.0)$ との誤差$\varepsilon = |x_{theo}(100) - x(100)|$ を 3 つの方法を用いて$\Delta t$ の関数として計算せよ. $\Delta t$ を 0.5 から $0.1, 0.05, 0.01, 0.005, 0.001, \ldots$ と順に小さくしていった場合, 打切り誤差$\varepsilon$は$\Delta t$ のどのような関数として表示されるか? 実際に数値的に計算を行い, $\Delta t$ と$\varepsilon$の両対数グラフとして図示せよ.また, 得られた図の結果となる理由について考察せよ.

### 方法

各解法に基づいて.式 (3) を刻み幅$\Delta t$ を変化させながら数値積分で解き, 刻み幅$\Delta t$ と打ち切り誤差$\varepsilon$の関係を図示する. なお, 解析解は $x_{theo}(t) = \frac{2e^t}{e^t+1}$である.

### 結果

prob4.c を実行し, 数値計算を行う. 各解法において, 刻み幅$\Delta t$ と打ち切り誤差$\varepsilon$の関係は図 1 となる. ただし, 両対数軸である.

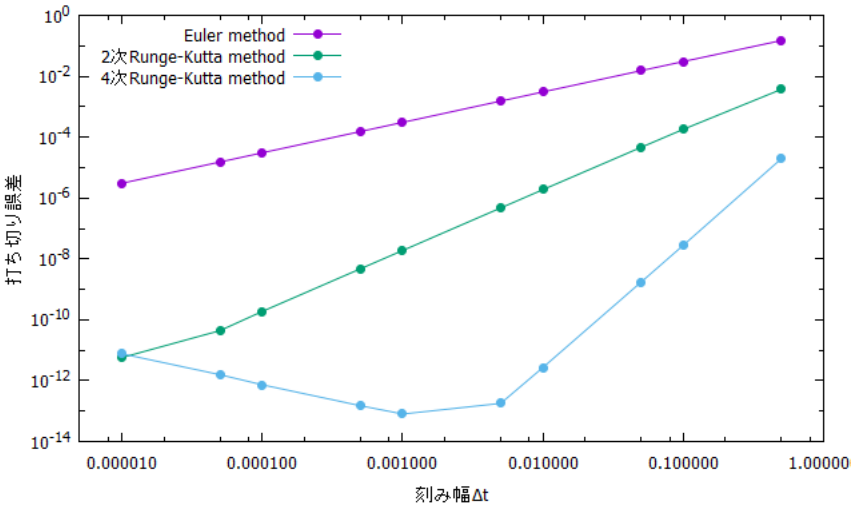


図 1　各解法における刻み幅と打ち切り誤差の関係

**考察**

図 1 より,Euler 法は傾き 1 の直線,2 次の Runge-Kutta 法は傾き 2 の直線,4 次の Runge-Kutta 法は $\Delta t$ が 0.01 より大きい領域では傾き 4 の直線になっている. このことから, 打ち切り誤差 $\varepsilon$ は Euler 法では $c_1\Delta t$,2 次の Runge-Kutta 法では $c_2(\Delta t)^2$,4 次の Runge-Kutta 法では $c_3(\Delta t)^4$という関数として表示される $(c_1, c_2, c_3$は定数). Euler 法,2 次の Runge-Kutta 法,4 次の Runge-Kutta 法の理論的な打ち切り誤差はそれぞれ $O(\Delta t), O((\Delta t)^2), O((\Delta t)^4)$ であるが, 今, 刻み幅は十分小さいので次数の大きい項が無視でき, 各解法の打ち切り誤差は$\Delta t, (\Delta t)^2, (\Delta t)^4$の比例式になる. よって, 両対数で図示すると図 1 のような直線となる. また,4 次の Runge-Kutta 法 では$\Delta t$ が 0.001 より小さい領域では刻み幅を小さくすると打ち切り誤差が増加している. これは刻み幅を小さくしたことで減少する離散化誤差より, 反復回数が増えたことで増加する丸め誤差や情報落ち誤差のほうが大きくなってしまうからである.

# 問 5

**問題**

- 台形則
- シンプソンの公式

を用いて定積分

$$\int_0^{\pi/2} \sin(2x)\mathrm{d}x \tag{4}$$

の値を求めよ. そして, 解析的に得られる定積分値と数値的に得られる値 $F$ との差 $E$

$$E = \left| \int_0^{\pi/2} \sin(2x)\mathrm{d}x - I_N \right| \tag{5}$$

を離散化刻み $h$ の関数として両対数軸で図示せよ. この図から, 両スキームの $h$ に対する誤差 $E$ の減少のオーダーを見積もれ. 更に, 和をとる順序を入れ替え, 小さい数から順に大きい数になるように和を計算せよ.

## 方法

台形則, シンプソン則を用いて, 離散化刻み $h$ を変化させながら数値積分により式 (4) の値を求め, 解析解との誤差を調べる. また, 台形則, シンプソン則の加算の順を変え, 値が小さい順に加算した場合の値についても調べる.

## 結果

prob5_Trape.c と prob5_Simpson.c を実行し, 数値積分を行う. 各数値積分法における離散化刻み $h$ と誤差 $E$ の関係は図 2 となる. 解析的に得られる定積分値は 2 である. なお,Trapezoidal rule は台形則,Improved Trapezoidal rule は値が小さい順に足す台形則,Simpson rule はシンプソン則,Improved Simpson rule は値が小さい順に足すシンプソン則を指している. Trapezoidal rule と Improved Trapezoidal rule のグラフが被っていることに注意されたい. また, グラフは両対数軸である.

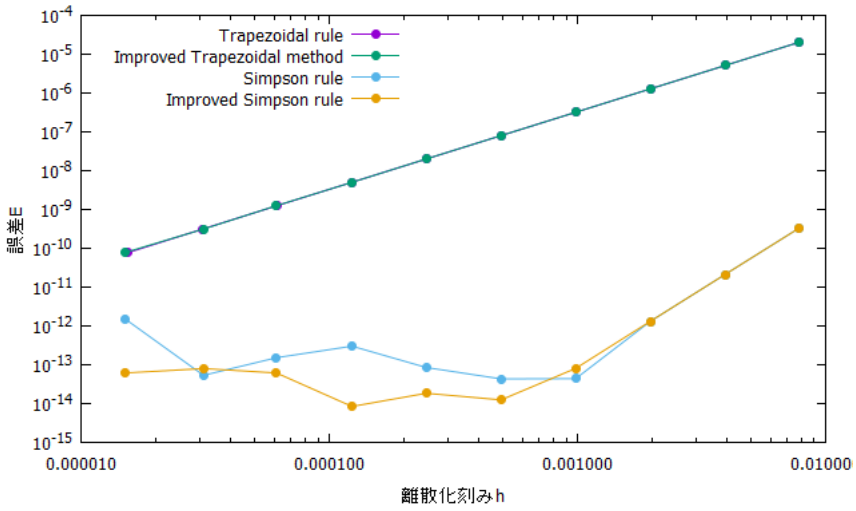


図 2　各数値積分法における離散化刻み $h$ と誤差 $E$ の関係

## 考察

図 2 より, 台形則は傾き 2 の直線, シンプソン則は $h$ が 0.001 より大きい領域では傾き 4 の直線になっている. このことから, 誤差 $E$ は, 台形則では $O(h^2)$, シンプソン則では $O(h^4)$ であることがわかる. また, 小さい数から順に足すように和のとる順序を変えたとき, 台形則では変化が生じなかったが, シンプソン則では $h$ が 0.001 より小さいとき, 誤差が減少した. また, シンプソン則は $h$ が 0.001 より小さいと, 離散化刻みを減少させると, 逆に誤差が増加している. これは離散化刻みを小さくしたことで減少する離散化誤差より, 反復回数が増えたことで増加する丸め誤差や情報落ち誤差のほうが大きくなってしまうからである.

# 問 6

**問題**

無限区間積分を行う問題に対して

$$I = \int_0^{\infty} e^{-x} \mathrm{d}x \tag{6}$$

の以下 2 つの方法の差異を調べよ. (1) 無限大をある大きな正の値 $R$ としたとき, $R$ を $1000, 10000, 100000$ と大きくしていくことにより無限区間積分を近似する. この時, $R$ に対する $I$ の値を求めて, 解析解への漸近の様子を図示せよ. (2) スケール変換 $x = \frac{1+u}{1-u}$ を用いて, 有限区間の定積分に変換し積分値を求めよ. ただし, 数値積分区間内に発散点を含まないように気をつけて積分区間を設定せよ.

**方法**

(1)

$$I = \int_0^{R} e^{-x} \mathrm{d}x \tag{7}$$

を $R$ の値を変えながら, 台形則を用いて数値積分を行う.刻み幅は 0.001 とする.
(2)$x = \frac{1+u}{1-u}$ とスケール変換し, $\frac{\mathrm{d}x}{\mathrm{d}u} = \frac{2}{(1-u)^2}$ より, 式 (6) は

$$I = \int_{-1}^{1} e^{-\frac{1+u}{1-u}} \frac{2}{(1-u)^2} \mathrm{d}u \tag{8}$$

となる.この積分を台形則を用いて値を求める.刻み幅は 0.000001 とする.ここで, $x = 1$ が発散点であり, そのままの積分範囲で台形則を用いると, 値が無限大になってしまう.したがって, 積分範囲を $-1$〜0.999999 とし積分範囲に発散点を含めないようにする.

**結果**

式 (6) の解析解は 1 である.
(1)prob6_1.c を実行し, 得られたデータをグラフにしたものが図 3 である. 緑線が数値解, 緑線が解析解である. また,R= 1 ,10,100,1000,10000,100000 に対する計算結果が表 11 である.

表 11　R の値と計算結果

| R | 計算結果 |
|---|---|
| 1 | 0.63212061150527021347 |
| 10 | 0.99995468339973458960 |
| 100 | 1.00000008344700153096 |
| 1000 | 1.00000006660046314444 |
| 10000 | 1.00000166187330696843 |
| 100000 | 1.00002616781962272441 |

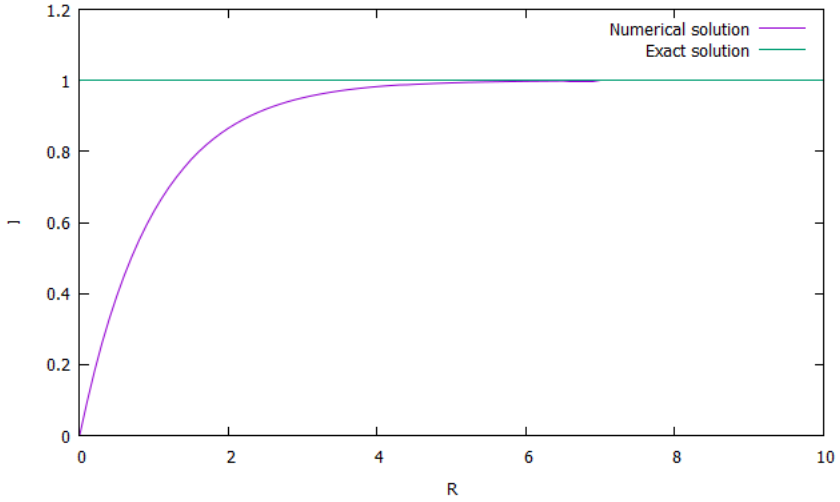


図 3　$R$ と $I$ の関係

(2)prob6_2.c を実行すると, 計算結果は 1.0000002499981897052 8 である.

**考察**

(1)R を大きくすると計算結果は解析解とほとんど同じになった. しかし, この手法では被積分関数により R の値を変える必要があり, また R が大きすぎると計算量が増大してしまう.
(2) スケール変換により積分範囲を有界にすることで,(1) の手法のように R の値を考慮する必要がなくなり, また積分範囲が狭くなるので計算量も減少する. しかし, スケール変換をすると積分範囲に発散点が含まれてしまうので, 数値計算する際は発散点を含まないように積分範囲を変える必要があり, また発散点付近での計算精度は悪くなる.

# 問 7

**問題**

正則化済み不完全ベータ関数 $\beta(x:a,b)(a>0,b>0)(0\leq x\leq 1)$

$$\beta(x;a,b)=\frac{\int_0^x t^{a-1}(1-t)^{b-1}\mathrm{d}t}{\int_0^1 t^{a-1}(1-t)^{b-1}\mathrm{d}t} \tag{9}$$

をシンプソン則を用いて計算するためのプログラムを作成せよ.

**方法**

分母, 分子をそれぞれシンプソン則を用いて数値積分し, 最後に分数に戻すプログラムを組む. シンプソン則で用いる関数は $f(t)=t^{a-1}(1-t)^{b-1}$, 刻み幅は 0.0001 とする. ここで, シンプソン則では $f(0)$ と $f(1)$ の値を用いるが, $a<1$ のときは $f(0), b<1$ のときは $f(1)$ の値が計算機では求めることができない. そこで, $f(0)$ と $f(1)$ の値のかわりに $f(10^{-16})$ と $f(1-10^{-16})$ の値を使う.
$x$ の値を 0 から 1 まで 0.01 ずつ増やしながら正則化済み不完全ベータ関数を計算し, プロットする.

**結果**

$a = 2, b = 3$ のとき,prob7.cを実行し, データをプロットしたものが図 4 である.

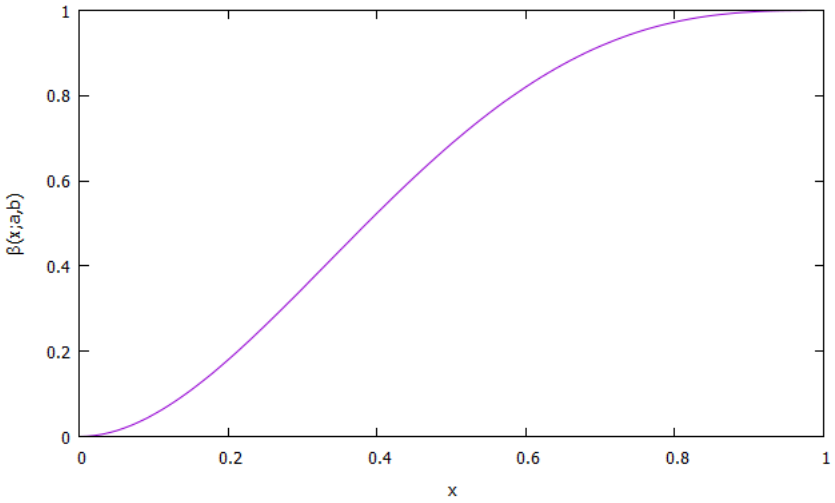


図 4　$x$ と$\beta(x; a, b)$ の関係

.

**考察**

$a = 2, b = 3, x = 0.4$ のときのプログラムで求めた値は 0.524800 であった. 解析的に求めると $\beta(0.4; 2, 3) = \frac{328}{625} = 0.5248$ であり, 確かに正確に求められている.

## まとめ

解析的に解くことのできない微分方程式や積分でも数値計算では数値的に求めることができるため, 数値計算は非常に強力である. しかし, 厳密解との誤差は必然的に生じてしまう. また, 計算機特有の浮動小数点表示に適したプログラムを設計しないと様々な誤差が生じる. 精度の良い数値計算を行うには, アルゴリズムだけでなく, 計算機での演算の仕組みも理解することが必要である.

## 参考文献

[1]「1 週間で学ぶ IT 基礎の基礎 - 【5 分で覚える IT 基礎の基礎】ゼロから学ぶ 2 進数　第 3 回 : ITpro」<http://itpro.nikkeibp.co.jp/members/ITPro/ITBASIC/20020619/1/> (2017/5/3 アクセス)

[2]「ビットで表す数字の世界〜浮動小数点編〜」<http://www.altima.jp/column/fpga_edison/bit_number_float.html> (2017/5/3 アクセス)