# 高次代数方程式の数値解法ライブラリの研究開発

廣田 千明 *　　小澤 一文 *
* 秋田県立大学システム科学技術学部

**概要** 代数方程式の数値解法の一つである Durand–Kerner 法のライブラリ・プログラムを提供する。提供するプログラムは倍精度版、多倍長版、並列多倍長版の 3 つで、解くべき問題の規模、性質、あるいは計算機環境により使い分ける必要がある。これらのプログラムの解説および使用法を紹介する。

## 1　はじめに

数値計算を行うにおいて、我々は、連立一次方程式の解法、乱数生成、代数方程式の解法などのライブラリ・プログラムを頻繁に利用する。また最近は並列計算機を用いて数値計算を行うことが多くなった。プログラムの並列化に関しては、連立一次方程式、固有値計算などの線形計算が主で、代数方程式の数値解法は必ずしも十分に研究されていないようである。本研究開発は、高次代数方程式の根を並列計算機上で高速に求めるためのサブルーチンライブラリを作成し、提供することを目的としている。

代数方程式の数値解法としては、これまでに多くのものが開発されてきたが、並列計算向きで収束性、安定性に優れているのが Durand–Kerner 型解法である（Durand–Kerner 型解法については例えば [5, pp.45–59] を参照せよ)。ここでは、いくつかある Durand–Kerner 型解法の中で 3 次の Durand–Kerner 法 (Ehrlich–Aberth 法)（以下簡単に DK 法と略す）の Fortran プログラムを作成する。ここで提供するのは、倍精度版、多倍長精度版、それに並列多倍長精度版の 3 つである。この 3 つのうち、倍精度版は PC や EWS で標準となっている IEEE754 規格の倍精度浮動小数点演算 (64bit 浮動小数点演算) に対応したものであり、Fortran 90/95 のコンパイラが動く計算機へはどこにでも移植できる。一方、多倍長精度版の 2 つは多倍長演算のための特殊なライブラリ・プログラムを必要とする。

ここで代数方程式の数値解法において多倍長演算の必要性について説明する。代数方程式は、Wilkinson の古典的な例 ([1, pp.41–43]) にあるように、係数に含まれる微少な誤差や計算過程でのわずかな丸め誤差が、予想もしないような大きさに拡大されることがある。このような問題は悪条件 (ill conditioned) な問題と呼ばれていて、計算精度を上げることによってしか解決できない。したがって多倍長演算を用いた DK 法のライブラリ

が必要となる。

Fortran や C などのコンパイラでは、単精度演算 (10 進 7 桁相当) および倍精度演算 (10 進 16 桁相当) は標準規格として備わっているが、それ以上の精度（拡張倍精度、4 倍精度など）は非標準である。したがって、倍精度演算以上の精度が要求されるがそれ以上の演算が利用できない場合、多倍長演算のソフトウェアを利用しなければならない。また、ほとんどの多倍長演算のソフトウェアでは精度が可変であるため、精度が固定されている拡張倍精度や 4 倍精度より柔軟な対応ができるという利点もある。ここでは多倍長演算ライブラリには FMLIB ライブラリ [8] を利用する。

本稿の構成は以下の通りである。2 節では多倍長演算ライブラリ FMLIB の簡単な使い方を説明する。3 節では代数方程式の数値解法である Durand–Kerner 法について述べ、その並列化について考察する。4 節では数値実験結果を与える。特に負荷均一化の観点から計算の各プロセスへの分担方法を検討する。5 節では提供するライブラリの仕様を説明する。なおライブラリの構築方法やサンプルプログラムは付録として最後に紹介する。

## 2　多倍長演算ライブラリ FMLIB

多倍長演算ライブラリには FMLIB[8]、GMP[9]、mppack[11]、exflib[7] などがあるが、Fortran からの利用が容易であること、各種初等関数が用意されていることなどの理由から、FMLIB を利用することにする。FMLIB は D.M. Smith により作成された Fortran 用多倍長演算ライブラリであり、最新バージョンは 1.2 で `http://myweb.lmu.edu/dmsmith/FMLIB.html` からダウンロードできる。このライブラリは多倍長型として 3 つの型を用意しており (表 1)、四則演算はもちろん、初等関数や各種定数 (円周率やオイラーの定数など) を用意している。FMLIB ライブラリでは構造体によって多倍長数が定義され、演算子も多重定義されている。そのため、Fortran90 規格のコンパイラを用いれば、多倍長数の四則演算や関数計算が (サブルーチンコールを用いないで) そのまま式を書き下すだけで可能になる。そこで Fortran90 での利用を想定し、利用法を簡単に説明する (FMLIB ライブラリの構築方法は付録 A.1 を参照せよ)。

表 1. FMLIB が提供する多倍長型の種類

| | |
|---|---|
| IM | 多倍長整数型 |
| FM | 多倍長浮動小数点型 |
| ZM | 多倍長複素数型 |

FMLIB の多倍長型変数および関数は FMZM というモジュールとして用意されているので、FMLIB の関数や変数を使用するにはプログラムの先頭に

```
use FMZM
```

を書いておく必要がある。

多倍長型変数の宣言は

```
type(IM) :: i,j
type(FM) :: x,y
type(ZM) :: z
```

のように行う。

多倍長浮動小数点型変数の仮数部の精度は、FMSET ルーチンを用いて

```
call FMSET(100)
```

のように設定すれば、10 進で 100 桁相当の演算が可能である。デフォルトでは 50 桁に設定されている。

多倍長数の表示には専用の関数を用いる必要がある。例えば

```
call IMPRNT(i)
call FMPRNT(x)
call ZMPRNT(z)
```

とすれば、それぞれの型の変数が用意されたフォーマットで印字される。また、これとは別に

```
character(LEN=32) :: ST
type(FM) :: x

ST=FM_FORMAT('F32.24',x)
write(*,*) ST
```

のように出力フォーマットを指定することができる。

型変換の関数は表 2 の通り用意されている。

表 2. 型変換関数

| | |
|---|---|
| TO_IM | 多倍長整数型へ |
| TO_FM | 多倍長浮動小数点型へ |
| TO_ZM | 多倍長複素数型へ |
| TO_INT | 整数型へ |
| TO_SP | 単精度浮動小数点型へ |
| TO_DP | 倍精度浮動小数点型へ |
| TO_SPZ | 単精度複素数型へ |
| TO_DPZ | 倍精度複素数型へ |

これらの型変換関数は非常に便利で、例えば多倍長浮動小数点型の定数や変数を定義したいときは型変換関数を用いて

```
x=TO_FM('1.2e10')

a=1.2d0
y=TO_FM(a)
```

とすればよい。また表示が倍精度で十分な場合には、

```
write(*,*) TO_DP(x)
```

として出力することができる。

FMZM モジュールの中で四則演算や比較演算子などが多重定義されているので、多倍長数の演算はサブルーチンコールでなく、次の例のように実行できる。

```
program sample
  use FMZM
  type(FM) :: x,y

  x=TO_FM('2.0')
  y=TO_FM('4.0')

  write(*,*) TO_DP(x+y)
  write(*,*) TO_DP(x*y)
  write(*,*) TO_DP(x/y)
end program sample
```

コンパイルと実行結果は

```
% f95 sample.f90 -lFM
% a.out
   6.00000000000000
   8.00000000000000
  0.500000000000000
```

となる。

科学技術計算において計算中にしばしば円周率 $\pi$ が現れる。多倍長精度を保つには円周率もまた多倍長精度で必要になる。FMLIB を用いると円周率の値も多倍長精度、すなわち FMSET ルーチンで指定した桁数で得ることができる。それには次のようにすればよい。

```
type(FM) :: pi
call FMPI(pi)

call FMPRNT(pi)
```

実行結果は

```
3.14159265358979323846264338327950288419716939937510+0
```

となる。

## 3 Durand–Kerner 法とその並列化

複素係数の $n$ 次代数方程式

$$P_n(z) = a_n z^n + a_{n-1} z^{n-1} + \cdots + a_1 z + a_0 = 0 \tag{1}$$

の数値解法には Durand–Kerner 法 (以下 DK 法と略す) を用いる (詳しくは例えば [5, pp.45–59] を参照せよ)。この解法を用いるとすべての根を同時に求めることができ、減次を行いながら繰り返し Newton 法を適用する方法より精度の面で有利である。DK 法には数々の種類が存在するが、ここでは標準的な 3 次公式を利用する。

### 3.1 3 次の DK 法 (Ehrich–Aberth 法)

ここで用いる 3 次の DK 法は

$$z_i^{(k+1)} = z_i^{(k)} - \frac{P_n(z_i^{(k)})}{P_n'(z_i^{(k)}) - P_n(z_i^{(k)}) \displaystyle\sum_{j \neq i} \frac{1}{z_i^{(k)} - z_j^{(k)}}}, \quad i = 1, \ldots, n,\ k = 0, 1, 2, \ldots \tag{2}$$

で与えられる。ここで $z_i^{(k)}$ は $i$ 番目の根の第 $k$ 近似を表す。この反復公式の初期値として小澤 [4] の初期値

$$z_i^{(0)} = \beta + r \exp\left\{\sqrt{-1}\left(\frac{2\pi(i-1)}{n}\right) + \phi\right\}, \quad i = 1, \ldots, n,$$

$$\beta = -\frac{a_{n-1}}{n\, a_n}, \qquad \phi = \frac{3}{2n}, \qquad r = \left|\frac{P_n(\beta)}{a_n}\right|^{1/n}$$

を用いる。これは根の重心 $\beta$ を中心とした半径 $r$ の複素平面上の円周上に等間隔に初期値を配置する方法である。また反復停止条件は森口 [6, pp.47–48] による条件

$$|P_n(z_i^{(k)})| < K\varepsilon \max_{0 \leq m \leq n} \{(m+1)\,|a_m|\,|z_i^{(k)}|^m\} \tag{3}$$

を用いる。ここで $\varepsilon$ はマシン・エプシロンを表し、$K \geq 1$ は安全係数を表す。$K$ は通常 2 から 3 程度が用いられる。条件 (3) の右辺は $P_n(z_i^{(k)})$ を計算するときに生じうる丸め誤差の上限である。この条件は $|P_n(z_i^{(k)})|$ の計算値がこの丸め誤差の上限より小さくなったとき、$z_i$ についてはこれ以上反復しても精度の改善が望めないので、$k+1$ ステップ以降は反復しないというものである。

浮動小数点演算で標準的な倍精度浮動小数点演算において条件 (3) をみたした近似根がどの程度の精度をもっているか調べてみる。Wilkinson[1, pp.41–43] が悪条件な例として示した多項式

$$\begin{aligned} P_{\mathrm{W}}(z) &= \prod_{i=1}^{20}(z-i) \\ &= z^{20} - 210z^{19} + 20615z^{18} - \cdots + 2432902008176640000 \end{aligned}$$

の根を倍精度で解いてみると、得られた根の相対誤差の最大値は `1.089311E-01` となり、倍精度の計算精度 (10 進法で約 16 桁) に比べて精度が著しく低下していることがわかる。これに対して FMLIB を用いて多倍長計算 (10 進法で 50 桁の精度を使用) を行うと、相対誤差の最大値は `1.836027E-044` となり多倍長計算が有効であることがわかる。

この考察から代数方程式の数値解法には多倍長計算が必要になることもあるが、多倍長計算はソフトウエアにより実装されているので、演算にかかる計算コストは倍精度計算より遥かに大きい。そこで並列化により計算時間の短縮を試みる。

## 3.2 並列 DK 法

反復公式 (2) は $i$ に関して並列に実行できる。したがって、各 $k$ ステップで $i$ について並列に計算を行い、すべての $i$ について近似根が計算できた段階で同期をとり、全プロセス間で各プロセスで計算された近似根のデータを交換する。この手順を次元数 4、プロセス数 2 として図示すると図 1 のようになる。データの交換には MPI(Messeage Passing Interface) [10] を用いることにする。

並列計算を行うとき注意しなければならないのは、一部のプロセスに負荷が偏ると並列化の効率が落ちてしまうことである。DK 法 (2) を式 (3) の反復停止条件のもとで反復すると、プロセス毎に未収束根の個数が異なり、プロセス間で負荷が不均一になる可能性がある。これを防ぐため、著者らは、各反復ステップの開始時に未収束根の個数を数え、それらを各プロセスに均一に配分する方法を提案した [3]。各反復ステップでの配分方法は以下のアルゴリズムを用いる。

次節ではこの [3] による方法が実際にどの程度有効であるか数値実験により検証する。

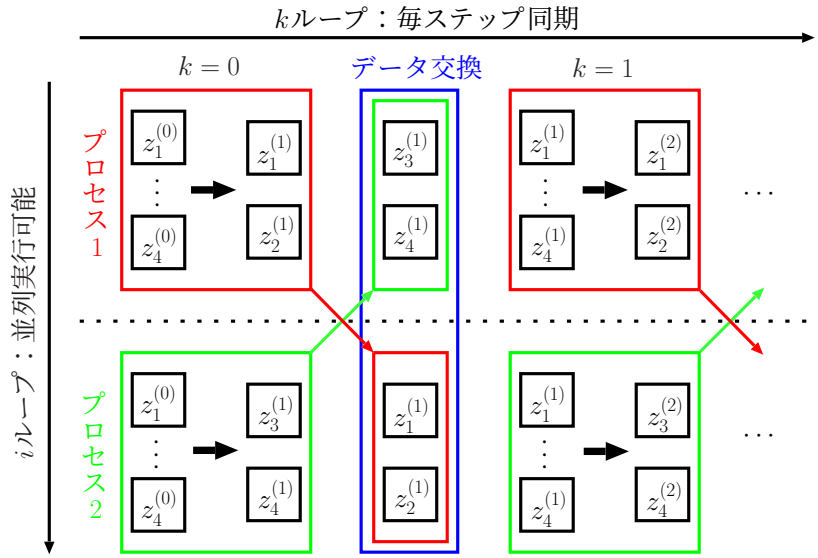


図 1. 近似根の計算手順 (次元数 4、プロセス数 2 の場合)

## 4 数値実験

数値実験には東北大学情報シナジーセンターの並列計算機 TX7/AzusA を使用する[1]。この計算機は 64 ビットプロセッサ Itanium を 16 個搭載した共有メモリ型並列計算機で 51.2GFLOPS の演算性能を有している。

**数値例 4.1 (負荷が不均一になる例)** テスト問題として根 $\alpha_i$ が

$$\alpha_i = \alpha_{i+1} = \exp\left(\frac{2\pi\sqrt{-1}(i-1)}{n}\right), \quad i = 1, 3, 5, \ldots, \lfloor n/2 \rfloor - 1,$$
$$\alpha_i = \exp\left(\frac{2\pi\sqrt{-1}(i-1)}{n}\right), \qquad i = \lfloor n/2 \rfloor + 1, \ldots, n$$

で与えられる代数方程式を考える。これは重根と単根が半分ずつ混じった方程式である。例えば $n = 64$ のとき、図 2 のように根が配置される。

この問題で次元数を $n = 512$ とした場合を DK 法で解いてみる。多倍長数の精度を 10 進法で 50 桁に設定して計算した場合、各近似根が収束までに必要とするステップ数は図 3 のようになる。この問題を並列に解いた場合、各プロセスの負荷バランスはどうなっているのだろうか。例えば 4 プロセスを用いた場合、プロセス 1 は第 1 成分から第 128 成分を担当し、プロセス 2 は第 129 成分から第 256 成分を担当する。以下 128 成分ずつ割り当てられる。分担を変えない場合、図 3 の色分けした各部分の面積が各プロセスの負荷に比例していると考えられるので、プロセス 1、2 は負荷が大きく、プロセス 4 はほと

[1] 今回提供する DK 法ライブラリは新システム TX7/i9610 でも動作することが確認済みである。

分担アルゴリズム

未収束の根の数を $m_0$、総プロセス数を $p$ とする
$m := m_0$
**for** $i = 1$ to $p$ **do**
  **if** $(m \geq \lceil m_0/p \rceil)$ **then**
    第 $i$ プロセスに $\lceil m_0/p \rceil$ 個の未収束根を割り当てる
    $m := m - \lceil m_0/p \rceil$
  **else**
    第 $i$ プロセスに残りの未収束根 ($m$ 個) を全て割り当てる
    $m := 0$
  **end if**
**end for**

んど仕事をしてないことがわかる。したがって分担を変えない場合、非効率になることが予想される。そこで、実際に並列化の効率がどの程度であるか調べるためにプロセス数を変化させて実験を行う。計算時間とスピードアップ

$$\text{スピードアップ} = \frac{1\text{プロセスでの計算時間}}{p\text{プロセスでの計算時間}}$$

を図 4 に与える。この図より各ステップ毎に負荷が均一になるよう分担を変更すれば高いスケーラビリティが得られることがわかる。

**数値例 4.2 (負荷が常に均一の例)** 前の例は反復過程で各プロセスの負荷が著しく不均一になる方程式であったため、我々の提案する動的に分担を変更するアルゴリズムが功を奏した。これに対して次の例は何もしなくとも負荷が常に均一になる方程式である。このような方程式に対して我々の提案するアルゴリズムが効果を発揮するかどうかをみることにする。負荷が常に均一の例として

$$P_n(z) = z^n - 1 = 0 \tag{4}$$

を考える。この代数方程式の根は複素平面上の単位円上に等間隔に並び、全根が同時に収束するという性質をもっている。実際、精度を 10 進 50 桁に設定して反復停止条件を満たすまでの反復回数を 512 個の近似根について調べると図 5 のようになっている。したがって、何もしなくても各プロセスの負荷が均一になることが予想され、動的に分担を変える我々の方法はむしろ不利になることも考えられる。

この例に対して並列化の効率を調べる。プロセス数を変化させて数値実験を行ったときの計算時間とスピードアップを図 6 に与える。この図からわかるように、我々の提案するアルゴリズムは、この問題においても分担を全く変えないシンプルな並列化とほぼ同等の性能を発揮している。

以上の結果から毎ステップ分担をし直す方法を採用した並列 DK 法のライブラリを提供する。次節ではライブラリの仕様を説明する。

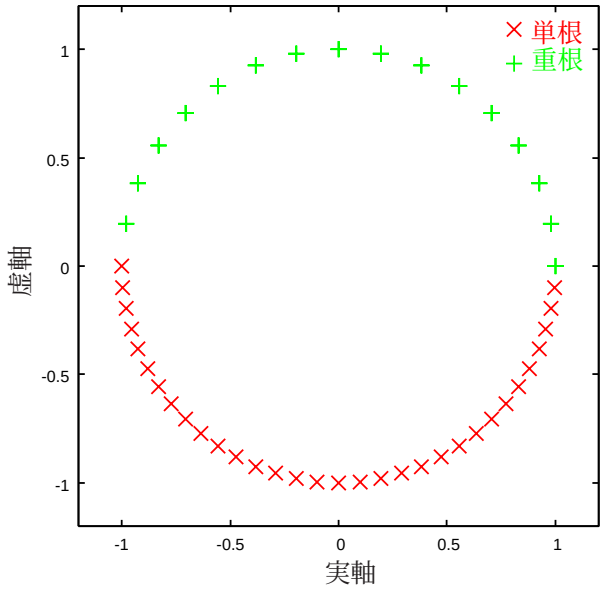


図 2. 根の配置 ($n = 64$)

## 5 DK 法ライブラリ

倍精度 DK 法、多倍長 DK 法、並列多倍長 DK 法の 3 つのサブルーチンをまとめたライブラリを提供する。ライブラリの構築方法は付録 A.2 を参照せよ。

### 5.1 倍精度 DK 法

倍精度 DK 法のサブルーチンは 6 つの引数をとる (表 3)。

```
subroutine DK(n,z,coef,itr,max_itr,isort)
```

入力として方程式の次数 `n` と係数を格納する配列 `coef` を与える必要がある。配列 `coef` には代数方程式 (1) の係数 $a_i$ を次元の低い方から順に

$$\texttt{coef}(i) := a_i, \qquad i = 0, \ldots, n$$

のように格納する。またプログラムの暴走を防ぐために、(2) の $k$ ループに対する最大反復回数を `max_itr` として与える必要がある。引数 `isort` は正の値のとき、得られた根を実部が昇順に並ぶようにソートし、負のときは降順にソートする。また 0 のときはなにもしない。出力用として用意する必要があるのは計算結果を格納する多倍長複素数型配列 `z(n)` と計算ステップ数を格納する整数型変数 `itr` である。以上を用意してサブルーチン `DK` を呼び出せば計算結果が出力用変数に格納される。サンプルプログラムとして数値例 4.2 を解くプログラム (`dr_DK.f90`) を付録 B.1 に与える。

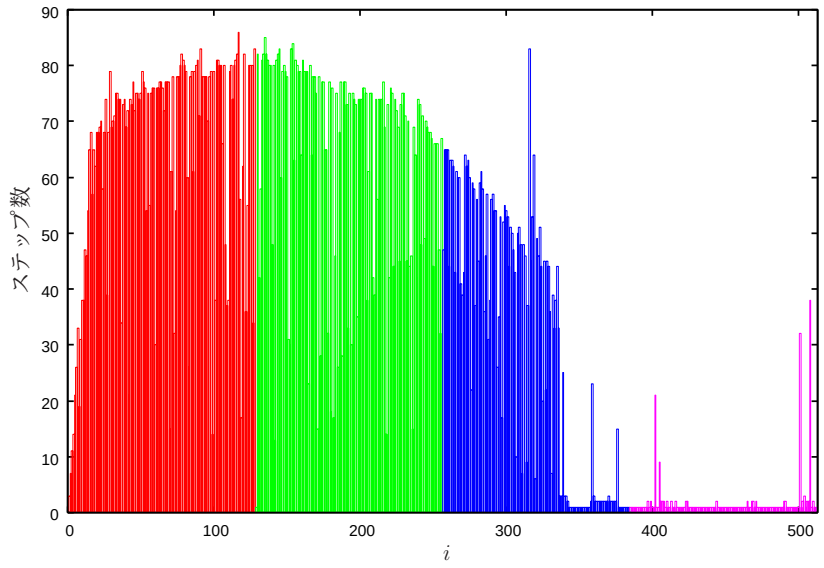


図 3. 反復停止条件をみたすまでに必要なステップ数 (例 4.1、512 次元)

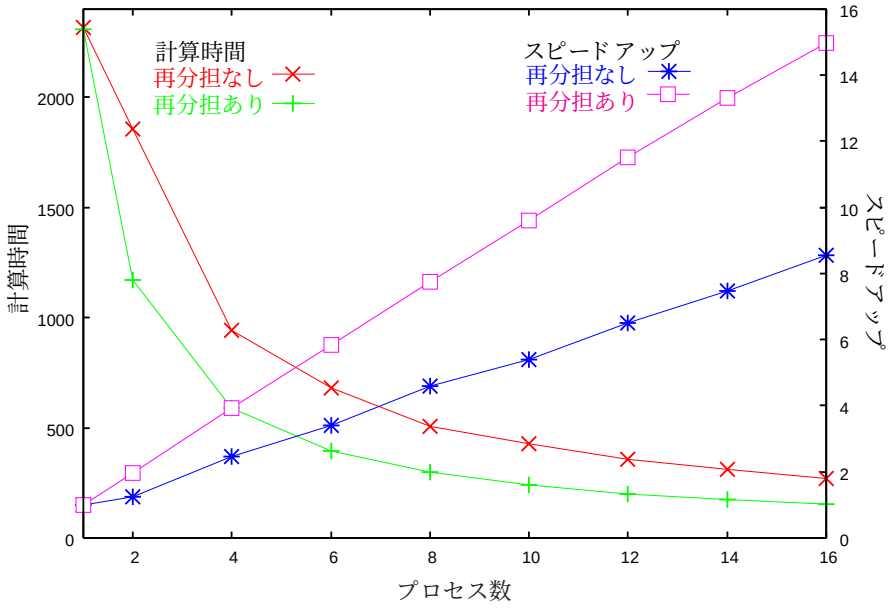


図 4. 計算時間とスピードアップ (例 4.1、512 次元)

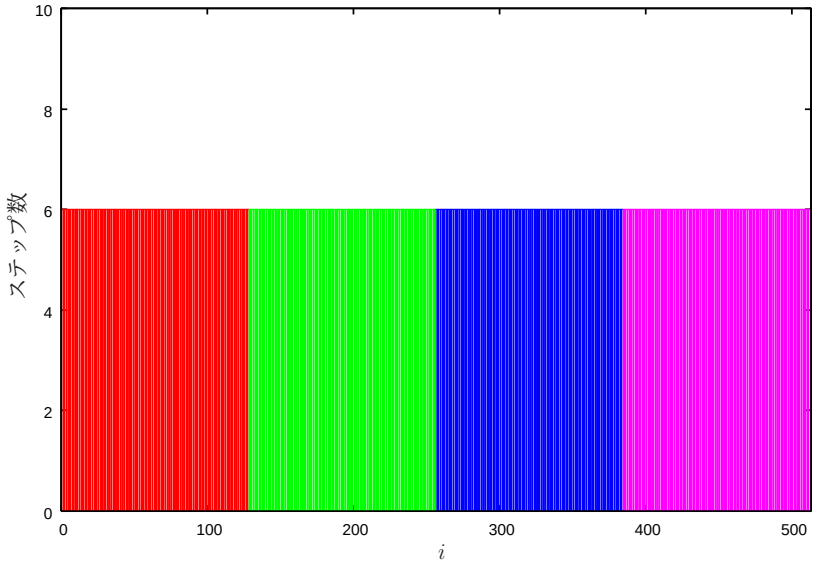


図 5. 反復停止条件をみたすまでに必要なステップ数 (例 4.2、512 次元)

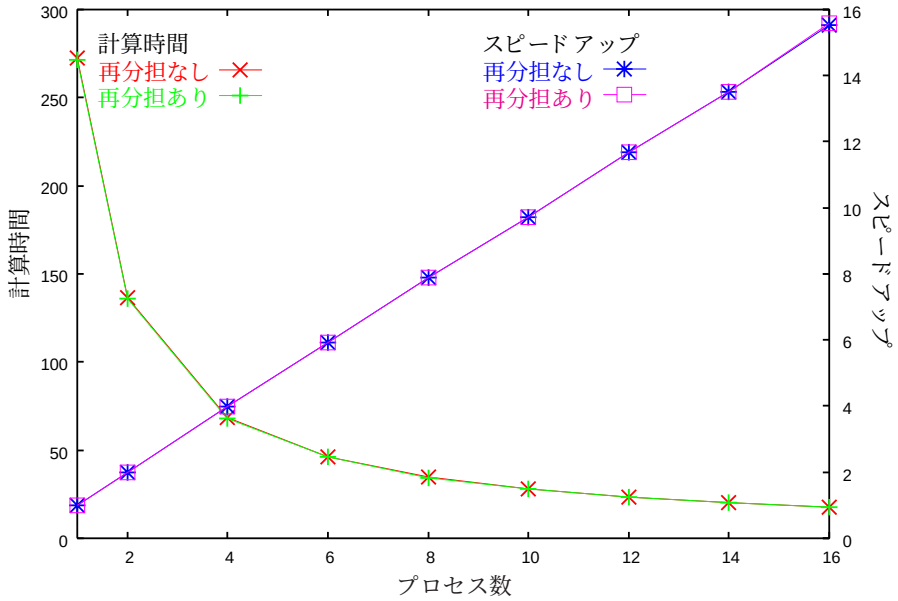


図 6. 計算時間とスピードアップ (例 4.2、512 次元)

## 5.2 多倍長 DK 法

多倍長 DK 法のサブルーチンは 7 つに引数をとる (表 4)。

```
subroutine FM_DK(n,z,coef,itr,acr,max_itr,isort)
```

倍精度 DK 法ルーチンとの違いは入力として多倍長型変数の精度を表す整数型変数 acr を与える必要があることである。acr は 10 進法での計算桁数を指定する。数値例 4.2 を解くサンプルプログラム (dr_FM_DK.f90) を付録 B.2 に与える。

## 5.3 並列多倍長 DK 法

並列多倍長 DK 法のサブルーチンは 8 つの引数をとる (表 5)。

```
subroutine FM_DK_mpi(n,z,coef,itr,acr,max_itr,isort,comm)
```

多倍長 DK 法ルーチンとの違いは入力用引数として MPI のコミュニケータを指定する必要があることである。また MPI ライブラリを使用するのでそのための準備として MPI の初期化、終了などは各ユーザが行う必要がある。MPI ライブラリの使用法は MPI の解説書 (例えば [2]) を参照せよ。数値例 4.2 を解くサンプルプログラム (dr_FM_DK_mpi.f90) を付録 B.3 に与える。

表 3. DK サブルーチンの引数

| | | |
|---|---|---|
| n | 入力用 | 代数方程式の次元 (整数型) |
| z | 出力用 | 代数方程式の根を格納する配列 (ZM 型) |
| coef | 入力用 | 代数方程式の係数を格納した配列 (ZM 型) |
| itr | 出力用 | 反復停止までに要したステップ数 (整数型) |
| max_itr | 入力用 | 最大反復回数 (整数型) |
| isort | 入力用 | ソートの種類 (整数型) |

表 4. FM_DK サブルーチンの引数

| | | |
|---|---|---|
| n | 入力用 | 代数方程式の次元 (整数型) |
| z | 出力用 | 代数方程式の根を格納する配列 (ZM 型) |
| coef | 入力用 | 代数方程式の係数を格納した配列 (ZM 型) |
| itr | 出力用 | 反復停止までに要したステップ数 (整数型) |
| acr | 入力用 | 計算桁数 (整数型) |
| max_itr | 入力用 | 最大反復回数 (整数型) |
| isort | 入力用 | ソートの種類 (整数型) |

表 5. FM_DK_mpi サブルーチンの引数

| | | |
|---|---|---|
| n | 入力用 | 代数方程式の次元 (整数型) |
| z | 出力用 | 代数方程式の根を格納する配列 (ZM 型) |
| coef | 入力用 | 代数方程式の係数を格納した配列 (ZM 型) |
| itr | 出力用 | 反復停止までに要したステップ数 (整数型) |
| acr | 入力用 | 計算桁数 (整数型) |
| max_itr | 入力用 | 最大反復回数 (整数型) |
| isort | 入力用 | ソートの種類 (整数型) |
| comm | 入力用 | MPI のコミュニケータ (整数型) |

### 5.4 実行例

代数方程式

$$P_4(z) = z^4 - 6z + 15z^2 - 18z + 10 = 0$$

の根は

$$z_1 = 1 - \sqrt{-1}, \quad z_2 = 1 + \sqrt{-1}, \quad z_3 = 2 - \sqrt{-1}, \quad z_4 = 2 + \sqrt{-1}$$

である。この方程式を DK ライブラリを用いて解いたときの誤差 ($|z_i^{(k)} - z_i|$) を表 6 に示す。ただし多倍長数の精度は 10 進 50 桁としている。

表 6. 実行結果

| $i$ | 根 (実部, 虚部) | 誤差 (倍精度) | 誤差 (多倍長) |
|---|---|---|---|
| 1 | (1.00E+00, -1.00E+00) | 6.66E-16 | 0.00E+00 |
| 2 | (1.00E+00, 1.00E+00) | 5.55E-16 | 0.00E+00 |
| 3 | (2.00E+00, -1.00E+00) | 6.66E-16 | 0.00E+00 |
| 4 | (2.00E+00, 1.00E+00) | 8.01E-16 | 3.49E-57 |

**謝辞** 東北大学情報シナジーセンターのライブラリ研究開発の援助を受けたことを感謝いたします。開発作業の一部には九州大学情報基盤センターの並列計算機を利用いたしました。同センター 藤野清次 教授に感謝いたします。

## 参考文献

[1] Wilkinson J.H., Rounding errors in algebraic processes, Dover, 1994.

[2] 青山幸也, 並列プログラミング入門 MPI 版: http://accc.riken.jp/HPC/training/text.html .

[3] 遠藤大喜, 廣田千明, 小澤一文, MPI を用いた超高次代数方程式の並列解法, 平成 17 年度電気関係学会東北支部連合大会講演論文集, p.168.

[4] 小澤一文, Durand–Kerner 法の効率的な初期値の簡単な設定法, 日本応用数理学会論文誌 **3** (1993), 451–464.

[5] 藤野清次, 数値計算の基礎 —数値解法を中心に—, サイエンス社, 1998.

[6] 森口繁一, 数値計算工学, 岩波書店, 1989.

[7] exflib: `http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/exflib/index.html` .

[8] FMLIB: `http://myweb.lmu.edu/dmsmith/FMLIB.html` .

[9] GMP: `http://www.swox.com/gmp/` .

[10] MPI standard: `http://www-unix.mcs.anl.gov/mpi/` .

[11] mppack: `http://phase.hpcc.jp/phase/mppack/` .

# A ライブラリの構築方法

## A.1 FMLIB ライブラリの構築

FMLIB ライブラリのソースプログラムは 3 つのファイル (`FM.f90`, `FMZM90.f90`, `FMSAVE.f90`) からなる。これらをコンパイルし、1 つのアーカイブとしてまとめておくと便利である。

```
% f95 -c FM.f90
% f95 -c FMZM90.f90
% f95 -c FMSAVE.f90
% ar ru libFM.a FM.o FMZM90.o FMSAVE.o
% ranlib libFM.a
```

以上によりライブラリファイル (`libFM.a`) とモジュールファイル (`fmvals.mod`, `fmzm_1.mod`, `fmzm_2.mod`, . . . , `fmzm_9.mod`, `fmzm.mod`) を作成することができる。コンパイル時にモジュールファイルのパスを指定し、ライブラリファイルをリンクすることにより FMLIB が利用できる。

```
% f95 -module [モジュールへのパス] [メインプログラム] -L[ライ
  ブラリのパス] -lFM
```

## A.2 DK ライブラリの構築

DK ライブラリのソースプログラムは 3 つのファイル (DK.f90, FM_DK.f90, FM_DM_mpi.f90) からなる。これらをコンパイルし結合することでライブラリを構築する。

```
% f95 -c DK.f90
% f95 -c FM_DK.f90
% mpif95 -c FM_DK_mpi.f90
% ar ru libDK.a DK.o FM_DK.o FM_DK_mpi.o
% ranlib libDK.a
```

以上によりライブラリファイル (libDK.a) を作成することができる。このファイルと libFM.a をリンクし、メインプログラム内でヘッダファイル DK.h をインクルードすることにより多倍長 DK 法を利用できる。

```
% f95 -module [モジュールへのパス] [メインプログラム] -L[ライ
  ブラリのパス] -lFM  -lDK
```

# B サンプルプログラム

## B.1 倍精度 DK 法のサンプルプログラム

```
program dr_DK
  implicit none
  include 'DK.h'
  integer,parameter :: n=4,max_itr=1024,isort=1
  complex(8) :: z(n),coef(0:n)
  integer :: itr,i

  !係数の設定 P(z)=z^n-1
  coef(0)=cmplx(-1.d0,0.d0)
  do i=1,n-1
     coef(i)=cmplx(0.d0,0.d0)
  enddo
  coef(n)=cmplx(1.d0,0.d0)

  call DK(n,z,coef,itr,max_itr,isort)

  !結果の表示
  do i=1,n
     write(*,*) i,z(i)
  enddo
  write(*,*) 'itr=',itr
end program dr_DK
```

## B.2 多倍長 DK 法のサンプルプログラム

```
program dr_FM_DK
  use FMZM
  implicit none
  include 'DK.h'
  integer,parameter :: n=4,acr=50,max_itr=1024,isort=1
  type(ZM) :: z(n),coef(0:n)
  integer :: itr,i

  call FMSET(acr)

  !係数の設定 P(z)=z^n-1
  coef(0)=cmplx(TO_FM('-1.0'),TO_FM('0.0'))
  do i=1,n-1
     coef(i)=cmplx(TO_FM('0.0'),TO_FM('0.0'))
  enddo
  coef(n)=cmplx(TO_FM('1.0'),TO_FM('0.0'))

  call FM_DK(n,z,coef,itr,acr,max_itr,isort)

  !結果の表示
  do i=1,n
     write(*,*) i,TO_DPZ(z(i))
  enddo
  write(*,*) 'itr=',itr

end program dr_FM_DK
```

## B.3 並列多倍長 DK 法のサンプルプログラム

```
program dr_FM_DK_mpi
  use FMZM
  implicit none
  include 'mpif.h'
  include 'DK.h'
  integer,parameter :: n=4,acr=50,max_itr=1024,isort=1
  type(ZM) :: z(n),coef(0:n)
  integer :: itr
  integer :: ierr,i,my_rank

  call FMSET(acr)

  !係数の設定 P(z)=z^n-1
  coef(0)=cmplx(TO_FM('-1.0'),TO_FM('0.0'))
  do i=1,n-1
     coef(i)=cmplx(TO_FM('0.0'),TO_FM('0.0'))
  enddo
  coef(n)=cmplx(TO_FM('1.0'),TO_FM('0.0'))

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD,my_rank,ierr)

  call FM_DK_mpi(n,z,coef,itr,acr,max_itr,isort,MPI_COMM_WORLD)

  !結果の表示
  if (my_rank==0) then
     do i=1,n
        write(*,*) i,TO_DPZ(z(i))
     enddo
     write(*,*) 'itr=',itr
  endif

  call MPI_Finalize(ierr)

end program dr_FM_DK_mpi
```