2003年度

卒業論文

連立一次方程式に対する
直接法の比較研究

2004年2月5日

指導教授： 山本　哲朗　教授

早稲田大学　理工学部　情報学科

G00P056-4　佐藤　裕介

目 次

第 1 章

序論

1.1 背景

数値計算とは, 方程式など数学的に定式化された問題を数値的に処理するための技術の総称である。問題を数値的に処理すると述べたが, 通常コンピュータを使うことになる．しかしコンピュータ技術が発達した現在においても, コンピュータは離散点しか扱えないという基本的な問題が存在する。そのためコンピュータで連続数学の問題を厳密に解くことは難しいのである。一般のコンピュータでは倍精度浮動小数点数を用いた小数以下１６桁有効桁数の浮動小数点数演算機能が備わっているので浮動小数点演算が使われるのが一般的である. よって実数演算を浮動小数点数で近似するために, 四則演算の結果は丸められ理論上, 無限けたの実数もコンピュータ上では当然有限けたの数として扱わなければならない. ここにコンピュータ内部における数の表現の問題が起こる．このとき, 無限けたの実数を有限けたで表現するのであるから, 当然, 誤差が生じる．これを丸め誤差と言う. この丸め誤差の存在が数値計算アルゴリズムの振舞いを複雑なものにしている元である． しかし, このような丸め誤差を厳密に評価する数値計算技術, いわゆる区間演算も実用になりつつある．数値計算によって連続数学の問題に対しても数学的に正しいことが保証された結論を導くための技術が発達してきた. つまり, 一つの実数に一つの有限けたの数を対応させるのではなく, それを含む区間 (区間の端点は有限けたの数である) を対応させ, 演算も区間の演算に拡張し, 数値計算全体を区間の演算で行うことも可能となった。以前は数値計算の誤差をよりよく評価するためには時間や手間がかかるといった理由のため, 難しいと思われてきた. しかし今現在では理論的にも実用的にも高精度で効率よく実現できることが明らかにされ, より深い研究が行われ続けている.

1.2 本論文の目的

コンピュータで行われる数値計算の時間のほとんどが、連立一次方程式を解いている時間だといわれるくらい、連立一次方程式は頻出する問題である。例えば電気回路、X 線トポグラフィ、線形の経済モデル、また偏微分方程式の解法である境界要素法、有限要素法、

差分法なども全て連立方程式に帰着される

ところがこのように工学的な応用で、連立 1 次方程式を解くために実際に利用されている方法は (ほとんどの) 線形代数の本には載っていない。実際に利用される解法は色々あるが、今回はその中でも、最もよく知られている直接法について様々な視点から考えてみた。

1.3 本論文の構成

本論文の構成は以下の通りである。

第 2 章では、連立一次方程式を解く準備として、浮動小数点システムの基本的な概念について述べる。

第 3 章では、連立一次方程式を解く方法である直接法の様々な解法についてのアルゴリズムや特徴について述べる。

第 4 章では、シミュレーションによる実行結果を示す。

第 5 章では、まとめや実行結果に対する考察、結論を述べる。

第 2 章

準備

2.1 はじめに

元来、数値計算を行う上では四則演算の結果が計算ごとに四捨五入などにより丸められたり、無限演算を有限演算に近似するなどで正確な結果が得られなかった。しかしここ最近のコンピュータ技術の発展やアルゴリズムの研究により、計算時間の高速化や真の解との誤差がより小さい精度の良い近似解を得ることもできるようになってきた。

本章では、誤差の原因となる浮動小数点数と浮動小数点数への丸めの指定について述べる。

2.2 浮動小数点数

現代の数値計算では、浮動小数点数が標準として用いられている。浮動小数点数は現在では標準化が進められ、ほとんどのパソコンやワークステーション、並列計算機などで同じ形式が用いられている。この体系に基づいて高速に浮動小数点演算が実行される回路がコンピュータに実装されているため、数値計算は浮動小数点数体系上で行われることがほとんどである。

そのために、まず浮動少数点数について詳しく説明する。浮動小数点数については IEEE 標準 754(IEEE standard 754, 以下 IEEE754 と略記する。) がパソコンやワークステーションなど、多くのコンピュータで標準的に用いられている。以下において IEEE754 に基づく 2 進数浮動小数点数システムを考えることにする。

2.2.1 浮動小数点数

IEEE754 では 4 つのタイプの数がある。それは、規格化 2 進浮動小数点数、零、非規格化 2 審浮動小数点数、NaN(Not a Number、非数) である。

規格化 2 進浮動小数点数

規格化 2 進浮動小数点数とは

$$a = \pm(\frac{1}{2} + \frac{d_2}{2^2} + \frac{d_3}{2^3} \cdots + \frac{d_N}{2^N}) \times 2^e, \quad (d_i = 0 \text{ または } 1).$$

とかける数をいう。$e_{\min}$ を負の整数、$e_{\max}$ を正の整数として e は $e_{\min} \leqq e \leqq e_{\max}$ となる整数である

$$m = \frac{1}{2} + \frac{d_2}{2^2} + \frac{d_3}{2^3} \cdots + \frac{d_N}{2^N}$$

を符号付き仮数 (signed mantissa) といい e を指数 (exponent) という。このとき指数 e も 2 進数で表される。単精度、倍精度、拡張倍制度の浮動小数点システムがあるが、それぞれは N が以下のようなものである。

$$N = 24, \quad (-126 \leqq e \leqq 127),$$
$$N = 53, \quad (-1022 \leqq \mathrm{e} \leqq 1023),$$
$$N = 64, \quad (-16382 \leqq \mathrm{e} \leqq 16383).$$

規格化 2 進浮動小数点において表される数の絶対値の最大値は

$$x_{\max} = (\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} \cdots + \frac{1}{2^N}) \times 2^{e_{\max}}$$

であり、その最小値は

$$x_{\min} = \frac{1}{2} \times 2^{e_{\min}}$$

である。$|x| > x_{\max}$ のときにオーバーフローが生じる。

非規格化 2 進浮動小数点数

IEEE754 では浮動小数点数は指数部が $e_{\min}$ となったとき、仮数部の最初の桁が 1 より小さい数を表すために、デフォルトで最初の桁を 1 とすることをやめ、ここが 0 となる数をおくことを可能とする規格となっている。これを非規格化数 (denormalized number) という。非規格化数の範囲に数が入ることを、漸近アンダーフロー (gradual underflow) という。

このような非規格化浮動小数点数の正で最も小さな数は

$$2^{-1074} = 4.940656458412465\cdots \times 10^{-324}$$

である。これ以下の数になると、アンダーフローが生じる。

NaN

このほかに。IEEE754 では次のような特別な数が用意されている。

- NaN(Not a Number) は $\sqrt{-5}, \frac{\infty}{\infty}, +\infty + (-\infty)$ など不当な演算の結果として得られる。
- $\pm\infty$ はオーバーフローの結果や零で割った結果として得られる。
- ± 0　はアンダーフローか　$\pm\infty$　での割り算の結果としての得られる。

零

零は規格化されて

$$a = +(\frac{0}{2} + \frac{0}{2^2} + \frac{0}{2^3}\cdots + \frac{0}{2^N}) \times 2^{e_{min}}$$

と表される。

2.2.2 丸め

IEEE754 では、次の 4 つの丸めモードが指定できる。ここで c を実数 ($c \in R$) とする。

上向きの丸め (round upward)

c 以上の浮動小数点数の中で最も小さい数に丸める。これを $\triangle : R \to F$ と表す。アルゴリズムでの表記は UP とする。

下向きの丸め (round downward)

c 以下の浮動小数点数の中で最も大きい数に丸める。これを $\triangledown : R \to F$ と表す。アルゴリズムでの表記は DOWN とする。

最近点への丸め (round nearest)

c に最も近い浮動小数点数に丸める。これを $\square : R \to F$ と表す。アルゴリズムでの表記は NEAR. とする。もし、このような点が 2 点ある場合には、仮数部の最後のビットが偶数である浮動小数点数に丸める。これを偶数丸め方式 (round to even) という。

切り捨て (round downward 0)

絶対値が c 以下の浮動小数点数の中で、c に最も近いものに丸める。アルゴリズムでの表記は ZERO とする。

2.2.3 浮動小数点数の性質

丸めの演算を写像として $\bigcirc : R \to F$ と書く。このとき $\bigcirc$ は $\triangle, bigtriangledown, \square$ のいずれかと考える。IEEE754 では丸めの演算は次の条件を満たす。

$$\bigcirc x = x, \text{ (任意の } x \in F \text{ において)}$$
$$x \leqq y \to \bigcirc x \leqq \bigcirc y \text{ (任意の } x, y \in R \text{ について)}$$

また $x \in F$ のとき、符号を変えることにより、$-x$ や $|x|$ が得られるので、これらは正確に計算される。IEEE754 では、次のような性質が成立する。

$$\square\,(-x) = -\square\, x, \text{(任意の } x \in R \text{ について)}$$
$$\triangle(-x) = -\bigtriangledown x, \text{ (任意の } x \in R \text{ について)}$$
$$\bigtriangledown(-x) = -\triangle x. \text{ (任意の } x \in R \text{ について)}$$

IEEE754 では、浮動小数点数演算 (F 上での四則演算) は丸めとの関係により、次のように定義されている。

$\cdot \in +, -, \times, /, \bigcirc \in \triangle, \bigtriangledown, \square$ のとき

$$x \odot y = \bigcirc(x \cdot y) \text{ (任意の } x, y \in R \text{ について)}$$

この式は、左辺の浮動小数点数の四則演算の結果 $x \odot y$ は、右辺の数学的に正しい (実数としての) 四則演算の結果 $x \cdot y$ を指定された丸めを行って得られた数 $\bigcirc(x \cdot y)$ に一致するように計算することを表している。

また平方根も

$$(\sqrt{x})_{fp} = \bigcirc(\sqrt{x}) \text{ (任意の } x, y \in R \text{ について)}$$

と、浮動小数点演算によって計算された平方根 $(\sqrt{x})_{fp}$ は、正確な実数演算で計算された平方根 $\sqrt{x}$ を指定された丸めの方向へ丸めた数となる。注意すべきことは、指数関数や三角関数などはこのような規格を満たしていない。

2.3　むすび

本章では誤差について関わりのある浮動小数点数と浮動小数点数への丸めの指定について述べた。次章では、一次方程式の連立方程式を解く方法の一つである直接法について述べる。

第 3 章

直接法

3.1 はじめに

前章では、誤差と関わりのある浮動小数点数について述べた。この章では連立一次方程式を解く解法の一つである直接法について言及する。直接法とは、もし個々の演算に丸め誤差がなければ、有限回の演算で解が得られる方法である。今回はガウスの消去法の具体的な中身と改良やLU分解などとの比較などを考える。

3.2 連立一次方程式について

連立一次方程式は

$$
\begin{aligned}
a_{0.0}x_0 + a_{0.1}x_1 + \cdots + a_{0.n-1}x_{n-1} &= b_0 \\
a_{1.0}x_0 + a_{1.1}x_1 + \cdots + a_{1.n-1}x_{n-1} &= b_1 \\
&\vdots \\
a_{n-1.0}x_0 + a_{n-1.1}x_1 + \cdots + a_{n-1.n-1}x_{n-1} &= b_{n-1}
\end{aligned}
$$

の形で表される。ここで $a_{i.j}$ を $n \times n$ の正方行列 (以下 A)、x_i, b_i をn次元ベクトル (以下 B, X) として表現すると、上の式は

$$
AX = B
$$

と行列とベクトルで簡単に表現できる。

$$
\begin{pmatrix}
a_{1,1} & \cdots & a_{1,n} \\
\vdots & \ddots & \vdots \\
a_{n,1} & \cdots & a_{n,n}
\end{pmatrix}
\times
\begin{pmatrix}
x_1 \\ \vdots \\ x_n
\end{pmatrix}
=
\begin{pmatrix}
b_1 \\ \vdots \\ b_n
\end{pmatrix}
$$

連立一次方程式を解くとは、n個の未知数 $x_0, \cdots x_{n-1}$ をn個の一次方程式全てを満たす様に求めることである。

3.3 解法の比較

ここでは直接法の解法のうち、部分ピボット選択を行わない Gauss-Jordan 法、部分ピボット選択を行う Gauss-Jordan 法、Gauss の消去法、LU 分解のそれぞれの計算スピード、精度について比較する。

3.3.1 Gauss-Jordan 法 (部分ピボット選択なし)

ここでは Gauss-Jordan の方法について説明する。
まず 0 行目の先頭の係数を 1 にするように変換する。

$$
\begin{aligned}
1x_0 + a'_{0.1}x_1 + \cdots + a'_{0.n-1}x_{n-1} &= b'_0 \\
a_{1.0}x_0 + a_{1.1}x_1 + \cdots + a_{1.n-1}x_{n-1} &= b_1 \\
&\vdots \\
a_{n-1.0}x_0 + a_{n-1.1}x_1 + \cdots + a_{n-1.n-1}x_{n-1} &= b_{n-1}
\end{aligned}
$$

次に、0 行目めの方程式を利用して、それ以外の下の方程式の最初の項を全て 0 にする。0 行目めの方程式を何倍かして、他の方程式から引けば最初の項は 0 になる。

$$
\begin{aligned}
1x_0 + a'_{0.1}x_1 + \cdots + a'_{0.n-1}x_{n-1} &= b'_0 \\
0 + a'_{1.1}x_1 + \cdots + a'_{1.n-1}x_{n-1} &= b'_1 \\
&\vdots \\
0 + a'_{n-1.1}x_1 + \cdots + a'_{n-1.n-1}x_{n-1} &= b'_{n-1}
\end{aligned}
$$

さらに、１行目の方程式を使い同じように他の方程式の２番目の項を全て 0 にする。

$$
\begin{aligned}
x_0 + \ 0 + \cdots + a''_{0.n-1}x_{n-1} &= b''_0 \\
1x_1 + \cdots + a''_{1.n-1}x_{n-1} &= b''_1 \\
&\vdots \\
0 + \cdots + a''_{n-1.n-1}x_{n-1} &= b''_{n-1}
\end{aligned}
$$

これを最後まで繰り返すと下のような式になる。

$$
\begin{aligned}
x_0 + \ 0 \ + \cdots + \ 0 \ &= b'''_0 \\
x_1 + \cdots + \ 0 \ &= b'''_1 \\
&\vdots \\
x_{n-1} &= b'''_{n-1}
\end{aligned}
$$

こうして対角の係数を 1、他の係数を全て 0 にできる。これを消去法といい、上のように完全に消してしまう方法を Gauss-Jordan 法と呼ぶ。この形から未知数 X は右辺の B''' の値と同じとわかり、求まる。

アルゴリズムのプログラムを最後に掲げる。

3.3.2 Gauss-Jordan 法 (部分ピボット選択あり)

上で紹介した 3.3.1 の消去法には欠点が 1 つある。もし 1 にしたい対角要素がゼロの場合、計算ができなくなってしまう。このような消去のために使われて 1 になった係数をピボットと言うが、ピボットが 0 になると計算が実行不能になってしまうことが欠点である。係数が 0 でない方程式と行を入れ換えれば、この問題を解決することができる。同じことは 0 行目以降でも起こる可能性がある。

そこで、計算のたびにピボットとして絶対値が最大の係数を持つ行を、まだ消去に使っていない行から探してピボット行を入れ換える。これをピボット選択と呼ぶ。係数が 0 でなくても以下の理由から部分ピボットを使うことは有効である。ここで行列 A の対角要素 (ピボット) を $a_{kk}^{k-1}(a_{11}^{(0)} = a_{11})$ とする。

- 仮に $a_{kk}^{k-1} \neq 0$ だったとしても絶対値が小さければ、$m_i^{(k)}$ を計算するときにオーバーフローを引き起こす恐れがある。そうでなくても小さい数での割り算をしていくと、一般的に精度は悪くなるから。

- $|a_{kk}^{k-1}|$ の値が小さくなったのは、それ以前の段階で桁落ちが発生したためかもしれないから。

- 分母の絶対値である $|a_{kk}^{k-1}|$ をできるだけ大きく選ぶほうが、$a_{ij}^{(k-1)}$ の情報を活かせる。

アルゴリズムのプログラムを最後に掲げる。

3.3.3 行交換 Gauss の消去法

Gauss-Jordan 法では消去で問題を解いたが、消去範囲を下側の方程式のみにかぎれば下のような連立方程式を得ることができる。これを前進消去という。

$$
\begin{aligned}
a_{0.0}x_0 + a_{0.1}x_1 + \cdots + a_{0.n-2}x_{n-2} + a_{0.n-1}x_{n-1} &= b_0 \\
a^{'}_{1.1}x_1 + \cdots + a^{'}_{1.n-2}x_{n-2} + a^{'}_{1.n-1}x_{n-1} &= b^{'}_1 \\
&\vdots \\
a^{''}_{n-2.n-2}x_{n-2} + a^{''}_{n-2.n-1}x_{n-1} &= b^{''}_{n-2} \\
a^{'''}_{n-1.n-1}x_{n-1} &= b^{'''}_{n-1}
\end{aligned}
$$

この連立方程式の最後の式から $x_{n-1} = b^{'''}/a^{'''}_{n-1.n-1}$ より x_{n-1} の解を求めることができる。この答えを上の連立方程式の下の方程式から代入していくと

$$
\begin{aligned}
a_{0.0}x_0 + a_{0.1}x_1 + \cdots + a_{0.n-2}x_{n-2} &= b_0 - a_{0.n-1}b^{'''}/a^{'''}_{n-1.n-1} \\
a^{'}_{1.1}x_1 + \cdots + a^{'}_{1.n-2}x_{n-2} + &= b^{'}_1 - a^{'}_{1.n-1}b^{'''}/a^{'''}_{n-1.n-1} \\
&\vdots \\
a^{''}_{n-2.n-2}x_{n-2} &= b^{''}_{n-2} - a^{''}_{n-2.n-1}b^{'''}/a^{'''}_{n-1.n-1} \\
x_{n-1} &= b^{'''}/a^{'''}_{n-1.n-1}
\end{aligned}
$$

となり、次に x_{n-2} を求めることができる。これを上の式に繰り返し行うことで全部の解を導くことができる。この過程を後退代入という。Gauss の消去法は前進消去とこの後退代入を組み合わせて解く方法である。この方法は Gauss-Jordan 法に比べて計算量が少なく済む。Gauss の消去法は最速の計算法といわれており、n が大きい場合の計算量は n^3 に比例するといわれる。

アルゴリズムのプログラムを最後に掲げる。

3.3.4 LU 分解

Gauss の消去法と同程度の速度で計算でき、かつもっと便利に使えるのが LU 分解である。LU 分解は、L行列とU行列の分解がベクトルbに依存しない。係数行列が同じならば、どんな連立一次方程式を解く場合においても LU 分解までは共通して計算できる。つまり同じ係数行列を使って何度も連立一次方程式を解く場合、すなわち右辺の b の値だけをいろいろ変化させて解かせたい場合、最初に LU 分解をしておけば後は $O(n^2)$ で解を導くことができる。これが Gauss の消去法と後退代入を組み合わせた方法や Gauss-Jordan 法に比べて、際立って優れている点である。同じ係数行列を使って何度も連立一次方程式を解くということは世の中に多々あり、例えば逆行列を求めるときである。

LU 分解とは下の様に正方行列を左下三角行列（L）と右上三角行列（U）に分解することで、この分解ができれば Gauss の後退代入と同様の計算で X が求められる。

例)

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ l_{1,0} & 1 & 0 \\ l_{2,0} & l_{2,1} & 1 \end{pmatrix} \times \begin{pmatrix} u_{0,0} & u_{0,1} & u_{0,2} \\ 0 & u_{1,1} & u_{1,2} \\ 0 & 0 & u_{2,2} \end{pmatrix}$$

3.4　むすび

本章では、連立一次方程式を解く直接法について、部分ピボット選択をしない Gauss-Jordan 法、部分ピボット選択をする Gauss-Jordan 法、行交換 Gauss の消去法、LU 分解、それぞれにおける特徴を述べた。

次章では結果について述べる。

第 4 章

実行結果

4.1 シミュレーション

行列 A の大きさを変化させた場合のその他の値の変化の推移

行列 A の大きさによる計算速度や誤差、メモリの変化がどのように推移するかを調べる。行列の大きさ n は、それぞれ 10,100,500,1000,2000,3000,4000 について調べた。

解法による比較

Gauss-Jordan 法 (部分ピボットなし)、Gauss-Jordan 法 (部分ピボットあり)、行交換 Gauss の消去法、LU 分解、それぞれにおいて計算速度、計算の誤差を比較を行った。

2 種類のコンピュータの性能による比較

性能の違う 2 台のコンピュータそれぞれでシミュレーションすることでどのような違いがあるかを調べた。

使用したコンピュータ

1 台目

NEC VALUSTAR C VC500/1D

CPU:Intel Pentium4 1.50GHz

MEMORY:512MB(SDRAM,PC133)

2 台目

SONY VAIONOTE PCG-FR55G/B

CPU:Mobile Intel Celeron Processer 2.0GHz

MEMORY:256MB(DDR SDRAM)

4.2 出力結果

NEC VALUSTAR C VC500/1D の場合

Gauss-Jordan 法 (部分ピボットなし)

行列	時間 [t]	誤差
10	0.000025	5.961081e-15
100	0.021330	1.162934e-12
500	2.884000	2.024504e-11
1000	22.421000	4.700162e-10
2000	178.047000	1.800523e-9
3000	591.902000	3.076151e-9
4000	—	—

n	10	100	500	1000	2000	3000	4000
メモリ消費量 (k)	884	1.764	8.736	24.512	79.516	141.776	251.376

—：計測できなかったもの、または値が不安定なもの

Gauss-Jordan 法 (部分ピボットあり)

行列	時間 [t]	誤差
10	0.000035	2.787301e-15
100	0.022330	1.213984e-13
500	2.904000	2.978030e-13
1000	22.122000	8.006687e-12
2000	179.689000	2.416873e-11
3000	598.382000	1.100924e-11
4000	—	—

n	10	100	500	1000	2000	3000	4000
メモリ消費量 (k)	888	1.772	8.744	24.520	79.524	141.784	251.380

行交換 Gauss の消去法

行列	時間 [t]	誤差
10	0.000021	2.602757e-15
100	0.008452	1.022751e-13
500	1.008500	2.413118e-13
1000	7.601000	7.463647e-12
2000	61.618000	1.619603e-11
3000	195.932000	6.629584e-12
4000	462.825000	1.064810e-11

n	10	100	500	1000	2000	3000	4000
メモリ消費量 (k)	884	1.772	8.760	24.560	79.584	165.880	283.444

LU 分解

行列	時間 [t]	誤差
10	0.000036	3.098895e-15
100	0.035451	1.266317e-13
500	1.191000	2.158943e-13
1000	10.405000	6.848941e-12
2000	138.319000	1.423527e-11
3000	913.767000	8.562336e-12
4000	—	—

n	10	100	500	1000	2000	3000	4000
メモリ消費量 (k)	892	1.780	8.704	24.500	79.324	165.308	251.424

SONY VAIONOTE PCG-FR55G/B の場合

Gauss-Jordan 法 (部分ピボットなし)

行列	時間 [t]	誤差
10	0.000018	5.961081e-15
100	0.016130	1.162934e-12
500	2.543000	2.024504e-11
1000	18.697000	4.700162e-10
2000	145.499000	1.800523e-9
3000	480.64000	3.076151e-9
4000	—	—

n	10	100	500	1000	2000	3000	4000
メモリ消費量 (k)	876	1.756	8.728	24.504	79.508	141.056	—

Gauss-Jordan 法 (部分ピボットあり)

行列	時間 [t]	誤差
10	0.000025	2.787301e-15
100	0.016630	1.213984e-13
500	2.554000	2.978030e-13
1000	18.997000	8.006687e-12
2000	146.101000	2.416873e-11
3000	488.482000	1.100924e-11
4000	—	—

n	10	100	500	1000	2000	3000	4000
メモリ消費量 (k)	880	1.764	8.736	24.512	79.516	141.052	—

行交換 Gauss の消去法

行列	時間 [t]	誤差
10	0.000015	2.602757e-15
100	0.006320	1.022751e-13
500	0.912200	2.413118e-13
1000	6.670000	7.463647e-12
2000	50.523000	1.619603e-11
3000	184.275000	6.629584e-12
4000	—	—

n	10	100	500	1000	2000	3000	4000
メモリ消費量 (k)	876	1.764	8.752	24.552	79.576	—	—

LU 分解

行列	時間 [t]	誤差
10	0.000025	3.098895e-15
100	0.017314	1.266317e-13
500	1.120000	2.158943e-13
1000	12.789000	6.848941e-12
2000	169.003000	1.423527e-11
3000	832.546000	8.562336e-12
4000	—	—

n	10	100	500	1000	2000	3000	4000
メモリ消費量 (k)	900	1.780	8.760	24.540	79.556	141.096	—

4.3 むすび

本章では 2 台の異なるコンピュータで n=10,100,500,1000,2000,3000,4000 における計算速度、誤差、メモリ使用量をそれぞれの解法において調べ、表にした。次章では、実験の考察と結論を述べる。

第 5 章

まとめ

5.1　はじめに

本章では、本論文の結びとしてまとめ、考察を述べる。

5.2　まとめ

連立一次元方程式の問題は現代において様々にわたる。本論文では連立一次方程式を解くことができる様々な解法を適用し比較することで、計算速度や誤差の違い、それぞれの特徴について述べた。

第 2 章では、連立一次元方程式を解く前の準備として、浮動小数点数について述べた。

第 3 章では、連立一次元方程式を解く方法である直接法のうち Gauss-Jordan 法 (部分ピボットなし)、Gauss-Jordan 法 (部分ピボットあり)、Gauss の消去法、LU 分解のアルゴリズムや特徴などを述べた。

第 4 章では、行列の大きさによって、どのように計算速度や誤差、メモリの消費量は推移するのか、それぞれの解法の違いは一体どれほどなのか、コンピュータの性能の違いから何が読み取れるかなどを調べるために実験を行った。

本論文の構成は以上の通りであった。

5.3　考察

5.3.1　行列 A の大きさを変化させた場合の値の変化の推移について

すべての解法において n を大きくすれば、もちろん計算時間は増え、精度は悪くなり、メモリの消費量は増えたが、それぞれの方法ごとに特徴が現れた。5.3.2 でそれを考察する。

5.3.2 解法による比較について

まずGauss-Jordan法の部分ピボットなしとありを見比べると、ありのほうが計算速度が遅く、逆に精度は良いことが分かる。これは、’あり’のほうは部分ピボット選択により行を入れ替えている分だけ’なし’より時間がかかるが、そのおかげで精度が良くなっていることを表している。よって、この部分の実験は成功であるといえるだろう。また’あり’のほうがメモリの消費量が多かった。これは部分ピボット選択の分だけメモリの消費量が多いと考えられる。

行交換Gaussの消去法は上の二つに比べ、抜群のパフォーマンスを示した。メモリの消費量こそ他の二つに比べ多かったが計算速度はかなり速く、メモリが十分にあれば他の3つの方法では1時間以上たってもできなかったn=4000の行列の計算を確認することができた。またこの方法はnが大きい場合の計算量はn^3に比例するが、実験の結果からもそれを読み取れることができる。よって、この部分の実験は成功であるといえるだろう。メモリ消費量が多かったのは後退代入の分であろうと考えられるが明確ではない。

結果をみて首をひねったのはLU分解である。理論的にはGaussの消去法と同等の計算量で計算できると思っていたが結果は全く異なっていた。n=500付近ではだいたい同等であったがnが大きくなるほどGaussの消去法との速度の差が広がっていくという結果となった。原因としてはLU分解のアルゴリズムでピボット選択の結果として行を入れ代えるために、この対応関係を配列に記憶するということをしているがこの部分がnが大きくなるほど負担になっているのではと思う。またはLとUに分解する行為が、nが大きいほど計算の負担になるのかもしれないが、要因がいろいろありこれがそうだというものはよく分からなかった。ひとつ気になったのはメモリの消費量でn=500より前の段階でメモリの消費量が逆転しているということである。これがなぜなのかいまいちよくわからなかったが、

メモリの消費量がだいたい同じくらいになる n の大きさで計算速度が同じになりやすい結果となった。メモリも原因のひとつに絡んでいるのかもしれない。

5.3.3　2種類のコンピュータの性能による比較について

2台目のほうが1台目より、すべての方法において計算速度が速かった。1台目はPentium4とはいえ約2年前のものであるから仕方ないが、最近のコンピュータの性能の良さを改めて認識させてもらった結果だった。

またどのくらいまでのnの大きさを処理できるかはメモリの量であることが結果から分かる。512MBを積んだコンピュータがn=4000のGauss消去を解いたのに対し256MBのほうはメモリ不足で解くことができなかった。もし512MB積んでいたら、より高速に解くのは容易に想像できる。ちなみに256MBのほうはnの大きさが3000で限界だったが512MBのほうは5000くらいまでは警告は出るが強引にやろうとすればできた。

今回はCPUもメモリも異なる2つのコンピュータによる比較だったためCPU、メモリそれぞれの性能の違いを明確に理解することができなく残念である。今度は同じCPU、同じメモリで比較してみたい。

5.4　結論

連立一次方程式を解く直接法で今回の4つの方法うちGaussの消去法の優秀性を示すことができた。このときnが大きいほど良い結果が出ることが分かった。しかしながら、実際の計算では100万元、1000万元、さらには1億元といった超大次元の連立一次方程式を解くことが求められる。このような方程式に対してはGaussの消去法はメモリ不足であるが、幸い、方程式の係数行列は零要素が多いため、共役勾配法の系統でメモリを節約できる反復法が用いられている。大次元方程式に対する反復法とGauss消去法との比較は今後の課題としたい。

謝辞

本研究を進めるに当たり、終始丁寧な御指導及び御激励を賜り、その他多くの面でも色々とご面倒を見て下さりご助言を与えてくださいました山本哲朗教授に深く感謝いたします。

また新しくできた研究室で先輩がいない中、大石研究室からわざわざ来ていただき多々に渡りアドバイスをしてくださったり、かつ日常生活の様々においてお力になってくださいました、山本研究室修士課程2年阿口誠司氏、小笹耕平氏に深く感謝いたします。

また同じ山本研究室4年として様々な相談に乗ってくださったり、意見交換をしてくださったりといろいろ協力していただいた嶋田陽介氏、藤田祐作氏、井上陽介氏に深く感謝します

参考文献

[1] Gene H.Golub and Charles F. Van Loan: Matrix Computations, third edition, The Johns Hopkins University Press, Baltimore and London (1996).

[2] 山本哲朗:数値解析入門 [増訂版], サイエンス社 (2003).

[3] 大石進一:数値計算, 裳華房 (1999)

Appendix A

付録

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


/*関数の宣言*/
void   Gyoretu(void);
double Gosa(void);
void   Keisoku(long N,char *s,void (*func)(void));
int    main(void);
void   GaussJordan1(void);
void   GaussJordan2(void);
void   Gauss(void);
void   LU(void);



const int MAX=3000;/*扱える次元の最大次元*/
int       dim=MAX;
double    a[MAX][MAX];
double    x[MAX];
double    w[MAX][MAX+1];
double    b[MAX];



/*A と x を決めて、そこから B を作る*/
void Gyoretu(){

    int i,j;
    srand(10);

        for(i=0;i<dim;i++){
            x[i]=1;
        }
        for(i=0;i<dim;i++){
           for(j=0;j<dim;j++){
                a[i][j]=(rand()-RAND_MAX)/10000.0;
           }
        }
        for(i=0;i<dim;i++){
             b[i]=0;
                for(j=0;j<dim;j++){
                        w[i][j]=a[i][j];
                        b[i]+=a[i][j]*x[j];
                }
                  w[i][dim]=b[i];
        }
}
```

```
/*Gauss-Jordan 法 (部分ピボット選択なし)*/
void GaussJordan1(void ){

    int i,j,k;
    double p;
    for(i=0;i<dim;i++){
        p=w[i][i];/*ピボット*/
        for(k=0;k<dim+1;k++){
                w[i][k]/=p;
        }
        for(j=0;j<dim;j++){
            if(i!=j){
                p=w[j][i];
                for(k=0;k<dim+1;k++){
                   w[j][k]-=p*w[i][k];
                }
            }
        }
    }
    for(i=0;i<dim;i++){
          b[i]=w[i][dim];
    }
}



/*Gauss-Jordan 法 (部分ピボット選択あり)*/
void GaussJordan2(void){

    int i,j,k,MAX;
    double p,tmp;

/*ピボット選択 */
    for(i=0;i<dim;i++){

        MAX=i;
        for(j=i+1;j<dim;j++){
            if( fabs(w[j][i]) > fabs(w[MAX][i]) )
                MAX=j;
        }
        if(i!=MAX){
                for(k=i;k<dim+1;k++){
                 tmp=w[i][k];
                 w[i][k]=w[MAX][k];
                 w[MAX][k]=tmp;
             }
        }

        p=w[i][i];
        for(k=0;k<dim+1;k++){
                w[i][k]/=p;
        }
        for(j=0;j<dim;j++){
            if(i!=j){
                p=w[j][i];
                for(k=0;k<dim+1;k++){
```

```
                    w[j][k]-=p*w[i][k];
                }
            }
        }
    }

    for(i=0;i<dim;i++){
        b[i]=w[i][dim];
    }
}



/*行交換 Gauss の消去法*/
void Gauss(void){

    int i,j,k,MAX;
    double p,tmp;

    /*前進消去*/
    for(i=0;i<dim-1;i++){
        MAX=i;
        for(j=i+1;j<dim;j++){
            if( fabs(w[j][i]) > fabs(w[MAX][i]) )
                MAX=j;
        }
        if(i!=MAX){
            for(k=i;k<dim+1;k++){
               tmp=w[i][k];
               w[i][k]=w[MAX][k];
               w[MAX][k]=tmp;}
        }
        for(j=i+1;j<dim;j++){
            p=w[j][i]/w[i][i];
            for(k=i;k<dim+1;k++){
                w[j][k]-=p*w[i][k];
            }
        }
    }

    /*後退代入*/
    for(i=dim-1;i>=0;i--){
        p=w[i][dim]/w[i][i];
        for(j=0;j<i;j++) {
            w[j][dim]-=p*w[j][i];
        }
        w[i][dim]=p;
    }
    for(i=0;i<dim;i++){
          b[i]=w[i][dim];
    }

}
```

```
/* LU分解*/

double y[MAX];
int index[MAX];

/*L とU に分解する*/
void lu1(void){

    int i,MAX,j,k,itmp;
    double big,dum,sum,tmp;

    for (i=0;i<dim;i++) {
        index[i]=i;
        big=0;
        for (j=0;j<dim;j++)
            if ((tmp=fabs(w[i][j])) > big) big=tmp;
            y[i]=1.0/big;
        }
     for (j=0;j<dim;j++) {
            for (i=0;i<j;i++) {
                sum=w[i][j];
                for (k=0;k<i;k++) sum -= w[i][k]*w[k][j];
                w[i][j]=sum;
            }
            big=0;
            for (i=j;i<dim;i++) {
                sum=w[i][j];
                for (k=0;k<j;k++) sum -=w[i][k]*w[k][j];
                w[i][j]=sum;
                if ( (dum=y[i]*fabs(sum)) >= big) {
                       big=dum;MAX=i;
                }
            }
            if (j != MAX) {
            for(k=0;k<dim;k++){
                tmp=w[j][k];
                w[j][k]=w[MAX][k];
                w[MAX][k]=tmp;
            }
            itmp=index[j];
            index[j]=index[MAX];
            index[MAX]=itmp;

            tmp=y[MAX];
            y[MAX]=y[j];
            y[j]=tmp;
        }
        if (j != dim) {
            for (i=j+1;i<dim;i++)
                  w[i][j] /= w[j][j];
        }
    }
}


/*X の計算*/
```

```
void lu2(void){

    int i,j;
    double sum;

    for(i=0;i<dim;i++)
          b[i]=w[index[i]][dim];
    for (i=0;i<dim;i++){
        sum=b[i];
        for (j=0;j<i;j++) sum -= w[i][j]*b[j];
            b[i]=sum;
        }
        for(i=dim-1;i>=0;i--){
            sum=b[i];
            for(j=i+1;j<dim;j++) sum -=w[i][j]*b[j];
            b[i]=sum/w[i][i];
      }
    return;
}


void LU(void)
{
        lu1();
        lu2();
}



/* 誤差を測る*/
double Gosa(void)
{
    int i;
    double rr;
    rr=0;
    for(i=0;i<dim;i++){rr+=(b[i]-x[i])*(b[i]-x[i]);}
    return sqrt(rr/dim);
}


/* 計算速度を計測する*/
void Keisoku(long N,char *s,void (*func)(void))
{
    long t0,t1,t2,t3;
    int i;
    double r;
    r=0;
    t0=clock();

    for(i=0;i<N;i++){
          Gyoretu();
          r+=Gosa();
    }
    t1=clock();

    r=0;
```

```
    for(i=0;i<N;i++){
          Gyoretu();
          func();
          r+=Gosa();
    }
    t2=clock();

    t3=(t2-t1)-(t1-t0);

    if(t3>1000) {
           printf("%s: 次元=%3d  計算速度=%f[s] 誤差=%10e ¥n",s,dim, t3/1000.0/N,
      }else {
            Keisoku(10*N,s,func);
     }
}


/*計る次元と解法を指定して実行する*/
int main(void)
{
   dim=1000;
    Keisoku(1,"LU  ",LU);


    return 0;
}
```