ガウス・ザイデル法レポート

平成 27 年 10 月 27 日
未来ロボティクス学科
B2 1426015
今井 良佑

1 目的

今回のレポートを通して、ガウス・ザイデル法の理論を学び、それをプログラムに書きガウス・ザイデル法の特性などについて学び理解し考察する。

2 理論

今回学ぶガウス・ザイデル法の理論は、連立方程式（1）が与えられたとき、

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \end{cases} \tag{1}$$

x_1, x_2, x_3 に適当な初期値を与え以下の式 (2) にしたがって x_1, x_2, x_3 を更新していく。

$$\begin{cases} x_1 = \frac{b_1 - a_{12}x_2 + a_{13}x_3}{a_{11}} \\ x_2 = \frac{b_2 - a_{22}x_2 + a_{23}x_3}{a_{21}} \\ x_3 = \frac{b_3 - a_{32}x_2 + a_{33}x_3}{a_{31}} \end{cases} \tag{2}$$

すると、与えられた連立方程式が式 (3) を満たす場合 x_1, x_2, x_3 は収束し解となる。

$$|a_{ii}| > \sum_{j=1, j \neq i}^{n} a_{ij} \tag{3}$$

なお、(3) の条件は収束されるための十分条件であり、必要条件ではない。

3 実験内容

今回私が注目したのは一つ目は収束条件である。そこで、可視化しやすいように二元一次連立方程式で収束する時としないときをそれぞれグラフに描画してそれぞれでどのような違いがあるのかを考察する。
もう一つは収束速度に注目し、初期値と次元をそれぞれ変化させて様子を考察する。

4　実験方法

まず、今回の実験での収束というのは前回の計算の値との差の絶対値がすべて 0.00000001 以内となった時とする。

4.1　収束について

二元一次連立方程式の拡大行列に当たる 2 × 3 の行列で式 (3) を満たす場合ものとその行を入れ替え条件を満たさないものとをそれぞれグラフに描画し比較する。
また行を入れ替えても式 (3) を満たさないものを使用し同様に考察する。

4.2　解の収束速度について

2〜5 次と 10 次の拡大行列で初期値を変化させ収束までの計算回数を比較する。

5　環境・使用機器

この実験に使用した機器は学科推奨パソコン (Panasonic Let's note CF-SX3) で、グラフを描画することを考慮して、使用言語はグラフ描画経験のある Python3.4 とした。作成したソースコードは以下のソースコード 1 の通りである。
また推奨されていた C 言語でのソースコードも作成したため今回の考察では使わなかったがこのレポートの末尾に追記にソースコード 3 として記載する。C 言語の開発環境は Visual Studio Express 2013 for Windows Desktop を使用した。

ソースコード 1: GaussSeidelMethod.py

```
# -*- coding: utf-8 -*-
#-------------------------------------------------------------------------------
# Name:   GaussSeidelMethod.py
# Author:   R.Imai
# Created:   2015/10/07
# Last Date:   2015/10/11
# Note: ガウス・ザイデル法
# コマンドライン引数は,Matrixcsv ファイル,結果書き込みcsv ファイル,保存するファイル名 (拡張子はなし)
#-------------------------------------------------------------------------------

import sys
import csv
import numpy as np
import matplotlib.pyplot as plt
import codecs
from math import *

argv = sys.argv
argNum = len(sys.argv)

u"""
    収束判定する誤差
"""
ACCURACY = 0.00000001

U"""
    importData
```

```
    コマンドライン引数で指定される csv ファイルの内容を連立方程式の拡大行列として読みこみ、その行数とともに返す

    mat: csv ファイルを取り込む配列
    cnt: mat の行数のカウンタ
"""
def importData():
    try:
        fp1 = open(argv[1], 'r')
        reader = csv.reader(fp1)
    except IOError:
        print (argv[1]+"cannot be opened.")
        exit()
    except Exception as e:
        print('type' + str(type(e)))
        exit()

    mat=[[]]
    cnt = 0
    for row in reader:
        if cnt == 0:
            mat[0] = [int(elm) for elm in row]
        else:
            mat.append([int(elm) for elm in row])
        cnt += 1

    return cnt,mat

u"""
    transPos
    行を入れ替えることにより収束するようになるかを判断する

    index: その列が最大の値になっている行の場所を示した配列 (ない場合は-1)
"""
def transPos(index):
    check = True

    for i in range(0,dimension):
        if index[i] == -1:
            check = False

    return check

u"""
    matrixSet
    収束する可能性が高いように行を入れ替える
    transPos()がFalse の場合はそのまま

    newMat: 入れ替え後の配列
    colIndex: その列が最大になる行の数 (ない場合は-1)

"""
def matrixSet(dimension,mat):
    newMat = [[]]
    colIndex = [-1] * dimension
    maxColList = [0] * dimension

    for i in range(0,dimension):
        maxCol = 0
        for j in range(1,dimension):
            if mat[i][maxCol] < mat[i][j]:
                maxCol = j
        maxColList[i] = maxCol
        colIndex[maxCol] = i

    if transPos(colIndex):
        for i in range(0,dimension):
```

```
            if i == 0:
                newMat[i] = mat[colIndex[i]]
            else:
                newMat.append(mat[colIndex[i]])

        return newMat

    else:

        return mat

u"""
    possibility
    収束の十分条件を満たしているかの判断
"""
def possibility(dimension,mat):
    check = True
    for i in range(0,dimension):
        cnt = 0
        for j in range(0,dimension):
            if i != j:
                cnt = cnt + mat[i][j]
        if mat[i][i] < cnt:
            check = False

    return check

u"""
    deformation
    mat を計算するための係数に移項する
"""
def deformation(mat):
    newMat = np.zeros([dimension,dimension])
    for i in range(0,dimension):
        k = 0
        for j in range(0,dimension+1):
            if i != j:
                newMat[i][k] = mat[i][j]/mat[i][i]
                k += 1
    print (newMat)

    return newMat

u"""
    check
    収束したかの判断
"""
def check(coe):
    check = False
    for i in range(0,dimension):
        if abs(coe[coe.shape[0]-1][i] - coe[coe.shape[0]-2][i]) > ACCURACY:
            check = True

    return check

u"""
    firstCalc
    一回目の解の更新 (変数宣言等があるため)
"""
def firstCalc(mat,coe):
    newCoe = np.zeros(dimension)
    for i in range(dimension):
        newCoe[i] = coe[i]
    for i in range(0,dimension):
        k = 0
        newCoe[i] = mat[i][dimension-1]
        for j in range(0,dimension):
```

```
            if(i != j):
                newCoe[i] = newCoe[i] - mat[i][k] * newCoe[j]
                k += 1

    return newCoe

u"""
    calc
    計算で解を更新
"""
def calc(mat,coe):
    newCoe = np.zeros(dimension)
    for i in range(dimension):
        newCoe[i] = coe[coe.shape[0]-1][i]
    for i in range(0,dimension):
        k = 0
        newCoe[i] = mat[i][dimension-1]
        for j in range(0,dimension):
            if(i != j):
                newCoe[i] = newCoe[i] - mat[i][k] * newCoe[j]
                k += 1

    return newCoe

u"""
    popCol
    配列coe の col 列目を返す
"""
def popCol(coe,col):
    line = coe[0][col]
    for i in range(1,coe.shape[0]):
        line = np.vstack([line,coe[i][col]])
    return line

u"""
    plot
    グラフ化する
"""
def graphPlot(coe):
    plt.figure(1)
    plt.subplot(111)
    for i in range(dimension):
        plt.plot(popCol(coe,i), label = "X" + str(i+1))
    plt.legend()
    #plt.xlim([0,12]) #x 軸の端を指定したい際に使用
    #plt.ylim([-20,100]) #y 軸の端を指定したい際に使用
    plt.show() その場で見たい時に使用
    #plt.savefig(argv[2]+".png") #保存をしたい際に使用


u"""
    writeCSV
    解の変化の流れをCSV ファイルに保存
"""
def writeCSV(coe):
    try:
        fp2 = open(argv[2]+".csv", 'w',newline='')
        csvWriter = csv.writer(fp2)
        for data in coe:
            csvWriter.writerow(data)
        fp2.close()
    except IOError:
        print ("cannot make file.")
        exit()
    except Exception as e:
        print('type' + str(type(e)))
        exit()
```

```
u"""
    dimension: 次元
    mat: 拡大行列
    coe: 解の流れを記録する二次元配列
"""
if __name__ == '__main__':
    dimension,mat=importData()
    print(str(dimension))
    coe = np.zeros(dimension) #初期値が全部ゼロならこっちを使用
    #coe = np.array([1,2,3,4]]) #初期値を指定するならこっち (必要に応じてコマンドライン引数使用)次元に注意
    mat = matrixSet(dimension,mat)
    if possibility(dimension,mat):
        mat = deformation(mat)
        print(coe.shape[0]-1)
        coe = np.vstack([coe,firstCalc(mat,coe)])
        while check(coe):
            coe = np.vstack([coe,calc(mat,coe)])
        print("the answers are "+ str(coe[coe.shape[0]-1]))
        writeCSV(coe)
        graphPlot(coe)

    else:
        print("this case can not calc.")
```

このソースコードの工夫点としてコマンドラインの引数から拡大行列の書かれた csv ファイルを指定することで何行でも対応できるようになっているところである。
さらに必要に応じて保存、初期値指定等、コマンドライン引数を使用することで汎用性を高くした。
さらに、matrixSet() 関数を実装することにより与えられた拡大行列の行の順番を並べ替えることで収束条件を満たす可能性が高くなるようにした

6　結果

6.1　収束について

二次元のグラフの描画にはソースコード 2 の関数を使用した

ソースコード 2: plot2D

```
def Plot2D(coe):
    dim,eq = importData()

    x1 = [-1.0]
    y1 = [(1/eq[0][1])*(eq[0][2]-eq[0][0]*(-1))]
    for i in range(-9,11):
        x1.append(i*0.1)
        y1.append((1/eq[0][1])*(eq[0][2]-eq[0][0]*i*0.1))

    x2 = [-1.0]
    y2 = [(1/eq[1][1])*(eq[1][2]-eq[1][0]*(-1))]
    for i in range(-9,11):
        x2.append(i*0.1)
        y2.append((1/eq[1][1])*(eq[1][2]-eq[1][0]*i*0.1))

    plt.figure(1)
    plt.subplot(111)
```

```
    plt.plot(x1,y1,label = "eqation1")
    plt.plot(x2,y2,label = "eqation2")
    plt.plot(popCol(coe,0), popCol(coe,1),label = "GaussSeidelResult")
    plt.plot(0,0,"ro")
    #plt.xlim([-20,10]) #x 軸の端を指定したい際に使用
    #plt.ylim([-12,40]) #y 軸の端を指定したい際に使用
    plt.legend()
    plt.show()
    #plt.savefig("Fig.png") #保存したい際に使用
```

実験に使用した連立方程式の拡大行列は式 (4) で、収束場合はソースコード 1 をそのまま使い、収束しない場合の時は matrixSet(dimension,mat) をコメントアウトして行の入れ替えを無効にした。

$$\tilde{A} = \begin{bmatrix} -6 & 10 & 10 \\ 10 & 5 & 5 \end{bmatrix} \tag{4}$$

初期値は (0 , 0) とした。

　収束するグラフは図 1、収束しないグラフは 2 のようになった。

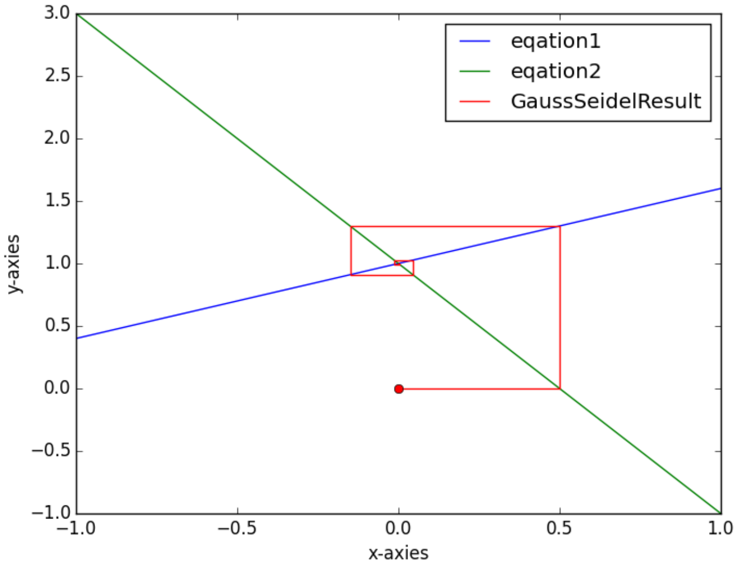

図 1: $\tilde{A}$ の行を入れ替え収束条件を満たすときの値の挙動

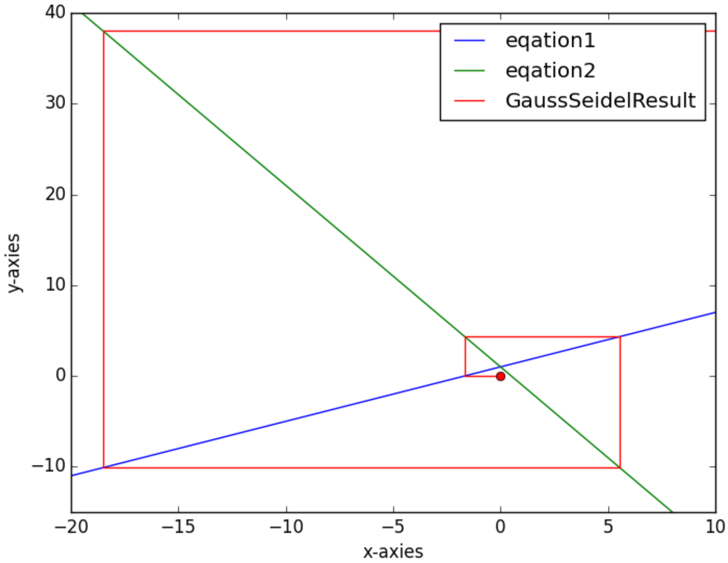

図 2: $\tilde{A}$ を使用し収束しない時の値の挙動

また、拡大行列の行を入れ替えても式 (3) を満たさない行列 $\tilde{B}$ を使用して行を入れ替えて二種類のグラフを出力した。結果は以下の図 3,4 通り

$$\tilde{B} = \begin{bmatrix} -10 & 4 & 6 \\ 6 & -3 & 9 \end{bmatrix} \tag{5}$$

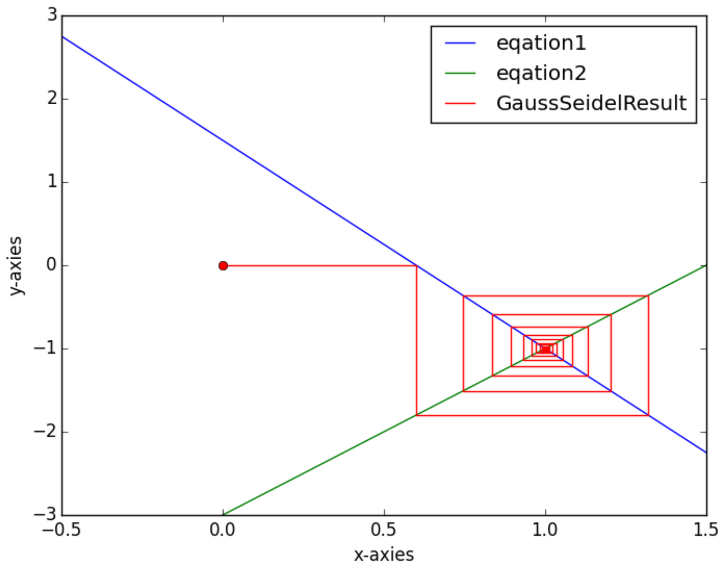

図 3: $\tilde{B}$ の時の挙動

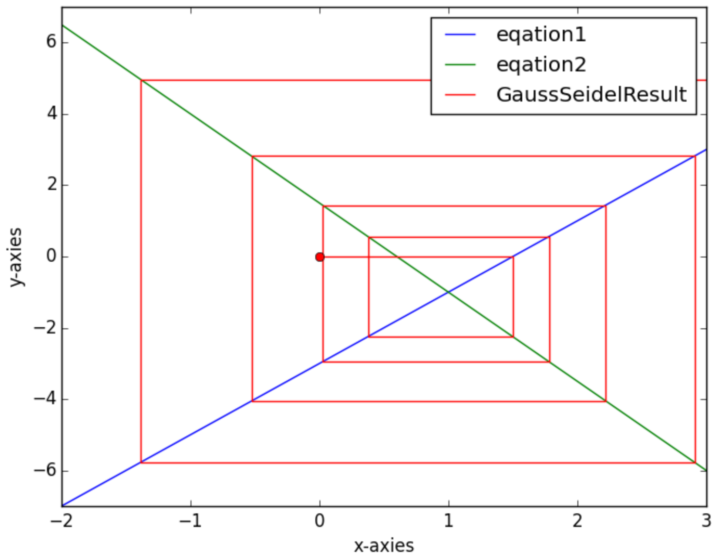

図 4: $\tilde{B}$ の行を入れ替えた時の値の挙動

これにより条件式 (3) を満たさない場合でも収束することがわかる。

さらにここまでの実験結果から二次の場合なら行を入れ替えれば収束するのではないかと仮定をたて様々な場合で試していると以下の式 (6)、図 5 の場合収束も発散もせずに同じ

ところを回り続けることが分かった。

$$\tilde{C} = \begin{bmatrix} 6 & -3 & 9 \\ 6 & 3 & 3 \end{bmatrix} \tag{6}$$

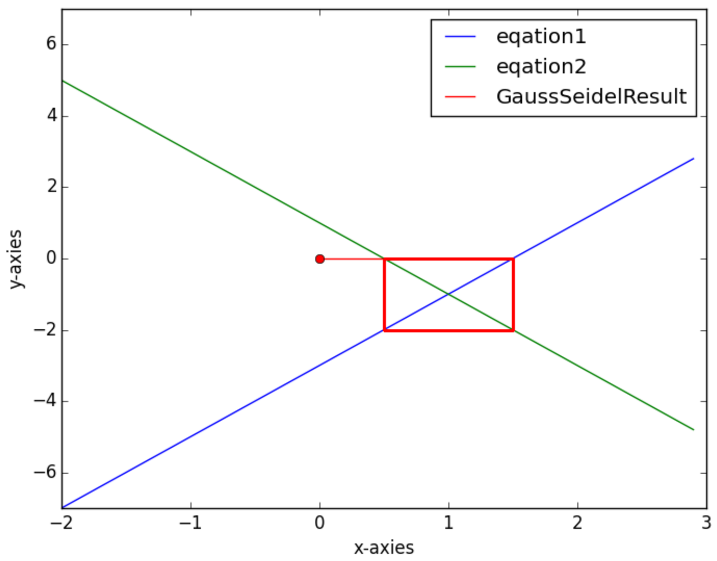

図 5: 収束も発散もしない場合

この時の解の変化の様子を描画すると図 6 のようになった。
なお、永遠に続いてしまうため 30 回計算したところで終了させた。

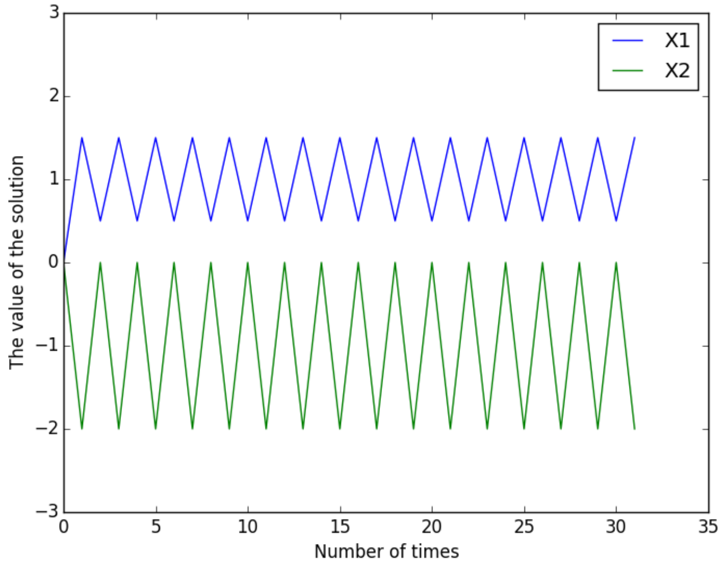

図 6: 解の変化の様子

図 6 より解が振動していることがわかる。

6.2 解の収束速度について

6.2.1 二次の場合

二次の場合に使用した行列は (7) で、その解は $x_1 = 2, x_2 = -1$ である。

$$\begin{bmatrix} 9 & 5 & 13 \\ 1 & 6 & -4 \end{bmatrix} \tag{7}$$

初期値は (0 , 0) , (-1 , -1) , (2 , 2) , (-1 , 2) と $(10n, 10n)_{(n \text{ は } 1 \leq n \leq 10 \text{ を満たす整数})}$ の 14 種類用意し左から順に、$(10n, 10n)$ は n が小さい順に 1,2,3,$\cdots$,14 とラベルを付ける。解の変化の様子をグラフにすると、図 7 のように収束と判断するまでの回数は表 1 となる。

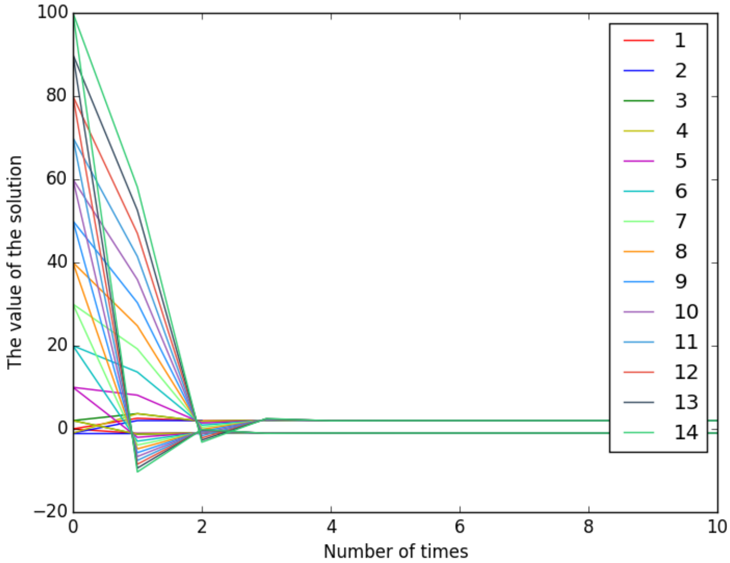

図 7: 二次の場合の初期値を変化させた際の解の挙動

表 1: 二次の場合の収束までの回数

初期値	収束までの回数
(0 , 0)	10
(-1 , -1)	2
(2 , 2)	10
(-1 , 2)	10
(10 , 10)	11
(20 , 20)	11
(30 , 30)	11
(40 , 40)	12
(50 , 50)	12
(60 , 60)	12
(70 , 70)	12
(80 , 80)	12
(90 , 90)	12
(100 , 100)	12

6.2.2 三次の場合

三次の場合に使用した行列は (8) で、その解は $x_1 = 4, x_2 = -1, x_3 = -4, x_4 = 1$ である。

$$\begin{bmatrix} 10 & 4 & 2 & 28 \\ 2 & 9 & 4 & 3 \\ 3 & 4 & 8 & -11 \end{bmatrix} \tag{8}$$

初期値は (0 , 0 , 0) , (1, -3, 3) , (-3 , 3, 1) , (-3 , 1, 3) と $(10n, 10n, 10n)_{(n\text{ は }1\leq n\leq 10\text{ を満たす整数})}$ の 14 種類用意し左から順に、$(10n, 10n)$ は n が小さい順に 1,2,3,⋯,14 とラベルを付ける。解の変化の様子をグラフにすると、図 8 のように収束と判断するまでの回数は表 2 となる。

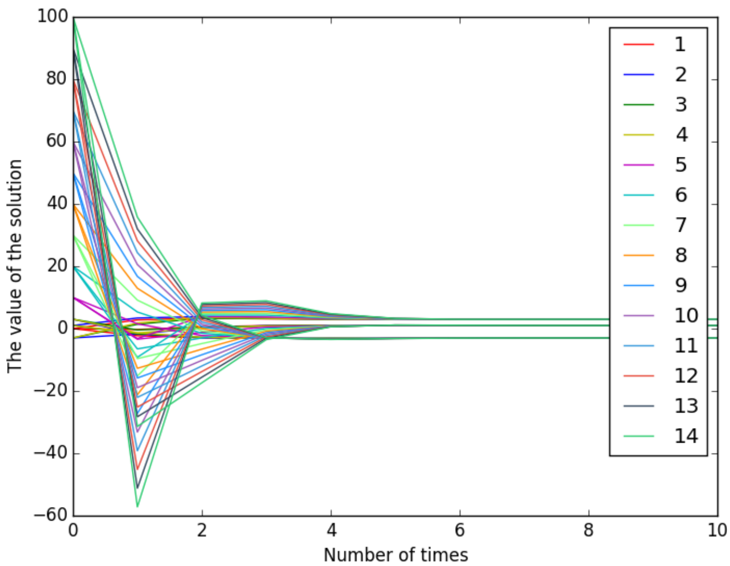

図 8: 三次の場合の初期値を変化させた際の解の挙動

表 2: 三次の場合の収束までの回数

初期値	収束までの回数
(0 , 0 , 0)	17
(1, -3, 3)	17
(-3 , 3, 1)	17
(-3 , 1, 3)	17
(10 , 10 , 10)	18
(20 , 20 , 20)	18
(30 , 30 , 30)	18
(40 , 40 , 40)	18
(50 , 50 , 50)	19
(60 , 60 , 60)	19
(70 , 70 , 70)	19
(80 , 80 , 80)	19
(90 , 90 , 90)	19
(100 , 100 , 100)	19

6.2.3 四次の場合

四次の場合に使用した行列は (9) で、その解は $x_1 = 3, x_2 = 1, x_3 = -3$ である。

$$\begin{bmatrix} 15 & 4 & 2 & 3 & 51 \\ 3 & 13 & 4 & 1 & -16 \\ 6 & 2 & 10 & 1 & -17 \\ 1 & 3 & 2 & 9 & 2 \end{bmatrix} \tag{9}$$

初期値は $(10n, 10n, 10n, 10n)_{(n\text{ は }0\leq n\leq 10\text{ を満たす整数})}$ の 11 種類用意し「ラベル = n」とする。解の変化の様子をグラフにすると、図 9 のように収束と判断するまでの回数は表 3 となる。

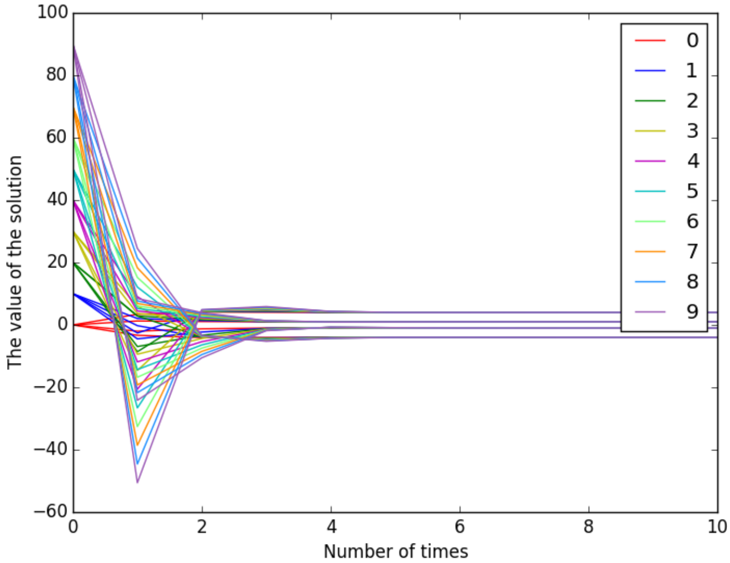

図 9: 四次の場合の初期値を変化させた際の解の挙動

表 3: 四次の場合の収束までの回数

初期値	収束までの回数
(0 , 0 , 0 , 0)	15
(10 , 10 , 10 , 10)	16
(20 , 20 , 20 , 20)	16
(30 , 30 , 30 , 30)	17
(40 , 40 , 40 , 40)	17
(50 , 50 , 50 , 50)	17
(60 , 60 , 60 , 60)	17
(70 , 70 , 70 , 70)	17
(80 , 80 , 80 , 80)	17
(90 , 90 , 90 , 90)	17
(100 , 100 , 100 , 100)	17

6.2.4 五次の場合

五次の場合に使用した行列は (10) で、その解は $x_1 = 10, x_2 = -5, x_3 = 3, x_4 = -10, x_5 = 1$ である。

$$\begin{bmatrix} 19 & 4 & 6 & 2 & 3 & 171 \\ 2 & 20 & 4 & 3 & 2 & -96 \\ 5 & 1 & 22 & 1 & 3 & 104 \\ 1 & 4 & 2 & 10 & 1 & -103 \\ 3 & 4 & 2 & 5 & 15 & -19 \end{bmatrix} \tag{10}$$

初期値は $(10n, 10n, 10n, 10n, 10n)_{(n \text{ は } 0 \leq n \leq 10 \text{ を満たす整数})}$ の 11 種類用意し「ラベル = n」とする。解の変化の様子をグラフにすると、図 10 のように収束と判断するまでの回数は表 4 となる。

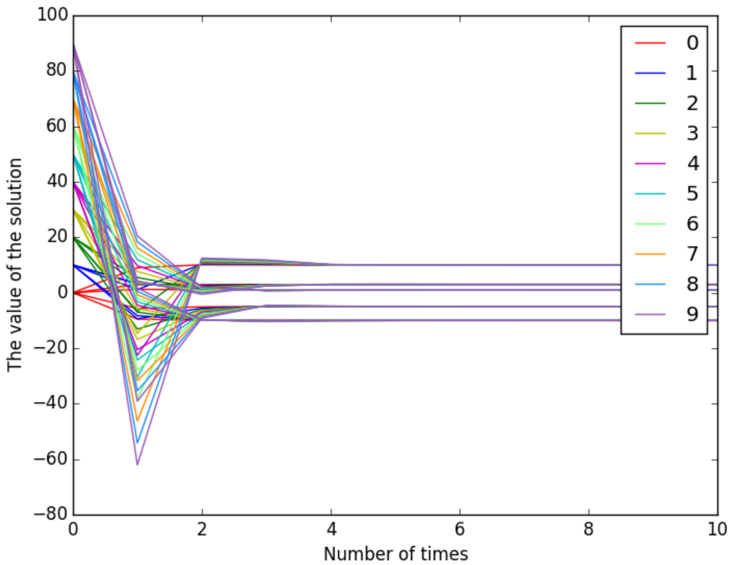

図 10: 五次の場合の初期値を変化させた際の解の挙動

表 4: 五次の場合の収束までの回数

初期値	収束までの回数
(0 , 0 , 0 , 0 , 0)	12
(10 , 10 , 10 , 10 , 10)	14
(20 , 20 , 20 , 20 , 20)	14
(30 , 30 , 30 , 30 , 30)	14
(40 , 40 , 40 , 40 , 40)	14
(50 , 50 , 50 , 50 , 50)	14
(60 , 60 , 60 , 60 , 60)	14
(70 , 70 , 70 , 70 , 70)	14
(80 , 80 , 80 , 80 , 80)	15
(90 , 90 , 90 , 90 , 90)	15
(100 , 100 , 100 , 100 , 100)	15

6.2.5　十次の場合

十次の場合に使用した行列は (11) で、その解は $x_1 = 1, x_2 = -1, x_3 = 5, x_4 = -5, x_5 = 10, x_6 = -10, x_7 = 20, x_8 = -20, x_9 = 15, x_{10} = -15$ である。

$$\begin{bmatrix} 30 & 3 & 4 & 3 & 1 & 1 & 3 & 0 & 2 & 1 & 107 \\ 1 & 30 & 2 & 4 & 1 & 2 & 0 & 1 & 2 & 3 & -84 \\ 2 & 1 & 25 & 1 & 2 & 0 & 2 & 1 & 0 & 2 & 131 \\ 4 & 2 & 6 & 40 & 1 & 3 & 2 & 4 & 5 & 2 & -183 \\ 1 & 3 & 4 & 2 & 45 & 2 & 3 & 1 & 2 & 3 & 463 \\ 1 & 3 & 2 & 3 & 4 & 40 & 1 & 2 & 3 & 1 & -357 \\ 1 & 2 & 3 & 4 & 5 & 6 & 70 & 8 & 9 & 0 & 1359 \\ 2 & 3 & 1 & 4 & 2 & 4 & 3 & 50 & 1 & 2 & -991 \\ 1 & 3 & 4 & 7 & 2 & 5 & 7 & 2 & 90 & 1 & 1388 \\ 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 100 & -1449 \end{bmatrix} \tag{11}$$

初期値は $(10n, 10n, 10n, 10n, 10n, 10n, 10n, 10n, 10n, 10n)_{(n\ \text{は}\ 0 \leq n \leq 10\ \text{を満たす整数})}$ の 11 種類用意し「ラベル = n」とする。解の変化の様子をグラフにすると、図 11 のように収束と判断するまでの回数は表 5 となる。

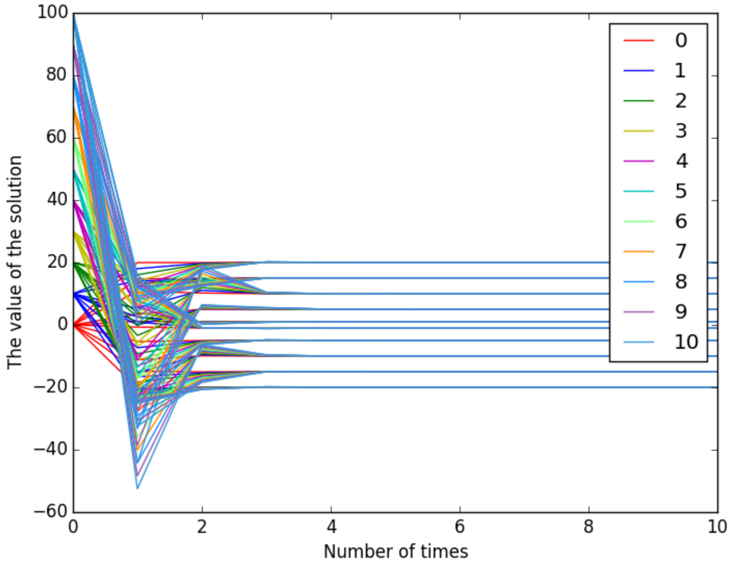

図 11: 十次の場合の初期値を変化させた際の解の挙動

表 5: 十次の場合の収束までの回数

初期値	収束までの回数
(0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0)	11
(10 , 10 , 10 , 10 , 10 , 10 , 10 , 10 , 10 , 10)	12
(20 , 20 , 20 , 20 , 20 , 20 , 20 , 20 , 20 , 20)	12
(30 , 30 , 30 , 30 , 30 , 30 , 30 , 30 , 30 , 30)	12
(40 , 40 , 40 , 40 , 40 , 40 , 40 , 40 , 40 , 40)	12
(50 , 50 , 50 , 50 , 50 , 50 , 50 , 50 , 50 , 50)	12
(60 , 60 , 60 , 60 , 60 , 60 , 60 , 60 , 60 , 60)	12
(70 , 70 , 70 , 70 , 70 , 70 , 70 , 70 , 70 , 70)	12
(80 , 80 , 80 , 80 , 80 , 80 , 80 , 80 , 80 , 80)	12
(90 , 90 , 90 , 90 , 90 , 90 , 90 , 90 , 90 , 90)	12
(100 , 100 , 100 , 100 , 100 , 100 , 100 , 100 , 100 , 100)	12

7 考察

7.1 収束について

この実験でわかったことは、まず $\tilde{B}$ より、やはり収束条件の式は十分条件であり必要条件ではないということである。
また、本実験の出力グラフより二次元の場合はそれぞれの直線上を交互に軸に平行に螺旋

を描くように動いていくということである。
そして、収束するときと発散するときの違いは動いていく向きである。例えば、時計回りに移動しているときに発散していたとすると、反時計回りに移動すれば収束する。つまり解の計算をする順番を入れ替えれば収束するということになる。
また $\tilde{C}$ のように振動してしまうときの条件は二直線の傾きの絶対値が等しい時であると考えられる。
以上より二次元の場合二直線の傾きの絶対値が等しくなければ、行を入れ替えることで必ず収束するのではないかと考える。
また、三次元以降の場合も収束条件を満たさなくてもその直線の絶対値が等しくなければ少なくとも一つは収束する行の組み合わせがあるのではないかと考えた。

7.2 解の収束速度について

これについては解の収束速度は初期値が離れるにつれ増加するのではないかと予想していたが、実際の結果はどの場合においてもそこまで大きな変化はなくさらにグラフにしてみると三、四回目あたりの計算ですでに真値に近づいていることがわかる。
このようになった理由として考えられるのは実験 1 でのグラフをみてわかるように真値から離れているほど移動量が多く修正のかかる量も多いため初期値が離れていても収束にそこまで多くの回数を要さないのではないかと考えた。
また、解と初期値の距離が違うため参考程度になるとは思うが、次元で比べてみてもそこまで収束に要する回数は変わらない。
これについては、次元が上がると一回で計算する量が増えるため収束回数に差が出ないのではないかと考える。

8 追記

今回の考察では使わなかったが、推奨されていた C 言語でのソースコードは以下の Listing3 の通りである。

ソースコード 3: GaussSeidelMethod.c

```
/*------------------------------------
 * Name:   GaussSeidelMethod.c
 * Author:   R.Imai
 * Created: 2015 / 09 / 30
 * Last Date: 2015 / 10 / 08
 * Note: ガウス·ザイデル法
 *
 *------------------------------------*/

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

/**
 * DIMENSION: 連立方程式の次数
 * ACCURACY: 収束判定の誤差
 *
 * mat:  連立方程式の拡大行列
```

```
 * eqation: 移項後の係数群
 * coe:  解
 * wasCoe:  一つ前の解
 * colIndex: 行を入れ替えた際の元の行のインデックス
 */

#define DIMENSION 6
#define ACCURACY 0.000001

double mat[DIMENSION][DIMENSION + 1] = { { 10, 1, 0, 0, 0, 1, 9 }, { 1, 10, 1, 0, 0, 0, 24 }, { 0, 1, 10, 1, 0, 0, 31 }, { 0, 0, 1, 10, 1, 0, -9 }, { 0, 0, 0, 1, 10, 1, -24 }, { 1, 0, 0, 0, 1, 10, -31 } };
double eqaision[DIMENSION][DIMENSION];
double coe[DIMENSION] = { 1, 1, 1, 1, 1, 1 };
double wasCoe[DIMENSION] = { 0, 0, 0,0,0,0 };
int colIndex[DIMENSION];


/**
 * transPos
 * 行を入れ替えて収束する可能性があるかの判断
 *
 */
bool transPos(int *index){
  bool check = true;

  for (int i = 0; i < DIMENSION; i++){
    if (index[i] == -1){
      check = false;
    }
  }
  return check;
}


/**
 * matrixSet
 * 収束する可能性が高いように行入れ替え
 */
void matrixSet(){
  int maxCol = 0;
  int maxColList[DIMENSION];
  double newMat[DIMENSION][DIMENSION + 1];

  for (int i = 0; i < DIMENSION; i++){
    colIndex[i] = -1;
  }
  for (int i = 0; i < DIMENSION; i++){
    maxCol = 0;
    for (int j = 1; j < DIMENSION; j++){
      if (mat[i][maxCol] < mat[i][j]){
        maxCol = j;
      }
    }
    maxColList[i] = maxCol;
    colIndex[maxCol] = i;
  }

  if (transPos(colIndex)){
    for (int i = 0; i < DIMENSION; i++){
      for (int j = 0; j <= DIMENSION; j++){
        newMat[i][j] = mat[colIndex[i]][j];
      }
    }

    for (int i = 0; i < DIMENSION; i++){
      for (int j = 0; j <= DIMENSION; j++){
```

```
        mat[i][j] = newMat[i][j];
      }
    }
  }


}


/**
* possibility
* 収束可能性判断
*/
bool possibility(){
  bool check = true;
  double cnt;

  for (int i = 0; i < DIMENSION; i++){
    cnt = 0;
    for (int j = 0; j < DIMENSION; j++){
      if (i != j){
        cnt += mat[i][j];
      }
    }
    if (mat[i][i] < cnt){
      check = false;
    }
  }
  return check;
}


/**
* deformation
* 移項
*/
void deformation(){
  int k;
  for (int i = 0; i < DIMENSION; i++){
    k = 0;
    for (int j = 0; j < DIMENSION + 1; j++){
      if (i != j){
        eqaision[i][k] = mat[i][j] / mat[i][i];
        k++;
      }
    }
  }
}


/**
* calc
* 実際に計算
*/
void calc(){
  int k = 0;

  for (int i = 0; i < DIMENSION; i++){
    wasCoe[i] = coe[i];
  }
  for (int i = 0; i < DIMENSION; i++){
    k = 0;
    coe[i] = eqaision[i][DIMENSION-1];
    for (int j = 0; j < DIMENSION; j++){
      if (i != j){
        coe[i] = coe[i] - eqaision[i][k] * coe[j];
        k++;
```

```
      }
    }
  }
  for (int i = 0; i < DIMENSION; i++){
    printf("%3.3f   ", coe[colIndex[i]]);
  }
  printf("\n");
}


/**
* check
* 収束判定
*/
bool check(){
  bool cnt = false;
  for (int i = 0; i < DIMENSION; i++){
    if (fabs(coe[i] - wasCoe[i])>ACCURACY){
      cnt = true;
    }
  }
  return cnt;
}


/**
* output
* 結果出力
*/
void output(){
  printf("The answers are ");
  for (int i = 0; i < DIMENSION; i++){
    printf("X%d = %f ", i + 1, coe[colIndex[i]]);
  }
}


void main(){
  matrixSet();
  if (possibility()){
    deformation();
    while (check()){
      calc();
    }
    output();
  }
  else{
    printf("this case can not calc.\n");
  }
}
```