レポート課題 No.2

川口廣伊智
学籍番号:051715223

2017/07/06

目次

0 レポートについての注意

0.1 各課題の解答の構成について

まず課題の解釈をし、解答するためのプログラムを記載した。次に得られた結果を記載した。考察すべき内容があった場合は簡単な考察も付けた。すべての課題に考察がついているわけではない。

0.2 プログラムについて

各課題についてその計算を行うためのプログラムを一部抜粋して記載した。

1 基本課題 EX3-1

1.1 課題概要

LU 分解を用いて行列の行列式を計算するプログラムを作成した。

ソースコード 1 LU 分解を用いて行列の行列式を計算するプログラムの抜粋

```
1 double **a;  // input matrix A
2
3 double det = 1.0; // det of martix A
```

```
4   double sgn = 1.0;  // sign fn
5
6   a = alloc_dmatrix(n, n);
7
8   /* perform LU decomposition */
9   ipiv = alloc_ivector(n);
10  dgetrf_(&n, &n, &a[0][0], &n, &ipiv[0], &info);
11  if (info != 0) {
12    fprintf(stderr, "Error: LAPACK::dgetrf failed\n");
13    exit(1);
14  }
15  printf("Result of LU decomposition:\n");
16  fprint_dmatrix(stdout, n, n, a);
17  printf("Pivot for LU decomposition:\n");
18  fprint_ivector(stdout, n, ipiv);
19
20  /* calculate the determinant of given matrix */
21  for(i = 0;i < n;i++){
22    if((i+1) != ipiv[i]){
23      sgn *= -1.0;  // the eigen value of matrix P
24    }
25  }
26  det *= sgn;
27  for(i = 0;i < n;i++){
28    det *= a[i][i]; // the eigen value of matrix A
29  }
```

見どころは 20 行目から 29 行目の行列式を計算する箇所だ。このように行列式を計算する理由は後の考察で述べる。

このプログラムに別途生成した Vandermonde 行列を入力して、その行列式を計算した。今回は 5 行 5 列で

$$(x_1, x_2, x_3, x_4, x_5) = (1, 2, 3, 4, 5)$$

の Vandermonde 行列を用いた。

以上のプログラムを用いて数値計算した行列式と厳密な値とを比較した。

1.2 結果

数値計算した結果行列式の値は 288 となり、厳密な行列式の値と一致した。

```
Matrix A:
5 5
   1.00000    1.00000    1.00000    1.00000    1.00000
   1.00000    2.00000    3.00000    4.00000    5.00000
   1.00000    4.00000    9.00000   16.00000   25.00000
   1.00000    8.00000   27.00000   64.00000  125.00000
   1.00000   16.00000   81.00000  256.00000  625.00000
Result of LU decomposition:
5 5
   1.00000    1.00000    1.00000    1.00000    1.00000
   1.00000    4.00000    0.50000    0.75000    0.25000
   1.00000   24.00000   -4.00000    0.75000    0.75000
   1.00000  124.00000  -36.00000   -3.00000   -1.00000
   1.00000  624.00000 -232.00000  -39.00000   -6.00000
Pivot for LU decomposition:
5
1 5 3 4 5
determinant is 288.000000
```

1.3 考察 (LU 分解を用いた行列式の計算)

LU 分解すると言っても自分が計算するわけではなく、LAPACK の dgetrf 関数を用いたので、どのように行列が分解されるのかはその関数の説明を読まないことには理解のしようがない。http://www.netlib.org/lapack/explore-html/d3/d6a/dgetrf_8f.html にはこう説明されている。

```
DGETRF computes an LU factorization of a general M-by-N matrix A
using partial pivoting with row interchanges.

The factorization has the form
   A = P * L * U
where P is a permutation matrix, L is lower triangular with unit
diagonal elements (lower trapezoidal if m > n), and U is upper
triangular (upper trapezoidal if m < n).

This is the right-looking Level 3 BLAS version of the algorithm.
```

また行列 A が dgetrf 関数に取り込まれるとどうなるかも説明されている。

```
A is DOUBLE PRECISION array, dimension (LDA,N)
On entry, the M-by-N matrix to be factored.
On exit, the factors L and U from the factorization
A = P*L*U; the unit diagonal elements of L are not stored.
```

つまり、LU 分解したい行列として A を入力すると、行列 A は対角成分より上が分解されてできた上三角行列の成分、対角成分より下が下三角行列の成分、が上書きされて出力される。なお対角成分は上三角行列の成

分で、下三角行列の対角成分は省略されるとのことだ。しかし下三角行列の対角成分は全て 1 なので問題ない。

というわけなので、行列 A は行置換行列 P、下三角行列 L、上三角行列 U に分解される。そして行列がこのように分解されるとき、行列 A の行列式は

$$\det A = (\det P)(\det L)(\det U)$$

と P、L、U それぞれの行列式の積である。

$$\begin{aligned} \det P = (-1)^n \qquad & n \text{ は行置換の回数} \\ \det L &= 1 \\ \det U &= trU \end{aligned}$$

detP は行置換の回数が分かれば求められる。これには dgetrf 関数の出力である IPIV を見ればよい。例えば今回の場合、

$$\mathrm{IPIV} = (1, 5, 3, 4, 5)$$

であったので、途中で 2 行目と 5 行目が入れ替えられていることが分かる。それ以外に行置換はなかったので奇置換であったことが分かる。このように IPIV の成分を見ることで何回行置換が行われたかが分かり、つまり detP を求めることが出来る。

detL は LU 分解すると L の対角成分は全て 1 になるので常に 1 である。

detU を求めるには U のトレースを求める、つまり U の対角成分が分かればよい。dgetrf 関数から出力された行列の対角成分は U の対角成分なので、この対角成分の積を計算したものが detU である。

従って行列 A の行列式は detU に $\mathrm{sgn}(\sigma)$ をかけたものになる。

以上の考察を踏まえて行列式を計算するのが冒頭のプログラムの 20 行目から 29 行目である。

2 基本課題 EX3-2

2.1 実験概要

LU 分解を用いて Dirichlet 境界条件の下での二次元 Laplace 方程式の解を求めるプログラムを作成した。

詳しくは述べないが、Laplace 方程式を差分で表して連立方程式としたときの係数行列など (いわゆる A と b) を別途計算し、その行列とベクトルを次のプログラムに入力した。

ソースコード 2　LU 分解で Laplace 方程式の解を求めるプログラム

```
1  double **a; // input coefficient matrix A
2  double *b; // input vector b
3
4  /* perform LU decomposition */
5  ipiv = alloc_ivector(n);
6  dgetrf_(&n, &n, &a[0][0], &n, &ipiv[0], &info);
7  if (info != 0) {
8    fprintf(stderr, "Error: LAPACK::dgetrf failed\n");
9    exit(1);
10 }
11
12 /* solve equations */
```

```
13  dgetrs_(&trans, &n, &nrhs, &a[0][0], &n, &ipiv[0], &b[0], &n, &info);
14  if (info != 0) {
15    fprintf(stderr, "Error: LAPACK::dgetrs failed\n");
16    exit(1);
17  }
```

このプログラムにより各格子点における u の値が計算される。LAPACK のパッケージを用いて計算するだけなので特に工夫したところはない。

その結果から、いくつかのメッシュ数で解の形と解の計算にかかった時間を記載した。

また解の計算にかかった時間がメッシュ数を増やしていくとどのように変わるかをグラフで分かりやすく示した。

2.2 実験結果

境界条件はどれも課題で与えられた通り

$$u(0,y) = \sin \pi y, u(x,0) = u(x,1) = u(1,y) = 0$$

とした。

各メッシュ数での解の形は次のようになった。

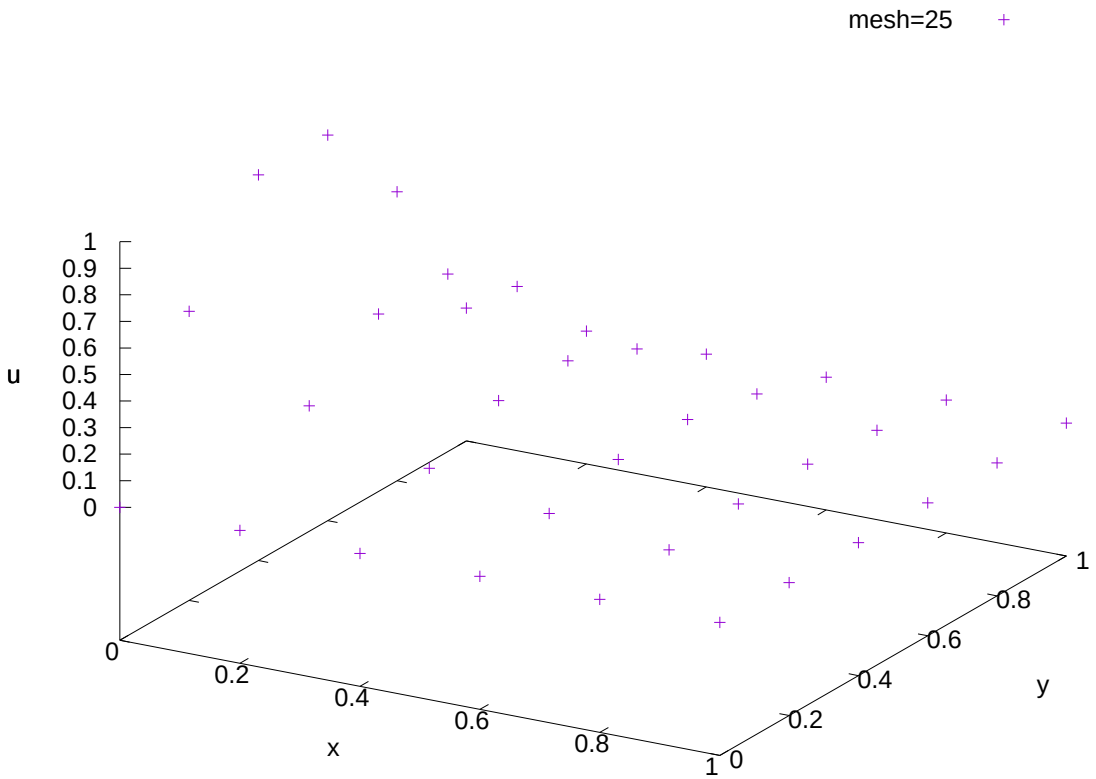

図 1　mesh 数が 25 の解。概形は分かりにくい。

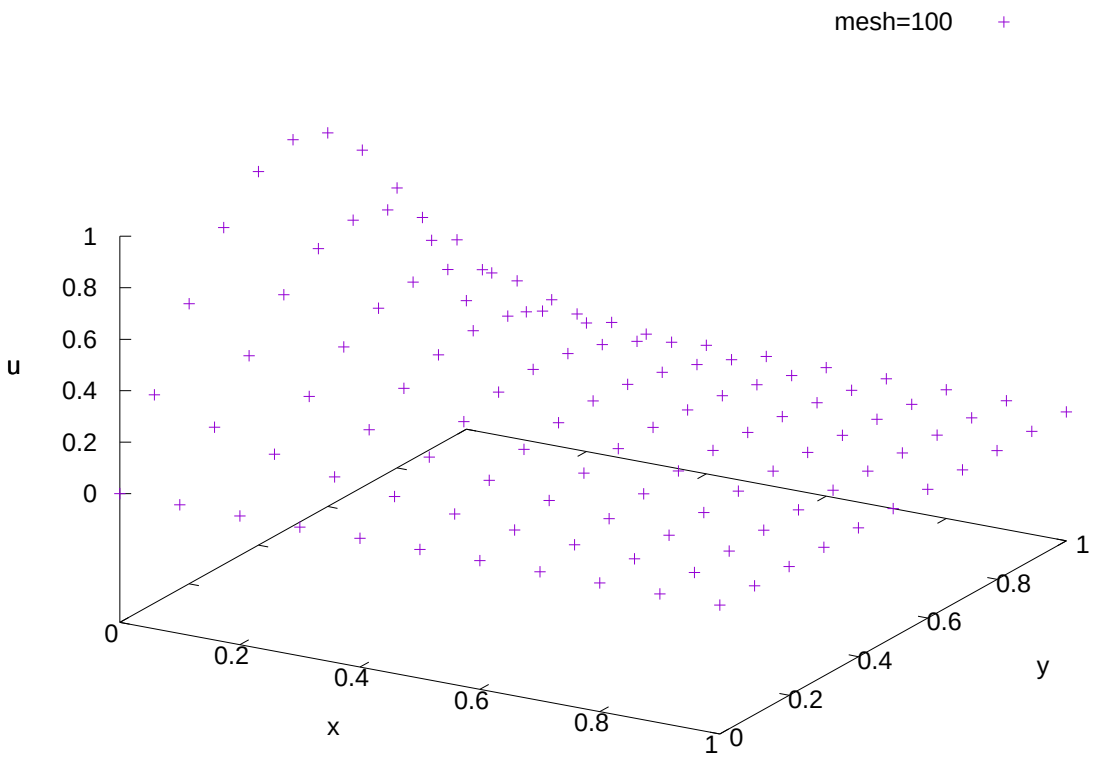

図 2　mesh 数が 100 の解。

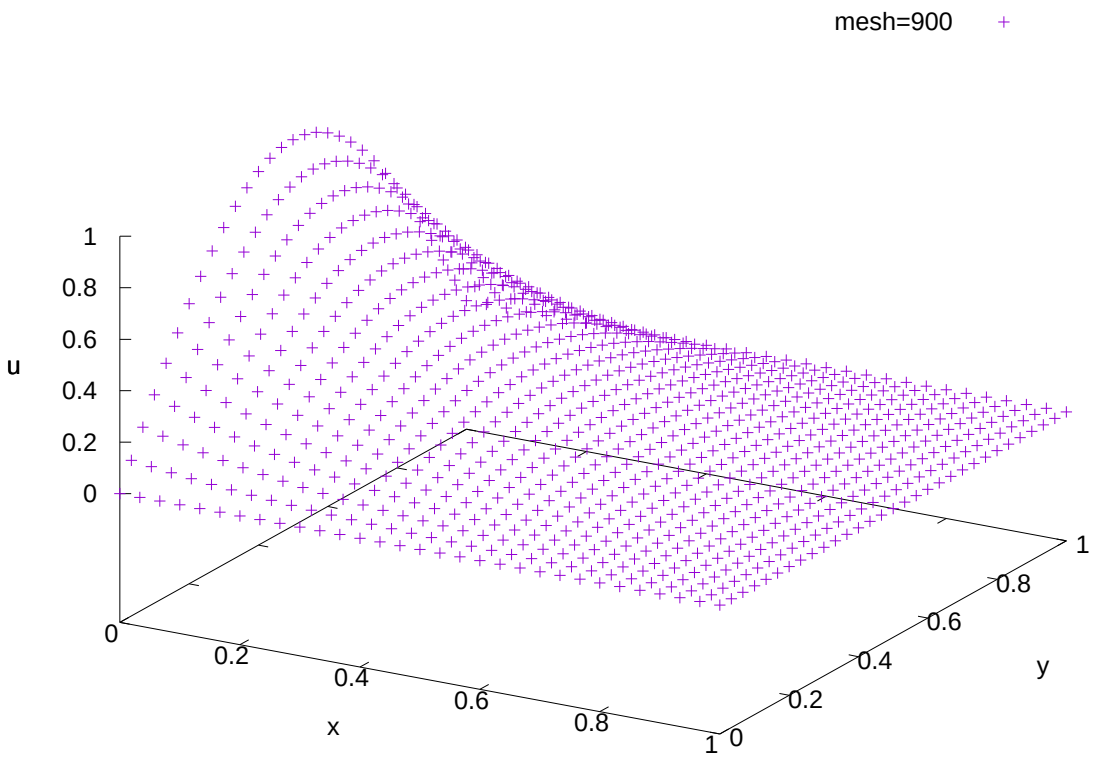

図 3　mesh 数が 900 の解。

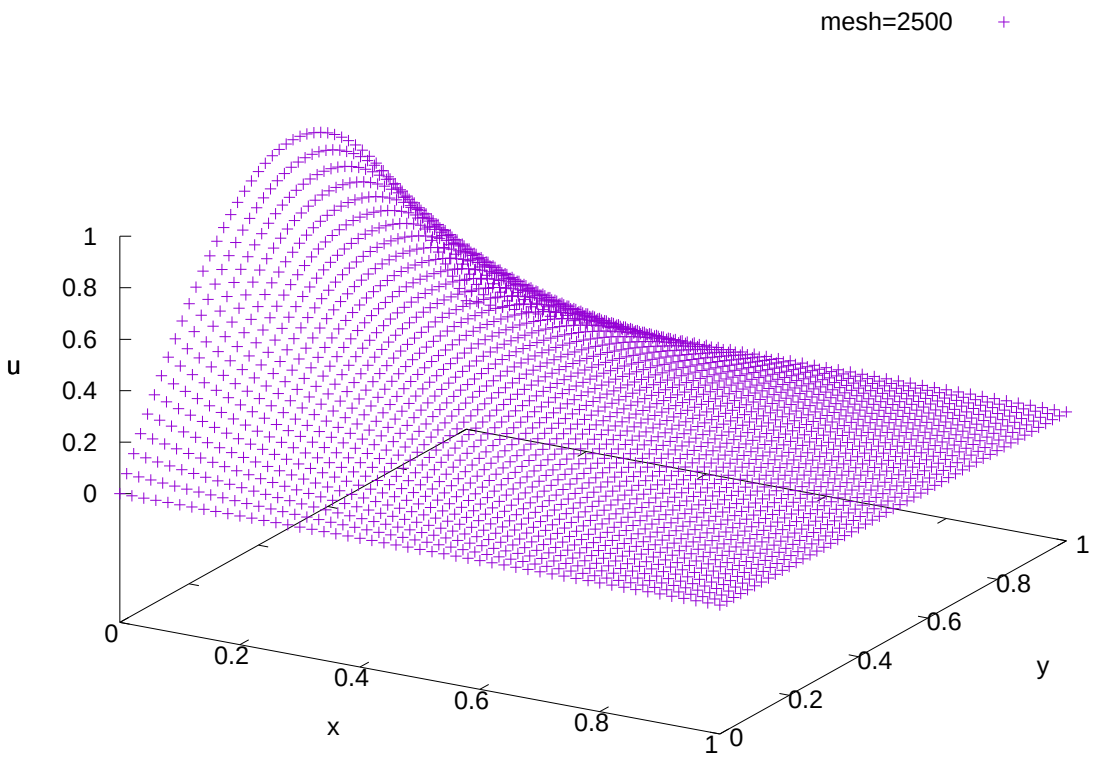

図 4　mesh 数が 2500 の解。

次に計算時間の変化を調べた。計算時間と言ってもどこからどこまでの処理時間をとるかは様々である。ここではプログラムの行列を LU 分解して解を得るところまでを測定した。測定は c 言語の time.h をインクルードして clock() 関数を用いた。

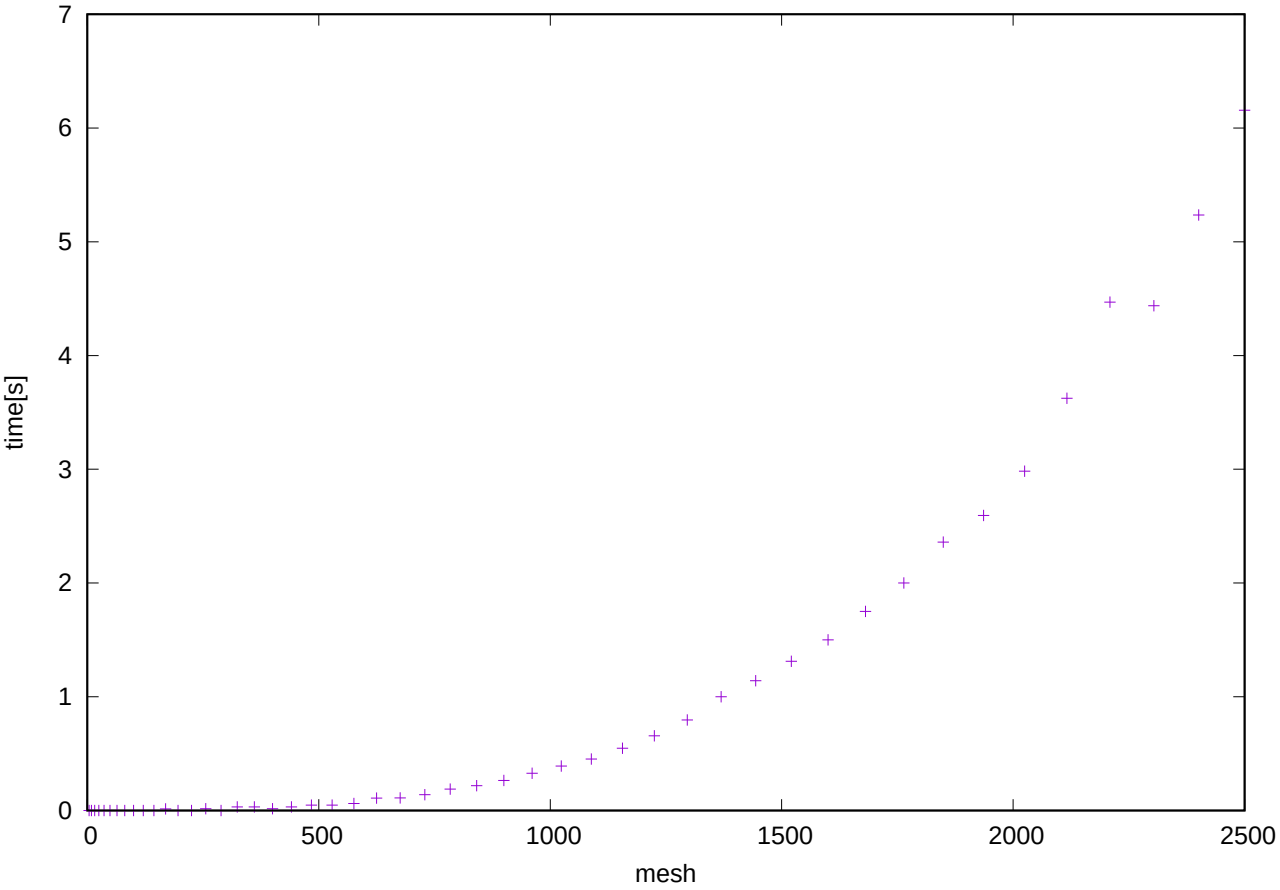

図 5　mesh を増やしたときの計算時間の変化。計算時間はメッシュ数の冪で増えていることが分かる。

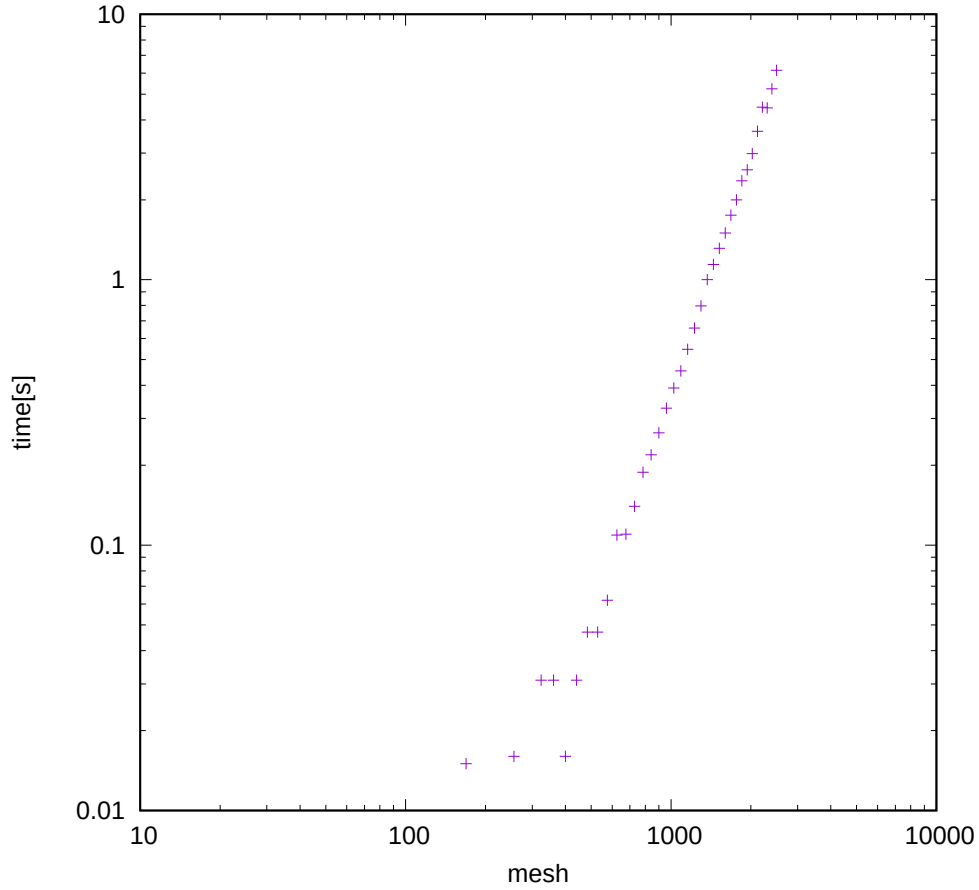

図 6　mesh を増やしたときの計算時間の変化。計算時間はメッシュ数の 3 乗に比例して増えていることが分かる。

2.3　考察 (直接法の計算量)

実験的に連立一次方程式の解を直接法で計算するのは計算量が $O(n^3)$ であることが分かった。
ここではその理論的根拠を示したい。

3　基本課題 EX3-3

3.1　実験概要

Laplace 方程式の境界値問題を jacobi 法で解くプログラムを作成した。

ソースコード 3　jacobi 法で Laplace 方程式を解くプログラム

```
1  double **g; // D^(-1)
2  double **h; // E + F
3  double **c; // D^(-1) * (E + F)
4
5  double *z; // D^(-1) * b
6  double *b; // vector b
7  double *x; // vector x
```

```
8   double *temp;
9
10  do{
11    /* initialize */
12    sum_of_error = 0.0;
13    for(i=0;i<m;i++){
14      y[i] = 0.0;
15    }
16
17    /* multiply matrix c and vector x */
18    for(i=0;i<m;i++){
19      for(j=0;j<m;j++){
20        y[i] += c[i][j] * x[j]; // D^(-1) * (E + F) * x
21      }
22    }
23
24    /* keep current x[i] */
25    for(i=0;i<m;i++){
26      temp[i] = x[i];
27    }
28
29    /* calc next x[i] , -D^(-1) * (E + F)*x + D^(-1) * b */
30    for(i=0;i<m;i++){
31      x[i] = -y[i] + z[i];
32    }
33
34    /* count iteration */
35    l = l + 1;
36
37    /* calc diff of temp[i] and x[i] */
38    for(i=0;i<m;i++){
39      sum_of_error += pow(x[i] - temp[i], 2);
40    }
41  }while((pow(sum_of_error, 0.5) > epsilon) && (l < LMAX)); // test for convergence and
      set a limit in case
```

まず適当なメッシュの数での計算結果から得られた解をプロットした。

次にメッシュ数を増やしていくと計算速度がどう変化するかを調べた。ここでは計算速度は解を求める際の反復回数とした。

3.2 実験結果

まず、解の形は次のようになった。

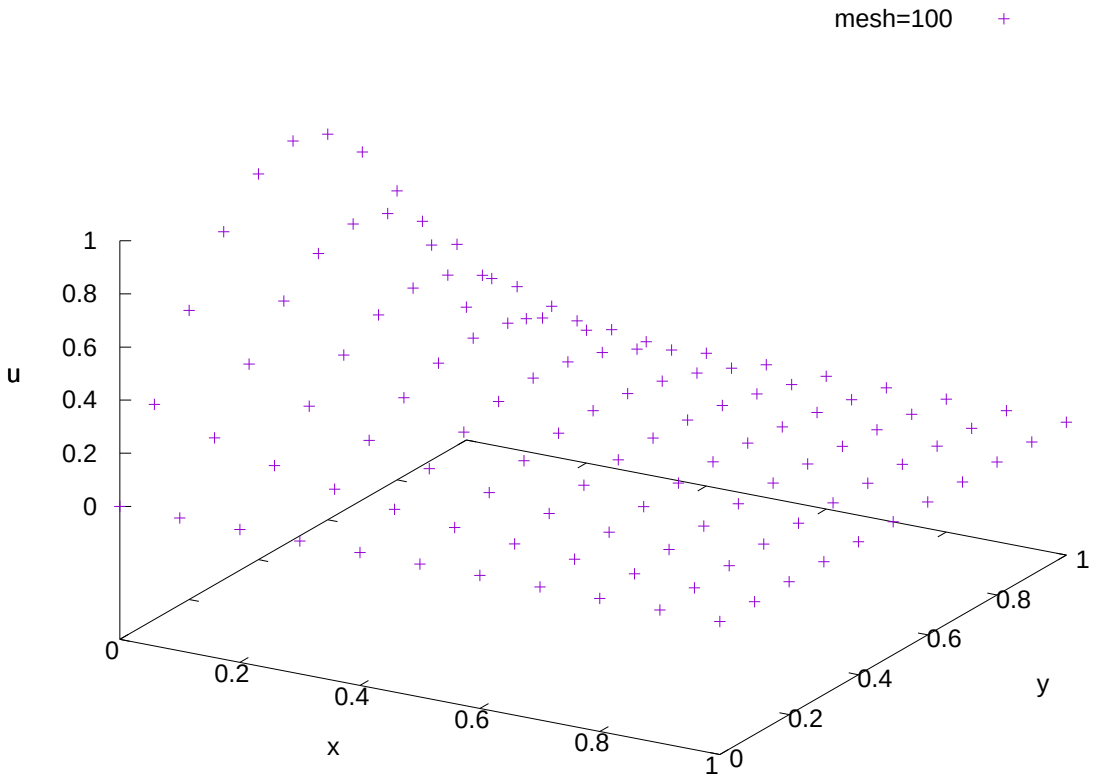

図 7　mesh 数が 100 の解。

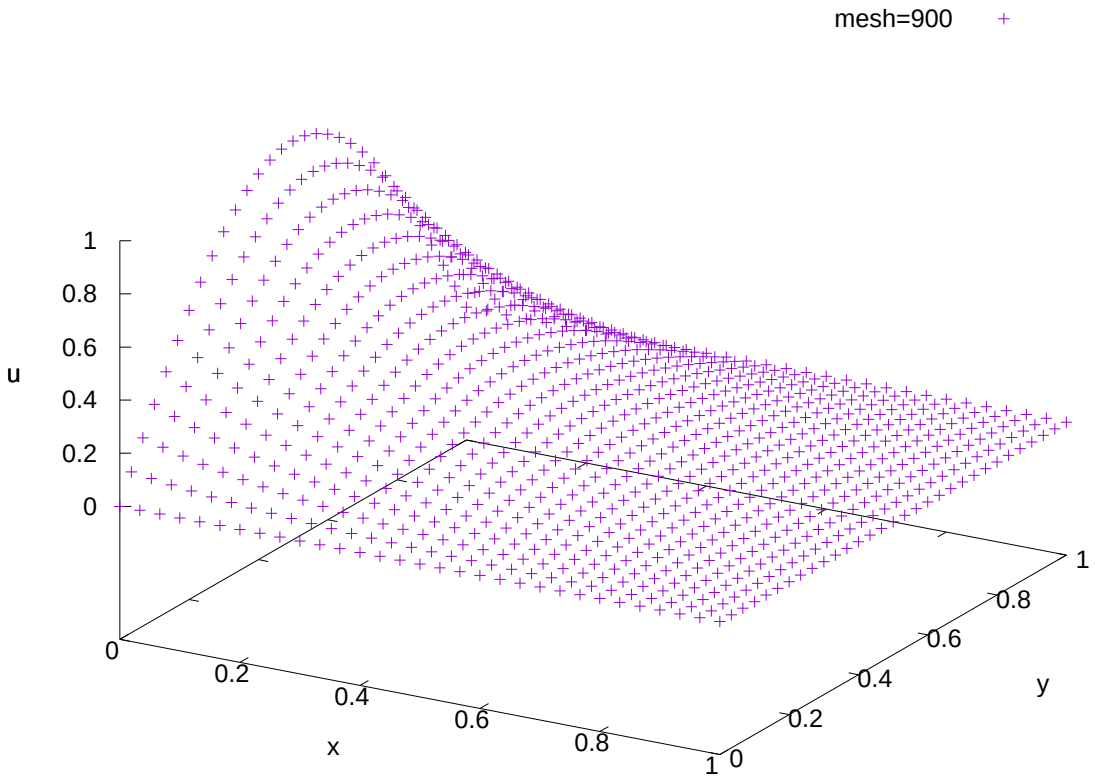

図 8　mesh 数が 900 の解。

次にメッシュ数を増やしたときの反復回数の変化である。

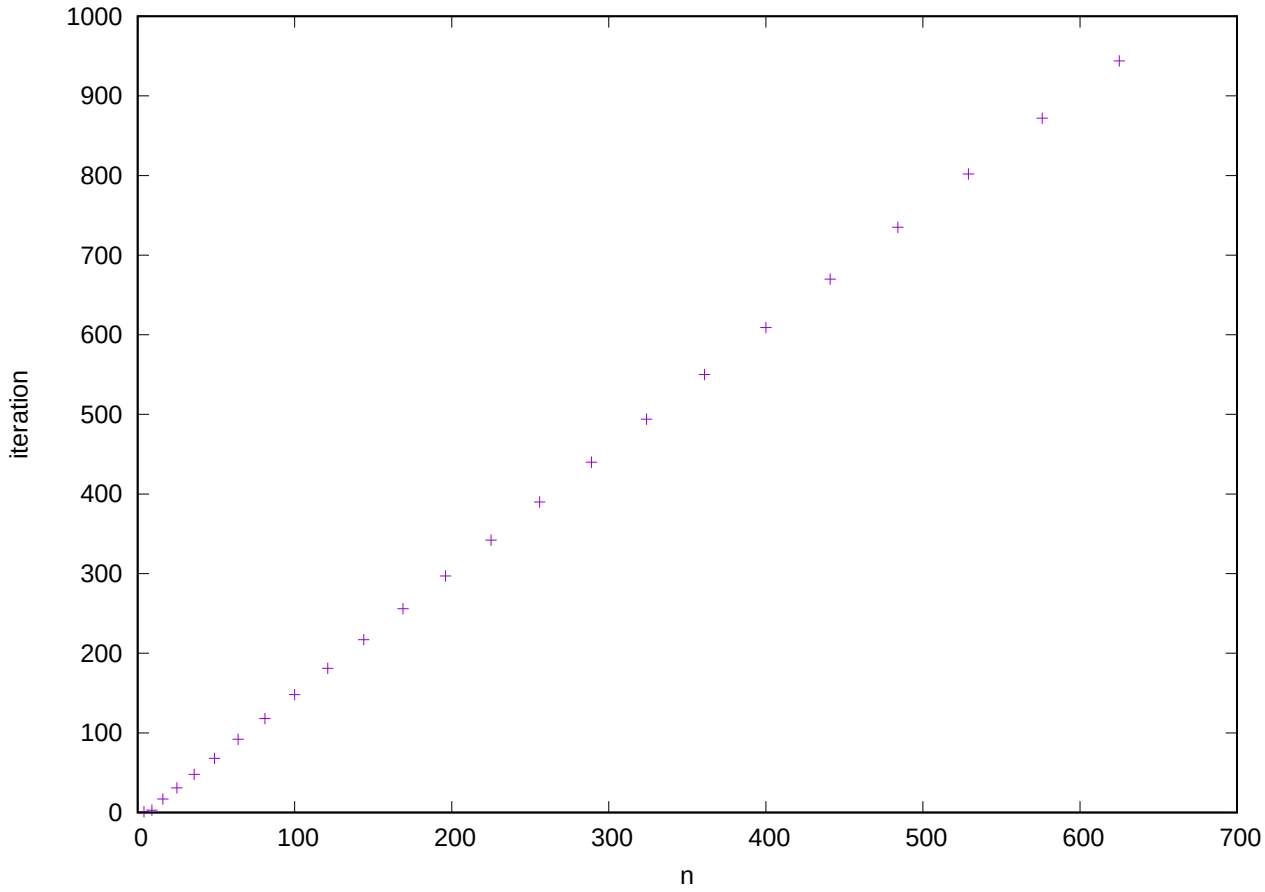

図 9　反復回数の変化。反復回数がメッシュ数に比例していることが分かる。

また解を求めるのにかかる処理時間を C 言語の clock() 関数で計測した。

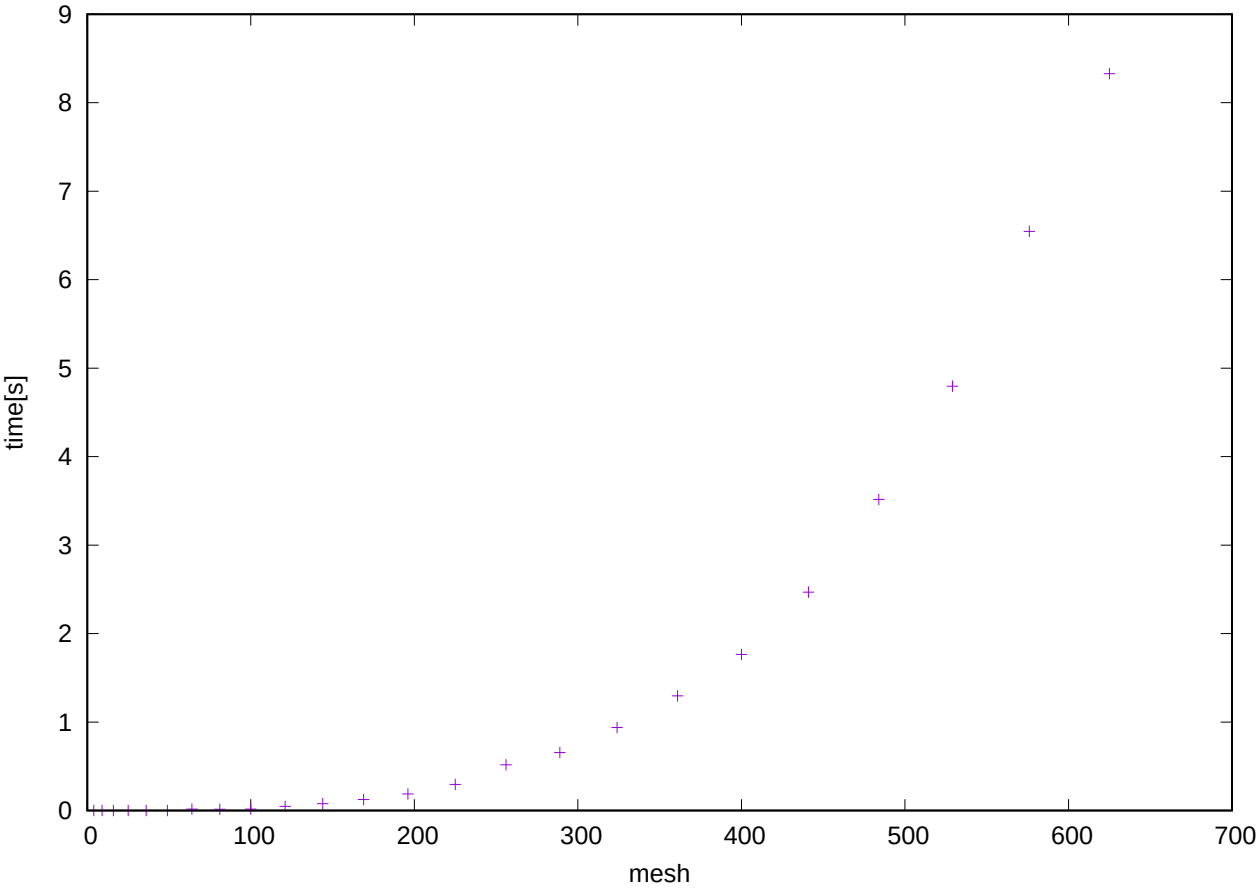

図 10　メッシュ数を増やしていったときの処理時間の変化。

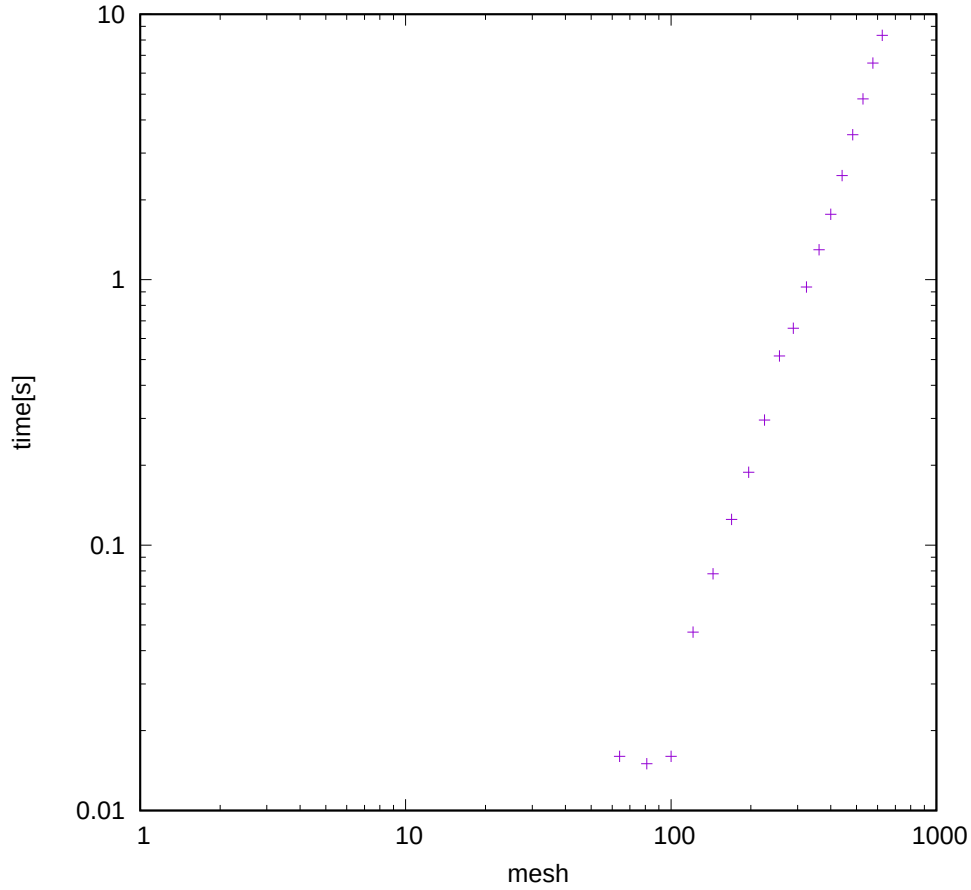

図 11　メッシュ数を増やしていったときの処理時間の変化を両対数プロットした図。処理時間がメッシュ数の 3 乗に比例していることが分かる。

3.3　考察 (反復法の計算量)

4　応用課題 EX3-1

4.1　実験概要

pointer.c のソースコードを見て出力される結果を予想し、実際にコンパイルして得た出力と比較した。まず、ベクトルの方について予想される出力を考え、結果と比較する。行列の方についても同様にする。

4.2　実験結果

pointer.c のソースコードはこのようであった。

ソースコード 4　pointer.c

```
1 #include "matrix_util.h"
2 #include <stdio.h>
3
4 int main() {
5   int n, i, j;
```

```
6   double *v;
7   double **m;
8   n = 10;
9
10  /* test for vector */
11  v = alloc_dvector(n);
12  for (i = 0; i < n; ++i) v[i] = i;
13  fprint_dvector(stdout, n, v);
14
15  printf("v     = %lu\n", (long)v);
16  printf("&v[0] = %lu\n", (long)&v[0]);
17
18  printf("(v+2) = %lu\n", (long)(v+2));
19  printf("&v[2] = %lu\n", (long)&v[2]);
20
21  printf("*v       = %10.5f\n", *v);
22  printf("v[0]     = %10.5f\n", v[0]);
23
24  printf("*(v+2)   = %10.5f\n", *(v+2));
25  printf("v[2]     = %10.5f\n", v[2]);
26
27  printf("(v+2)[3] = %10.5f\n", (v+2)[3]);
28  printf("*(v+2+3) = %10.5f\n", *(v+2+3));
29
30  free_dvector(v);
31
32  /* test for matrix */
33  m = alloc_dmatrix(n, n);
34  for (i = 0; i < n; ++i)
35    for (j = 0; j < n; ++j)
36      m[i][j] = 100 * i + j;
37  fprint_dmatrix(stdout, n, n, m);
38
39  printf("m        = %lu\n", (long)m);
40  printf("&m[0]    = %lu\n", (long)&m[0]);
41
42  printf("m[0]     = %lu\n", (long)m[0]);
43  printf("&m[0][0] = %lu\n", (long)&m[0][0]);
44
45  printf("m[2]     = %lu\n", (long)m[2]);
46  printf("&m[2][0] = %lu\n", (long)&m[2][0]);
47
48  printf("m+2      = %lu\n", (long)(m+2));
49  printf("&m[2]    = %lu\n", (long)&m[2]);
```

```
50
51    printf("(*(m+2))[3] = %10.5f\n", (*(m+2))[3]);
52    printf("*(*(m+2)+3) = %10.5f\n", *(*(m+2)+3));
53    printf("m[2][3]     = %10.5f\n", m[2][3]);
54
55    printf("*(m+2)[3]   = %10.5f\n", *(m+2)[3]);
56    printf("*((m+2)[3]) = %10.5f\n", *((m+2)[3]));
57    printf("*(m[5])     = %10.5f\n", *(m[5]));
58    printf("m[5][0]     = %10.5f\n", m[5][0]);
59
60    free_dmatrix(m);
61  }
```

6 行目で v がポインタ変数として定義されている。

15 行目のプリント関数は v を出力している。v は v[0] のアドレスを表すので出力されるのは v[0] のアドレス (具体的にはわからない) であるはずだ。

16 行目は&v[0] を出力している。&v[0] は v[0] が格納されているアドレスを表すので出力されるのは 15 行目と同じ v[0] のアドレスであるはずだ。ただしアドレスは具体的にはわからないし、プログラムの実行環境によって異なる。

18 行目は v+2 を出力している。v+2 は v[2] のアドレスを表すので出力されるのは v[2] のアドレスである。*v は double 型で定義されていたので 8byte である。なので v[0] のアドレスに 16 を足したものが出力される。

19 行目は&v[2] を出力している。&v[2] は v[2] が格納されているアドレスを表すので出力されるのは 18 行目と同じ v[2] のアドレスである。

21 行目は*v を出力している。*v は v のアドレスに格納されている数値を表すので出力されるのは v[0] の値である。出力の表示桁数の指定が %10.5f になっているので出力は全体の桁数が最大で 10、小数点以下の桁数が最大 5 である。なので出力は 0.00000 である。(数値の出力は以下もこれと同じ理由で小数点以下 5 桁まで表示される。)

22 行目は v[0] を出力している。これは取りも直さず v[0] の値を表すので出力されるのは 21 行目と同じ 0.00000 である。

24 行目は*(v+2) を出力している。*(v+2) はアドレス v+2 に格納されている数値を表すので出力されるのじゃ v[2]、すなわち 2.00000 である。

25 行目は v[2] を出力している。これは取りも直さず v[2] の値なので出力は 2.00000 である。

27 行目は (v+2)[3] を出力している。これはアドレス v+2 から 3 つ (24 バイト) 進んだ先のアドレスに格納されている値を表すので、出力されるのは v[5]、すなわち 5.00000 である。

28 行目は*(v+2+3) を出力している。これはアドレス v+5 に格納されている値を表すので出力は v[5] す、すなわち 5.00000 である。

次に行列のテストについても同様に出力を予想する。

7 行目で m がポインタ変数 (ポインタのポインタ変数) として定義されている。

39 行目は m を出力している。これは m[0] のアドレスを表すので出力されるのは m[0] のアドレス。

40 行目は&m[0] を出力している。&m[0] は m[0] のアドレスを表す。出力されるのは m[0] のアドレス。

42 行目は m[0] を出力している。これは m[0][i] の配列の先頭アドレス、すなわち m[0][0] のアドレスを表す。なので出力されるのは m[0][0] のアドレス。

43 行目は&m[0][0] を出力している。これは m[0][0] のアドレスを表すので出力されるのは m[0][0] のアドレス。

45 行目は m[2] を出力している。m[2] は m[2][i] の配列の先頭アドレスを表す。なので出力されるのは m[2][0] のアドレスで配列は double 型で定義されていて (8byte)、配列のサイズが 10 なのでこのアドレスは m[0][0] のアドレスに 160 を足したものになると考えられる。

46 行目は&m[2][0] を出力している。これは m[2][0] のアドレスを表すので出力されるのは m[2][0] のアドレス。

48 行目は m+2 を出力している。これは m[2] のアドレスをあらわすので出力されるのは m[2] のアドレスである。またこれは配列の定義から m[0] のアドレスに 16 を足したものになると考えられる。

49 行目は&m[2] を出力している。これは m[2] のアドレスを表すので出力されるのは m[2] のアドレス。

51 行目は (*(m+2))[3] を出力している。ちょっと複雑なので丁寧に考える。まず、m+2 は m[2] のアドレスを表すのであった。そして*をポインタ変数に作用させるとそのアドレスに格納されている値を返すのであった。なのでと考えることができて、出力されるのは m[2][3] つまり 203.00000 である。

52 行目は*(*(m+2)+3) を出力している。これも丁寧に考える。まず*(m+2) は m[2] に格納されている値つまり m[2][0] のアドレスを表している。ここで v がポインタ変数であるとすると v+2 は v[2] のアドレスを表す。なので m[2]+3 は m[2][3] のアドレスを表す。同じく v がポインタ変数だとすると*v は v に格納されている値を表すので*(m[2]+3) は m[2][3] を表す。よって出力されるのは m[2][3] の値、すなわち 203.00000 である。

53 行目は m[2][3] を出力している。これは取りも直さず m[2][3] の値なので出力されるのは 203.00000 である。

55 行目は*(m+2)[3] を出力している。間接演算子*より添え字演算子 [3] の方が優先順位が高い。なのでまず (m+2)[3] について考える。m[3] は m+3 を表すのだから (m+2)[3] は m+5、つまり m[5] を表す。m[5] は m[5][i] の先頭アドレス、つまり m[5][0] のアドレスを表すので出力されるのは 500.00000 である。

56 行目は*((m+2)[3]) を出力している。(m+2)[3] は m[5] のことである。m[5] は m[5][0] のアドレスを表すので出力されるのは m[5][0]、つまり 500.00000 である。

57 行目は*(m[5]) を出力している。これはアドレス m[5] に格納されている値を表す。m[5] は m[5][0] のアドレスなので出力されるのは 500.00000 である。

58 行目は m[5][0] を出力している。これは取りも直さず m[5][0] の値を表すので出力されるのは 500.00000 である。

以上が予想される出力である。

実際に pointer.c を走らせて得られた出力をまとめた。

```
10
   0.00000    1.00000    2.00000    3.00000    4.00000    5.00000    6.00000    7.00
v      = 25769804784
&v[0] = 25769804784
(v+2) = 25769804800
&v[2] = 25769804800
*v        =    0.00000
v[0]      =    0.00000
*(v+2)    =    2.00000
v[2]      =    2.00000
(v+2)[3] =    5.00000
*(v+2+3) =    5.00000
10 10
   0.00000    1.00000    2.00000    3.00000    4.00000    5.00000    6.00000    7.00
 100.00000  101.00000  102.00000  103.00000  104.00000  105.00000  106.00000  107.00
 200.00000  201.00000  202.00000  203.00000  204.00000  205.00000  206.00000  207.00
 300.00000  301.00000  302.00000  303.00000  304.00000  305.00000  306.00000  307.00
 400.00000  401.00000  402.00000  403.00000  404.00000  405.00000  406.00000  407.00
 500.00000  501.00000  502.00000  503.00000  504.00000  505.00000  506.00000  507.00
 600.00000  601.00000  602.00000  603.00000  604.00000  605.00000  606.00000  607.00
 700.00000  701.00000  702.00000  703.00000  704.00000  705.00000  706.00000  707.00
 800.00000  801.00000  802.00000  803.00000  804.00000  805.00000  806.00000  807.00
 900.00000  901.00000  902.00000  903.00000  904.00000  905.00000  906.00000  907.00
m         = 25769804784
&m[0]     = 25769804784
m[0]      = 25770100704
&m[0][0] = 25770100704
m[2]      = 25770100864
&m[2][0] = 25770100864
m+2       = 25769804800
&m[2]     = 25769804800
(*(m+2))[3] =  203.00000
*(*(m+2)+3) =  203.00000
m[2][3]      =  203.00000
*(m+2)[3]    =  500.00000
*((m+2)[3]) =  500.00000
*(m[5])      =  500.00000
m[5][0]      =  500.00000
```

結果は予想と一致していた。

5　応用課題 EX3-2

5.1　実験概要

Laplace 方程式の境界値問題を Gauss-Seidel 法、SOR 法で解くプログラムを作成した。
まず、Gauss-Seidel 法のプログラムである。

ソースコード 5　jacobi 法で Laplace 方程式を解くプログラム

```
1   double **g; // D^(-1)
2   double **h; // E + F
3   double **c; // D^(-1) * (E + F)
4
5   double *z; // D^(-1) * b
6   double *b; // vector b
7   double *x; // vector x
8   double *temp;
9
10  double w; // optimal parameter
11
12  do{
13    /* initialize */
14    sum_of_error = 0.0;
15    for(i=0;i<m;i++){
16      y[i] = 0.0;
17    }
18
19    /* keep current x[i] */
20    for(i=0;i<m;i++){
21      temp[i] = x[i];
22    }
23
24    /* multiply matrix c and vector x and update x[i] asap  */
25    for(i=0;i<m;i++){
26      for(j=0;j<m;j++){
27        y[i] += c[i][j] * x[j]; // D^(-1) * (E + F) * x
28      }
29      x[i] = -y[i] + z[i];
30    }
31
32    /* count iteration */
33    l = l + 1;
34
35    /* calc diff of temp[i] and x[i] */
36    for(i=0;i<m;i++){
```

```
37        sum_of_error += pow(x[i] - temp[i], 2);
38      }
39    }while((pow(sum_of_error, 0.5) > epsilon) && (l < LMAX)); // test for convergence and
        set a limit in case
```

次に SOR 法のプログラムである。

ソースコード 6　jacobi 法で Laplace 方程式を解くプログラム

```
1   double **g; // D^(-1)
2   double **h; // E + F
3   double **c; // D^(-1) * (E + F)
4
5   double *z; // D^(-1) * b
6   double *b; // vector b
7   double *x; // vector x
8   double *temp;
9   double *temp_x;
10
11  do{
12    /* initialize */
13    sum_of_error = 0.0;
14    for(i=0;i<m;i++){
15      y[i] = 0.0;
16    }
17
18    /* keep current x[i] */
19    for(i=0;i<m;i++){
20      temp[i] = x[i];
21    }
22
23    /* multiply matrix c and vector x and update x[i] with accel parameter  asap  */
24    for(i=0;i<m;i++){
25      for(j=0;j<m;j++){
26        y[i] += c[i][j] * x[j]; // D^(-1) * (E + F) * x
27      }
28      temp_x[i] = -y[i] + z[i];
29      x[i] = x[i] + w * (temp_x[i] - x[i]);
30    }
31
32    /* count iteration */
33    l = l + 1;
34
35    /* calc diff of temp[i] and x[i] */
36    for(i=0;i<m;i++){
37      sum_of_error += pow(x[i] - temp[i], 2);
```

```
38      }
39    }while((pow(sum_of_error, 0.5) > epsilon) && (l < LMAX)); // test for convergence and
        set a limit in case
```

まず Gauss-Seidel 法について、jacobi 法について行ったのと同様の実験をした。

次に SOR 法について、かなり詳しく修正パラメータと反復回数の関係を調べた。

また最後に、jacobi 法、gauss-seidel 法、いくつかの修正パラメータでの SOR 法の計算速度 (反復回数) の比較をした。

5.2 実験結果

まずは Gauss-Seidel 法について。

解の形は次のようになった。

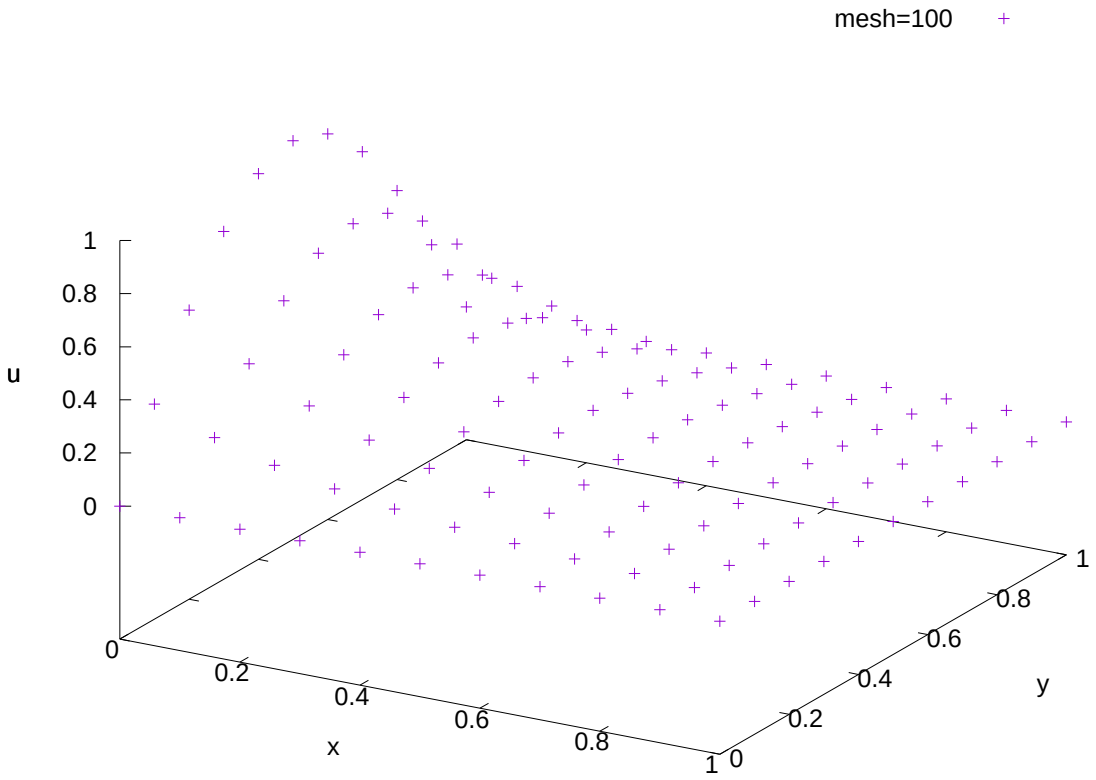

図 12　mesh 数が 100 の解。

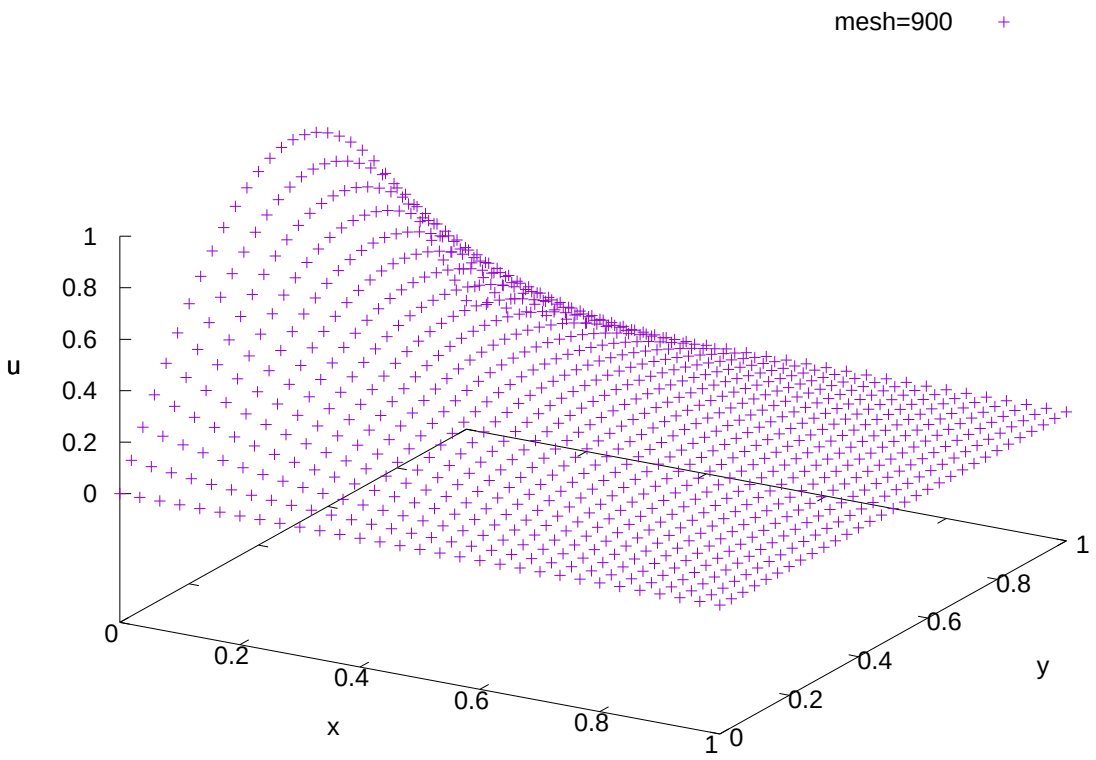

図 13　mesh 数が 900 の解。

次にメッシュ数を増やしたときの反復回数の変化である。

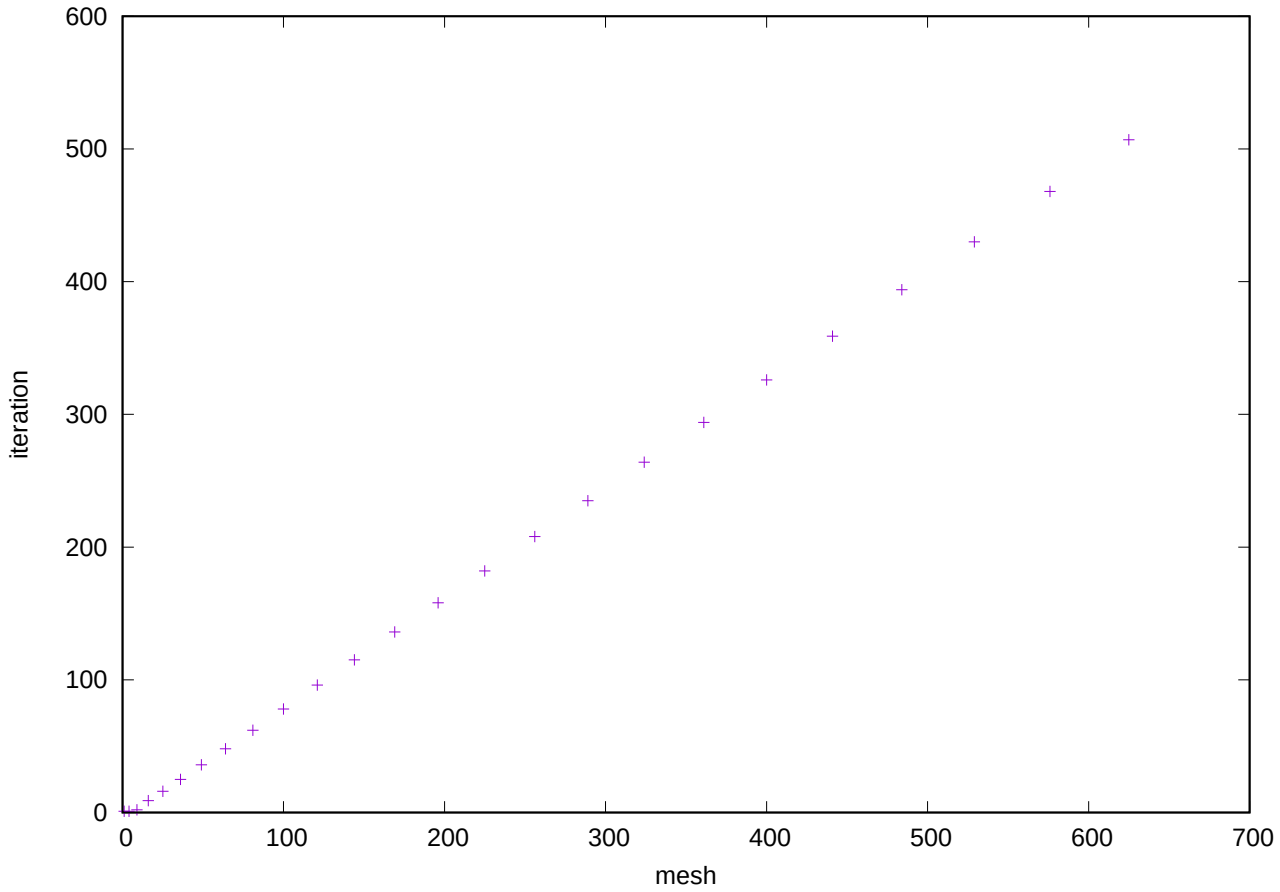

図 14　反復回数の変化。反復回数がメッシュ数に比例して増加していることが分かる。

次に SOR 法について。以下 ω は修正パラメータのことを指す。

まず、$\omega = 1$ のとき、反復回数が Gauss-Seidel 法の場合と一致することを確かめた。

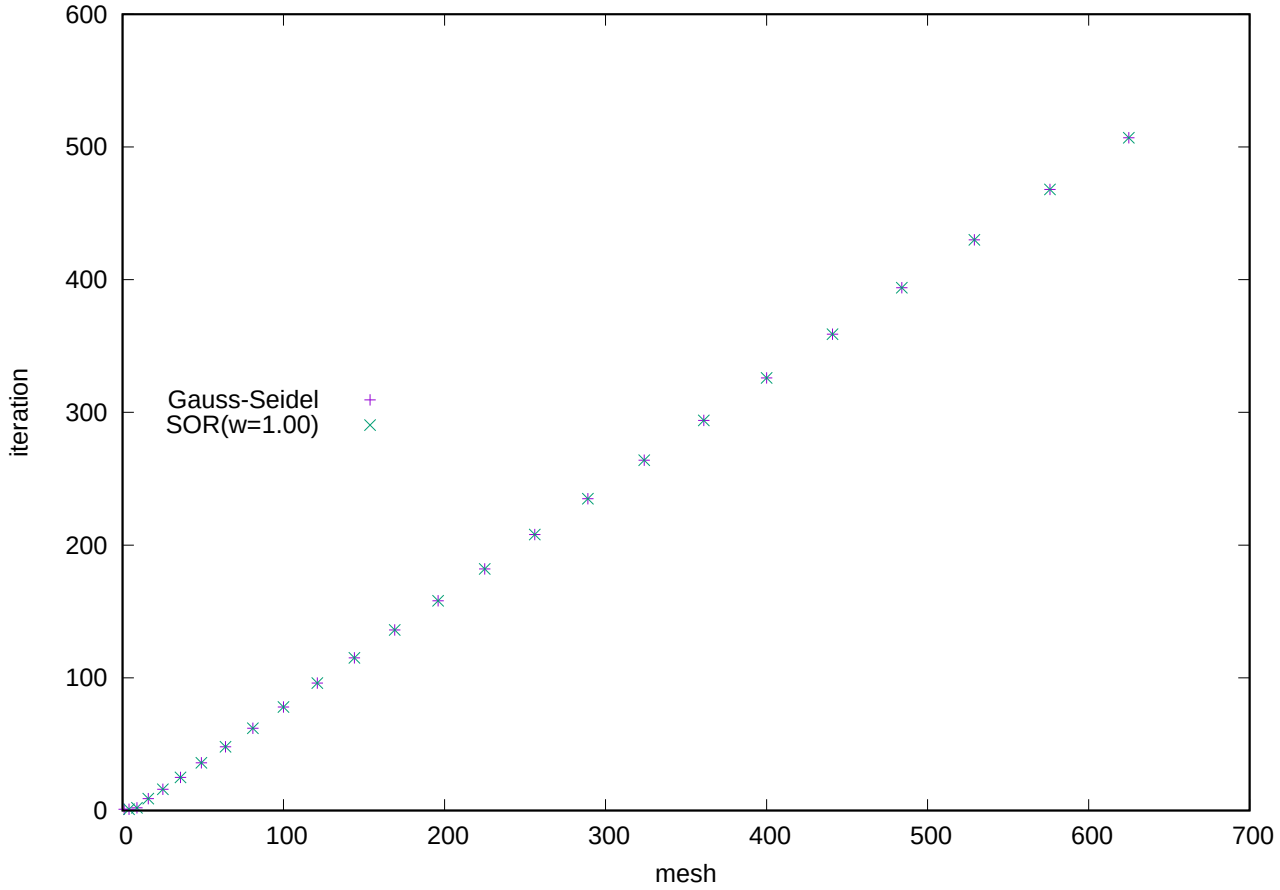

図 15　Gauss-Seidel 法と SOR 法 ($\omega = 1$) の反復回数の変化の比較。完全に一致している。

次に試しに $\omega = 1.50$ で同様に反復回数の変化を調べてみた。

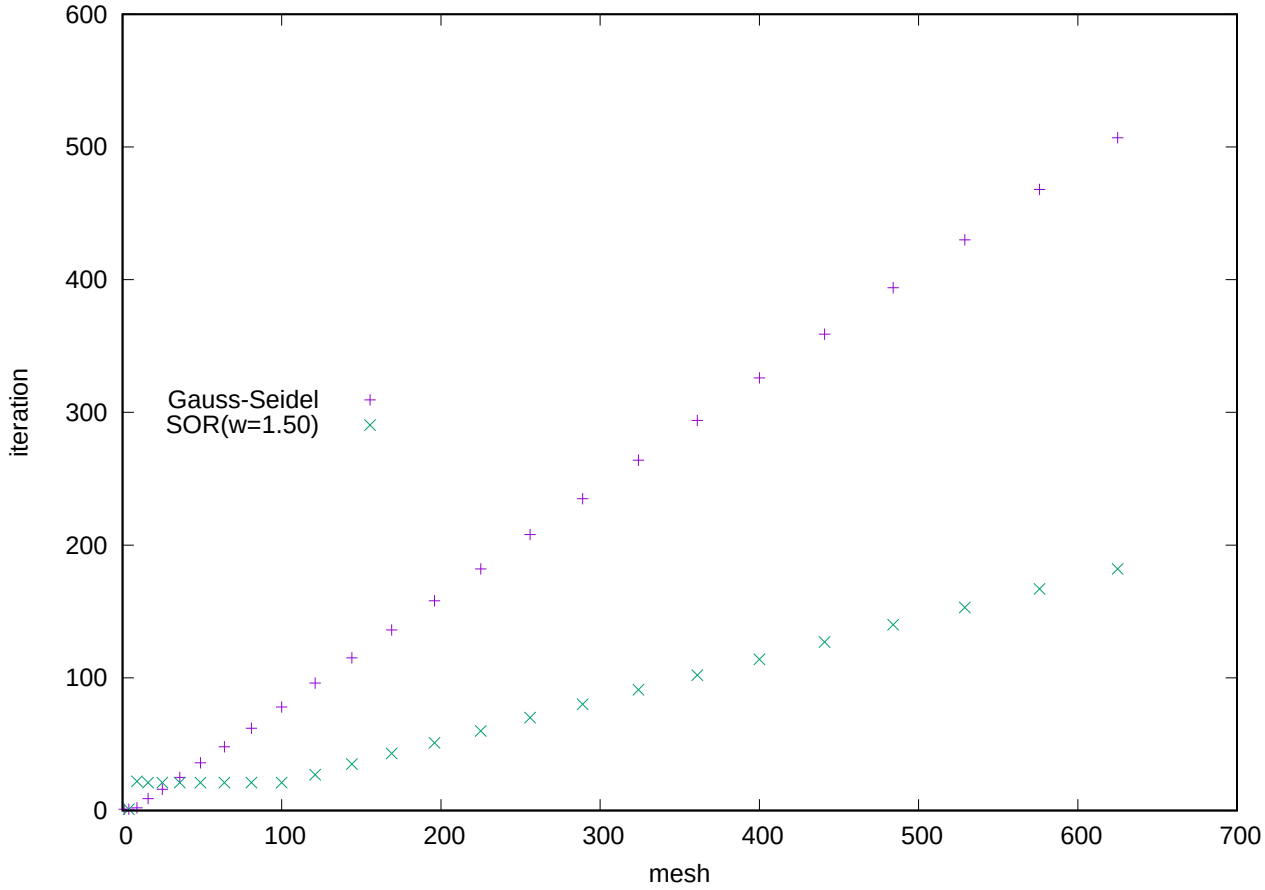

図 16　Gauss-Seidel 法と SOR 法 ($\omega = 1.50$) の反復回数の変化の比較。

これを見ると、SOR 法の方が反復回数が優れているというだけでなく SOR 法の反復回数の振る舞いが特徴的であることが分かる。具体的にはこの場合だと、$n = 10$ ぐらいまでは反復回数が一定値をとり、その後べき的に増加しているように見える。

特に、$n = 10$ 付近の反復回数の振る舞いが変化する点が何を意味しているかが気になる。以下、このような点 (以下転位点と呼ぶ) の意味について考える。

というわけで、次は少し方向性を変え、ある n の値について、ω を変化させたとき、反復回数はどう変化するかを調べてみた。

$n = 20$ について。次のような結果が得られた。

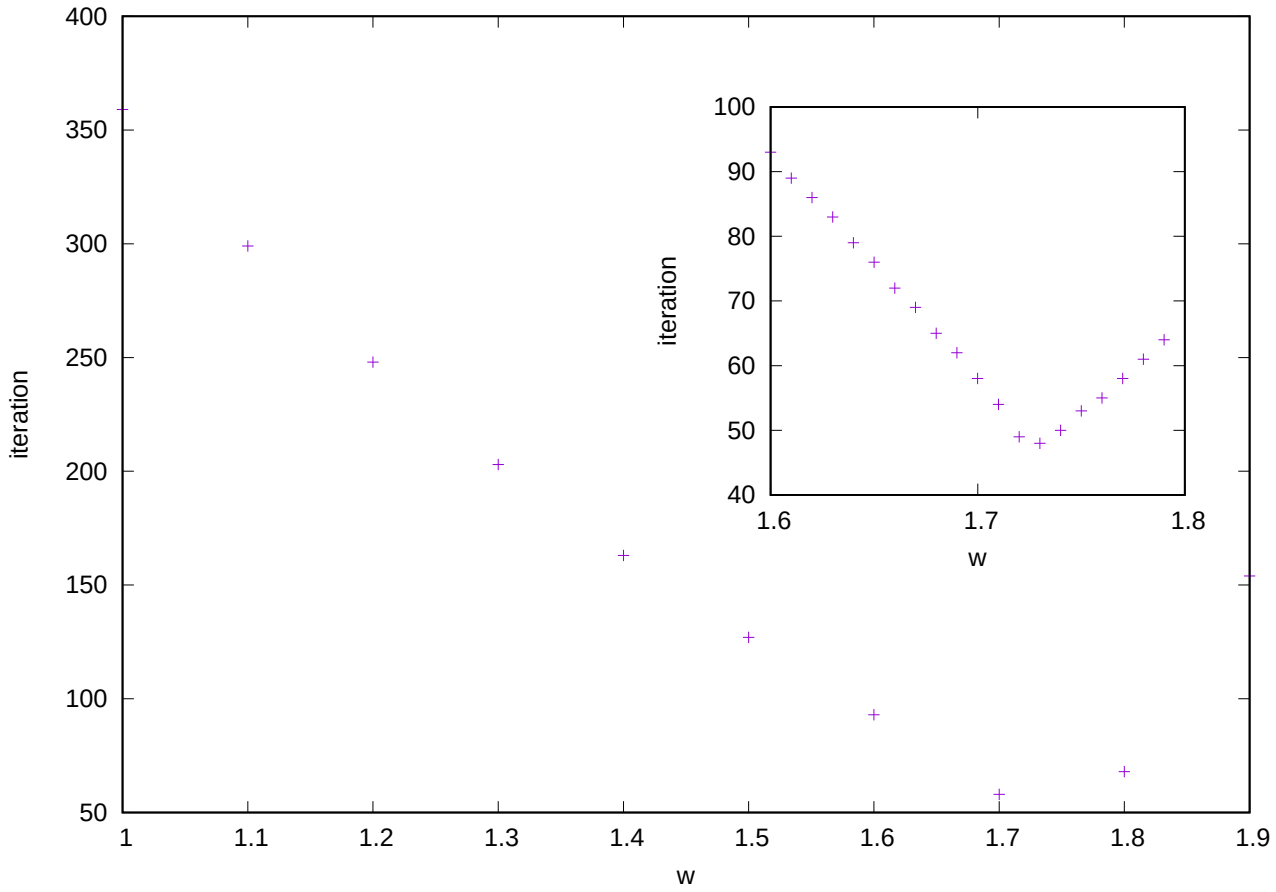

図 17　修正パラメータ ω をいろいろ変化させたときの反復回数の変化。$\omega = 1.73$ で反復回数が最小値を取ることが分かった。

この反復回数を最小にする修正パラメータは先ほどの転位点と何かしら関係があるに違いないと考え $\omega = 1.73$ に固定して n を変化させて反復回数の振る舞いを調べた。

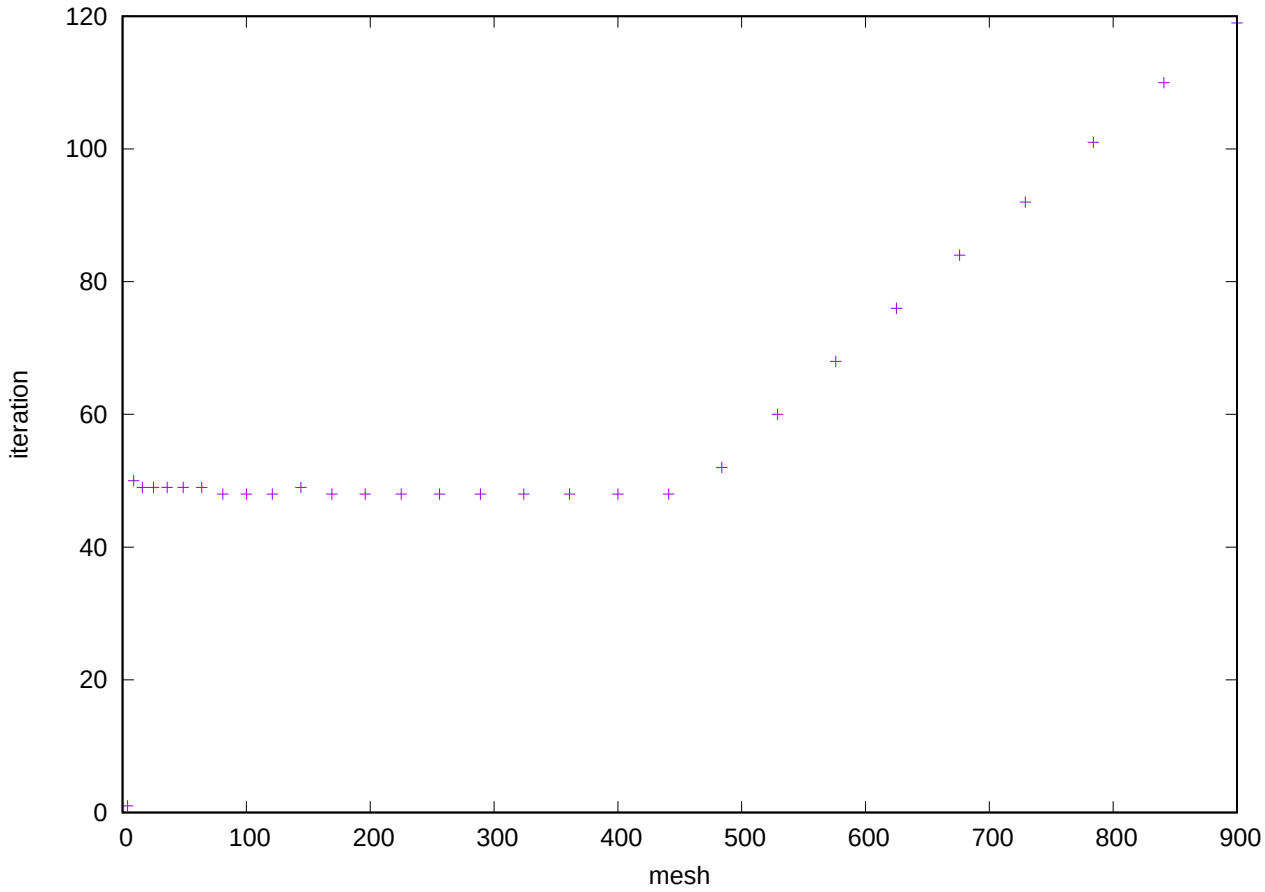

図 18　SOR 法 ($\omega = 1.73$) の反復回数の変化。

この場合転位点は $n = 20$ である。

以上の結果から得られる結論は、

「ある ω を固定して n を変化させたときに反復回数の振る舞いが変化する転位点の n の値は固定した ω が反復回数を最小にする n の値と一致する」

ということだ。

ところで本筋からは外れるが $w < 1$ の場合の反復回数の変化を興味があったので調べてみた。

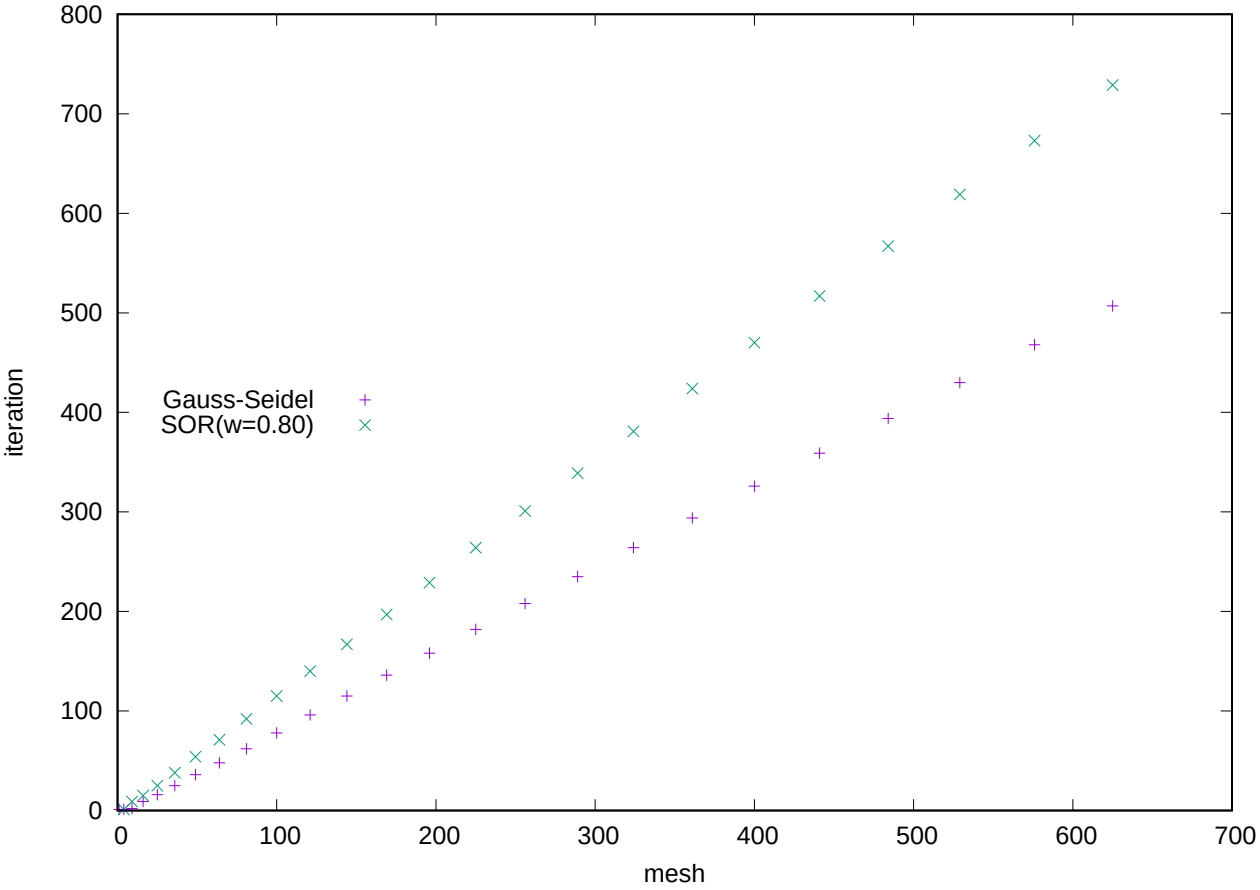

図 19　Gauss-Seidel 法と SOR 法 ($\omega = 0.80$) の反復回数の変化の比較。

確かに Gauss-Seidel 法よりも収束までにかかる反復回数が多くなることが分かった。
一方 $\omega > 2$ の場合は次のようになった。

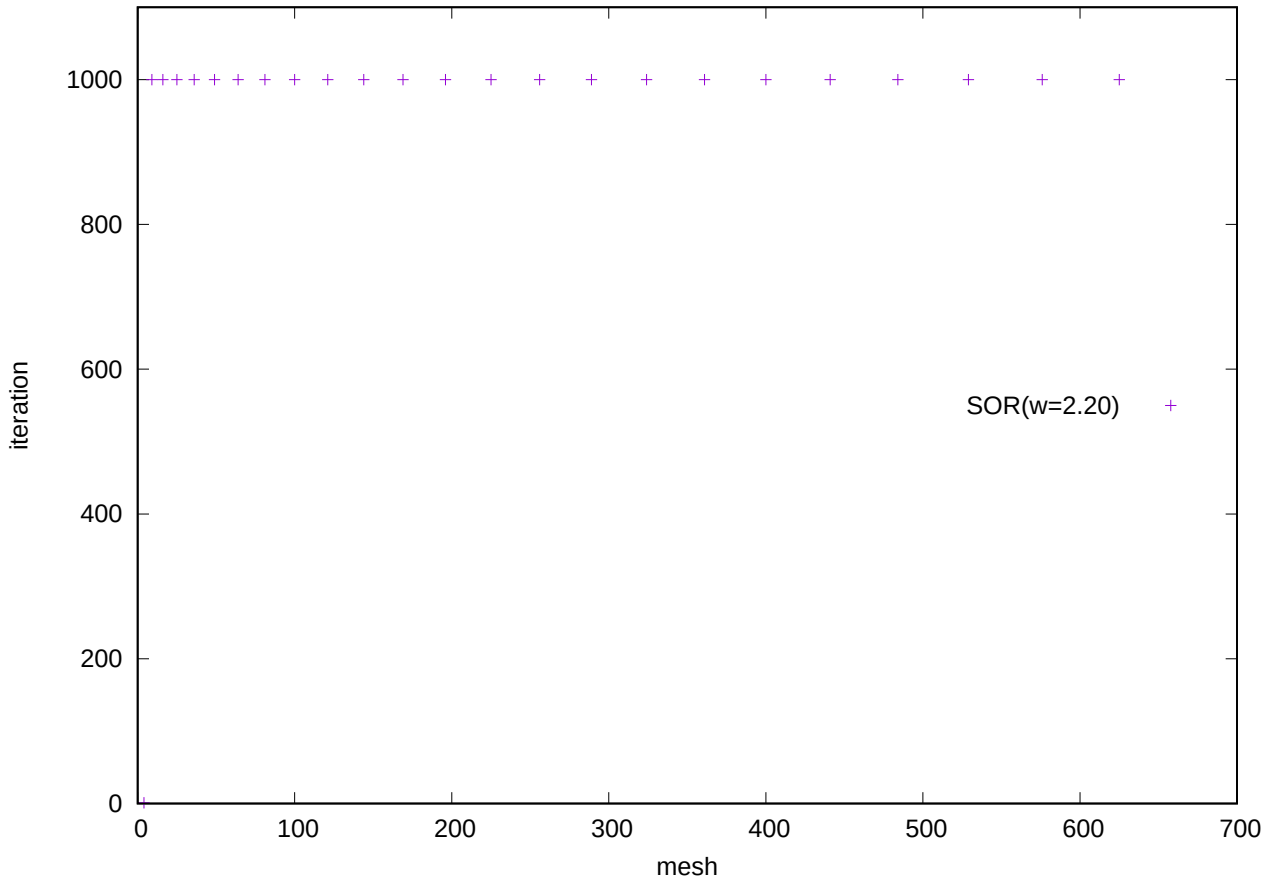

図 20　SOR 法 ($\omega = 2.20$) の反復回数の変化。

全て反復回数が 1000 になっているのはプログラムで反復回数の上限を設定していたからである。つまり、この場合、収束していないということになる。

ちなみにわざわざ記載しないが $\omega < 0$ の場合も収束しない。

修正パラメータで遊ぶのはここまでにして、課題の解答に移る。

$\omega = 1.50$、$n = 20$ の時の解の形は次のようになった。

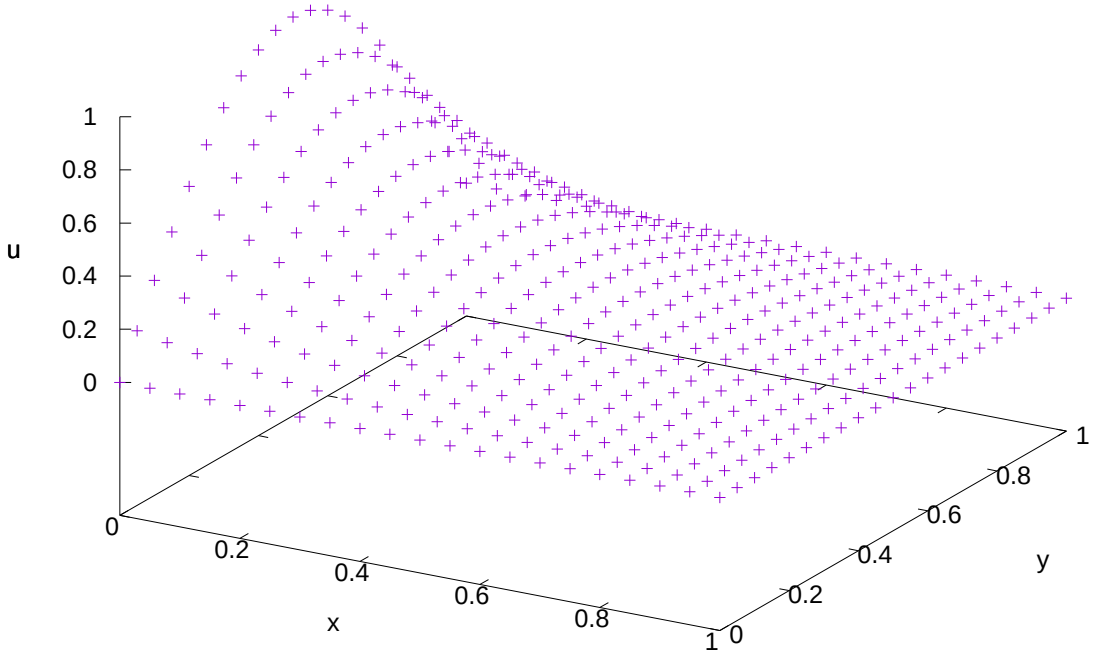

図 21　mesh 数が 400 の解。

最後に Jacobi 法、Gauss-Seidel 法、いくつかの修正パラメータでの SOR 法の反復回数の変化を重ねてプロットすると次のようになった。

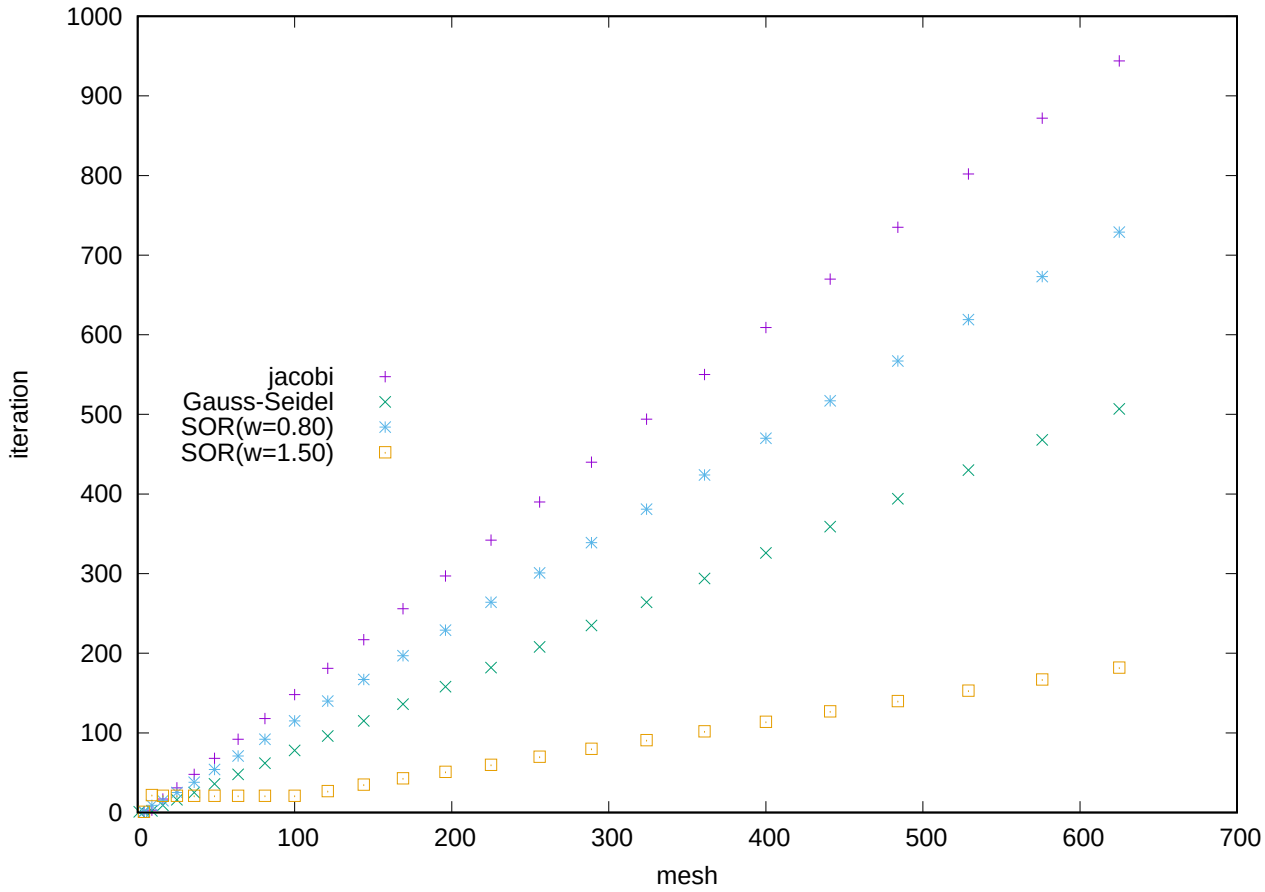

図 22　Jacobi 法、Gauss-Seidel 法、SOR 法 ($\omega = 0.80$、$\omega = 1.50$) の反復回数の変化の比較。

5.3　考察 (SOR 法について)

6　応用課題 EX3-3

6.1　実験概要

6.2　実験結果

6.3　考察

7　基本課題 EX4-1

7.1　実験概要

べき乗法を用いて最大固有値を計算するプログラムを作成し成分が $v_{ij} = min(i, j)(1 \leq i \leq n, 1 \leq j \leq n)$ の $n \times n$ 対称行列の固有値を求めた。

ソースコード 7　べき乗法により行列の最大固有値を計算するプログラム

```
1 /* べき乗法により与えられた行列の第一固有値を求めるプログラム */
2
3
```

```c
#include "../matrix_util.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define LMAX 2000
#define epsilon pow(10, -5)

int imin(int x, int y) { return (x < y) ? x : y; }
int imax(int x, int y) { return (x > y) ? x : y; }

int main(int argc, char** argv) {
  char* filename;
  FILE *fp;
  int i, j, k, l;

  int m, n, r;
  double **a;
  double *v, *c, *temp;
  double theo, eigen_value, diff, mother, child;


  if (argc < 2) {
    fprintf(stderr, "Usage: %s inputfile\n", argv[0]);
    exit(1);
  }
  filename = argv[1];

  /* read matrix A from a file */
  fp = fopen(filename, "r");
  if (fp == NULL) {
    fprintf(stderr, "Error: file can not open\n");
    exit(1);
  }
  read_dmatrix(fp, &m, &n, &a);
  printf("Matrix A:\n");
  fprint_dmatrix(stdout, m, n, a);

  /* allocate matrices and vectors */
  v = alloc_dvector(n);
  c = alloc_dvector(n);
  temp = alloc_dvector(n);
  /* calc theoretical value */
  theo = 0.5 / (1 - cos(M_PI/(2 * n + 1)));
```

```
48
49
50    /* initial value of vector v */
51    for(i=0;i<n;i++){
52      v[i] = 0.1 * (i+1);
53    }
54    for(i=0;i<n;i++){
55      c[i] = v[i];
56    }
57
58    /* perform power iteration */
59    l = 0;
60    do{
61    mother = 0.0;
62    child = 0.0;
63    for(i=0;i<n;i++){
64      temp[i] = c[i];
65    }
66    for(i=0;i<n;i++){
67      c[i] = 0.0;
68    }
69    for(i=0;i<n;i++){
70      for(j=0;j<n;j++){
71        c[i] += a[i][j] * temp[j];
72      }
73    }
74
75    for(i=0;i<n;i++){
76      mother += c[i] * temp[i];
77    }
78
79    for(i=0;i<n;i++){
80      child  += c[i] * c[i];
81    }
82    eigen_value = child / mother;
83    diff = fabs(eigen_value - theo);
84      printf("%d %20.15lf\n", l+1, eigen_value);
85    l += 1;
86    }while((diff > epsilon) && (l < LMAX));
87
88
89
90    printf("theoretical value is %lf\n", theo);
91  /*
```

```
92    行列//の出力 A
93
94    printf("%d %d\n", n, n); サイズの出力//
95
96    for (i = 0;i < n;i++){
97      for (j = 0;j < n;j++){
98        if(j < n - 1){
99          printf("%lf ", a[i][j]);
100       }else{
101         printf("%lf\n", a[i][j]);
102       }
103     }
104   }
105 */
106   free_dmatrix(a);
107   free_dvector(v);
108   free_dvector(c);
109   free_dvector(temp);
110
111 }
```

7.2 実験結果

まず 10×10 行列について最大固有値を求めた。得られた結果は 44.76606、と期待通り理論値と 10^{-5} の精度で一致していた。

次に行列の大きさ n を変えた時の最大固有値の収束の速さの変化を調べた。ここでは収束の速さを「反復前と後との差が 10^{-5} になるまでにかかった反復回数」とした。また反復回数とは初期ベクトルに行列 A をかけた回数である。

n を増やしたときの反復回数の変化は次のようになった。

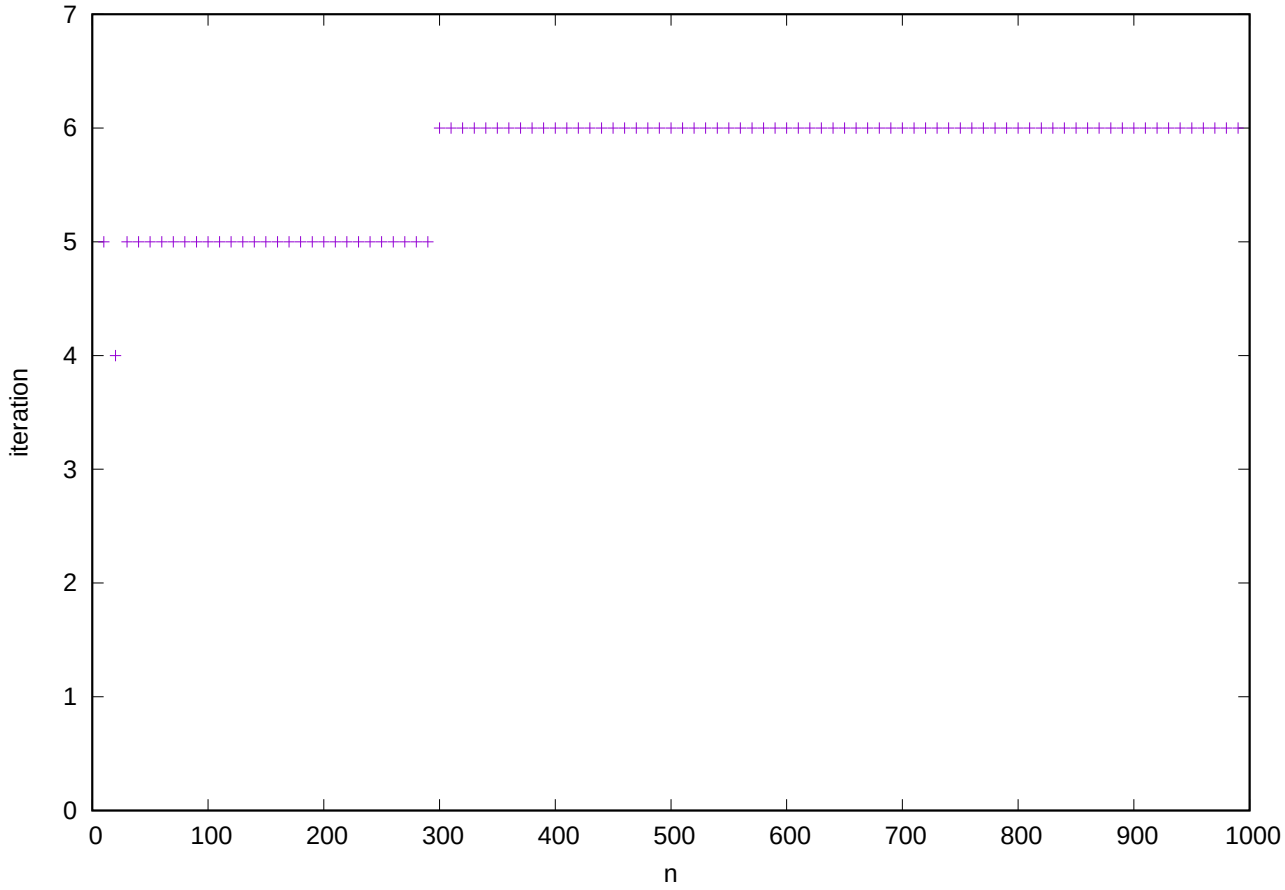

図 23　n を変化させたときの反復回数の変化。

このように n を増やしても反復回数はほとんど一定である。

7.3　考察 (反復回数が変わらない理由)

8　基本課題 EX4-2

8.1　実験概要

ファイル measurement.dat に収められている実験データを最小二乗法により任意の次数の多項式でフィッティングできるプログラムを作成した。

このプログラムを用いていくつかの次数の多項式を用いて実験データをフィッティングした。

その後、与えられた実験データに対してどの次数の多項式を用いるべきかを考察した。

8.2　実験結果

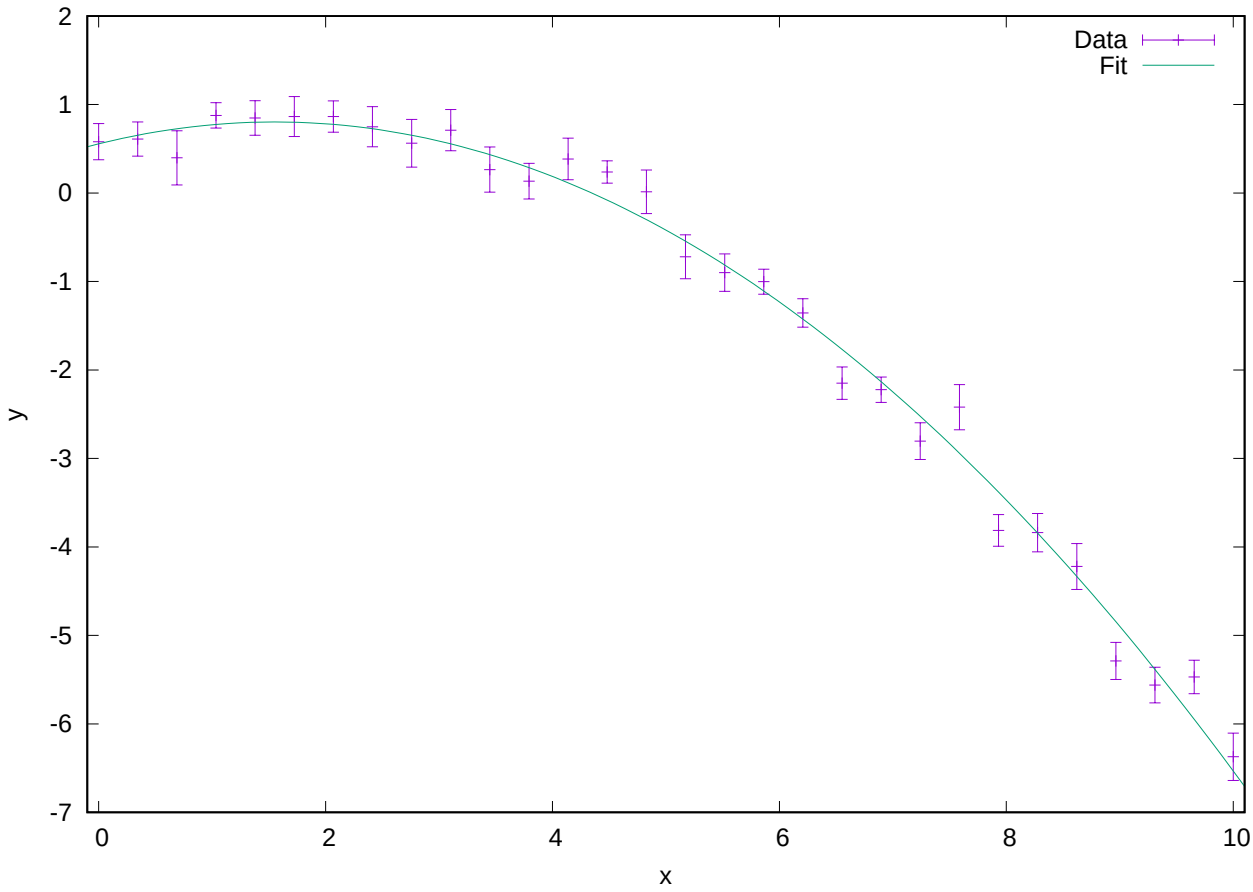

図 24　2 次の多項式によるフィッティング。具体的な関数形は $0.55584 + 0.31884x - 0.10277x^2$ である。残差は 270.283764 であった。

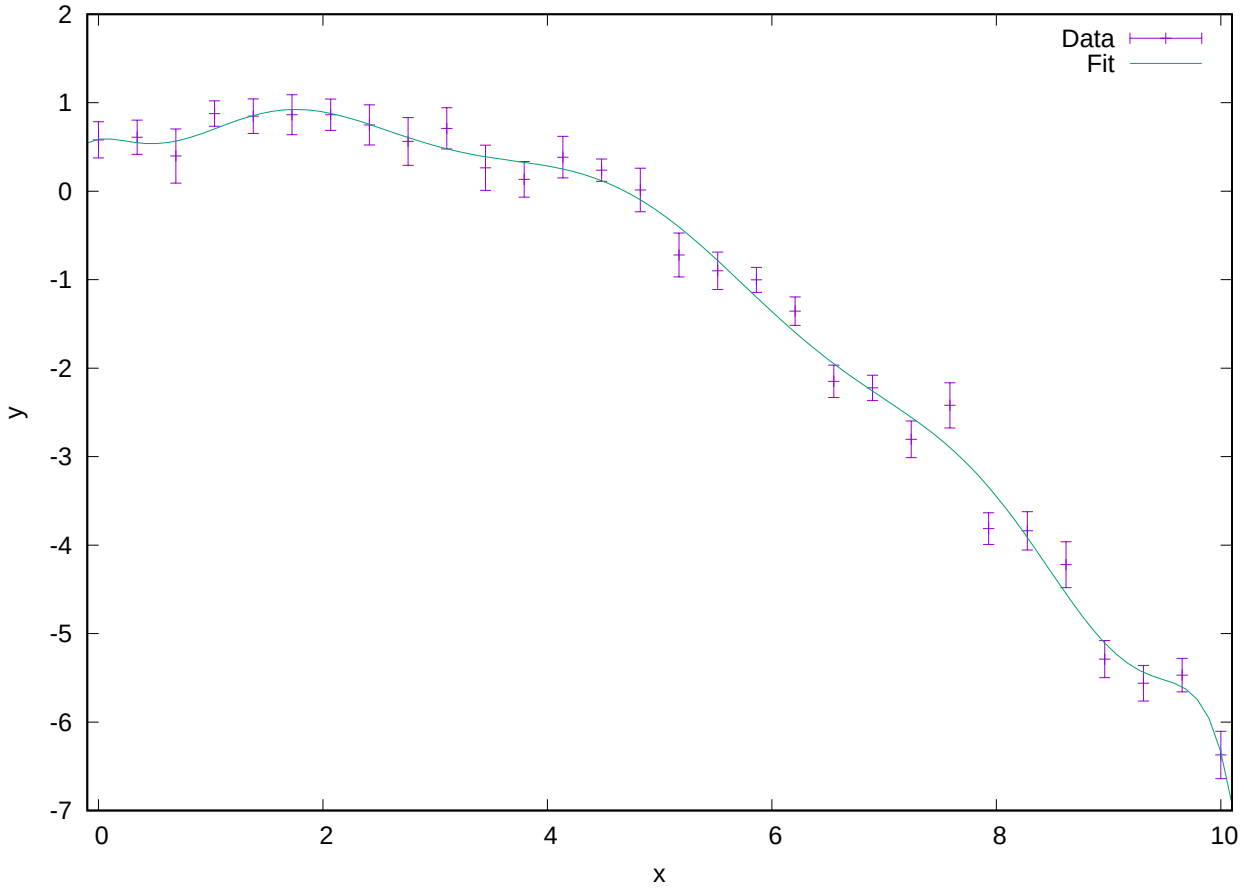

図 25　10 次の多項式によるフィッティング。具体的な関数形は 0.588516023906752 + 0.185402638994059 ∗ x ∗ ∗1 + −1.994525558824886 ∗ x ∗ ∗2 + 4.312846427303525 ∗ x ∗ ∗3 + −3.660413525518576 ∗ x ∗ ∗4 + 1.608970595609172 ∗ x ∗ ∗5 + −0.409620479440985 ∗ x ∗ ∗6 + 0.062714067684458 ∗ x ∗ ∗7 + −0.005696507048031 ∗ x ∗ ∗8 + 0.000282866752094 ∗ x ∗ ∗9 + −0.000005915134526 ∗ x ∗ ∗10 である。残差は 268.023440 であった。

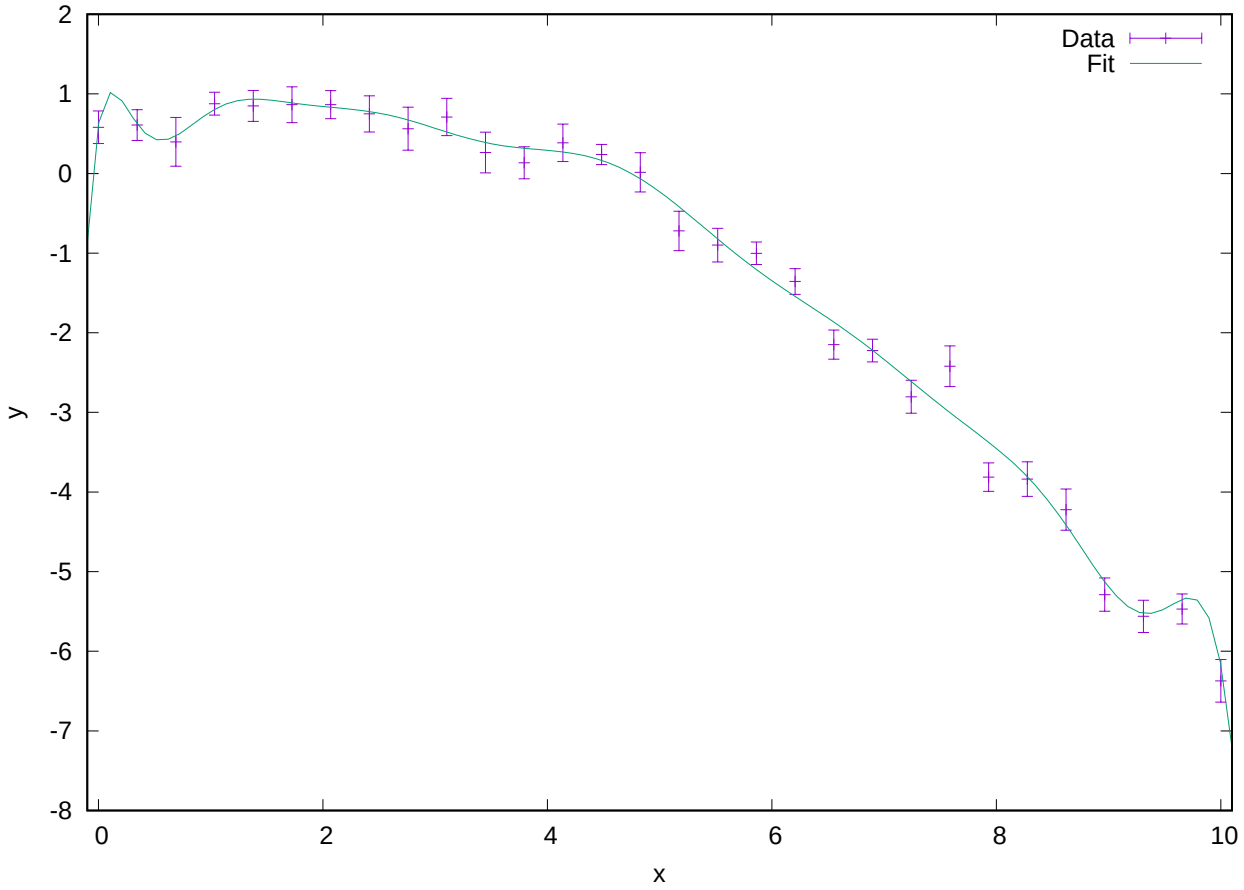

図 26　15 次の多項式によるフィッティング。具体的な関数形は長すぎるので略。残差は 181.754709 である。

以上の結果より、次数を大きくすればするほど実験データ一点一点によりフィットした曲線が得られて、残差も小さくなるので最も良い多項式の次数は 29(30 より大きいと実験データの数よりも未知パラメータの方が多くなり曲線が決まらない) である、と結論付けるべきだろうか？いやそうではない。

目的が残差を最も小さくすることであればもちろん次数を大きくとればいい話だが、”実験データを演繹的に予測する” 曲線を求めるというのが本来の目的であるので、最良の次数を求めるには違ったアプローチが必要である。

そこで今回は交差検証法を用いて最適な次数を求めることにする。

交差検証法とはここでは実験データをトレーニングデータとテストデータの 2 つに分けて、トレーニングデータに対して回帰分析を行った得られたフィッティング曲線がテストデータに対してもフィットしているかを調べる手法である。

そのために mesuament.dat で与えられている実験データを 3 つの組に分ける。x の値の小さい方から 10 つをデータ 1、次の 10 つをデータ 2、最後の 10 つをデータ 3 と呼ぶことにする。

そしてこの 3 組のデータをトレーニングデータとテストデータの次のように 2 組に分け、3 回検証を行った。

表 1　各検証で用いたデータの分類

	検証 1	検証 2	検証 3
トレーニングデータ	データ 1 と 2	データ 2 と 3	データ 3 と 1
テストデータ	データ 3	データ 1	データ 2

2、3、4 次の多項式をトレーニングデータにフィットするように回帰分析で求めこの多項式でテストデータを含めた全データに対してフィットしているかどうかを確かめた。データ点の分布から線形近似では適切な回帰分析とは言えない、そして 5 次以上などの高次の多項式では以下で 4 次に対して見られるように回帰分析として適当でないので最初から検証対象から除外した。

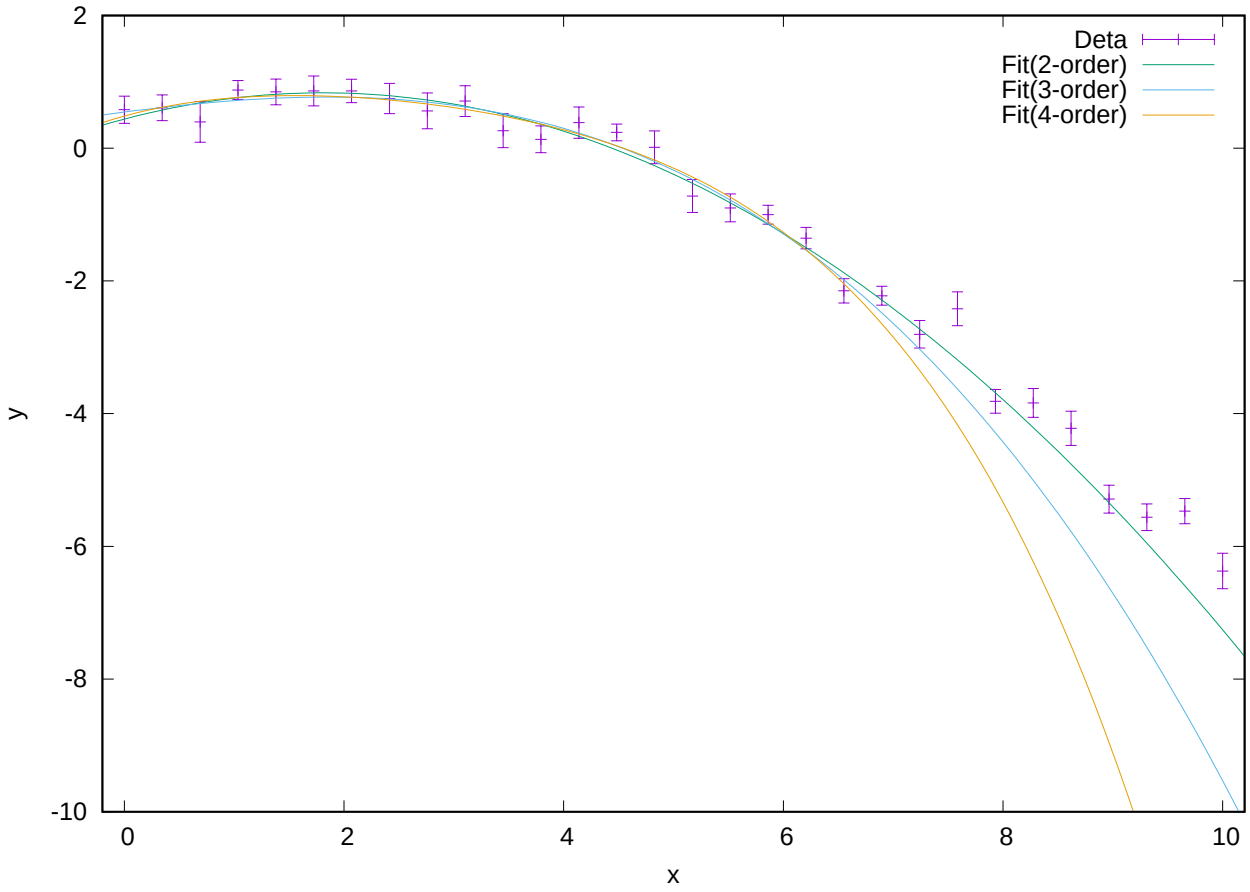

図 27　検証 1 の結果。2 次の多項式が最もよくテストデータにフィットしている。

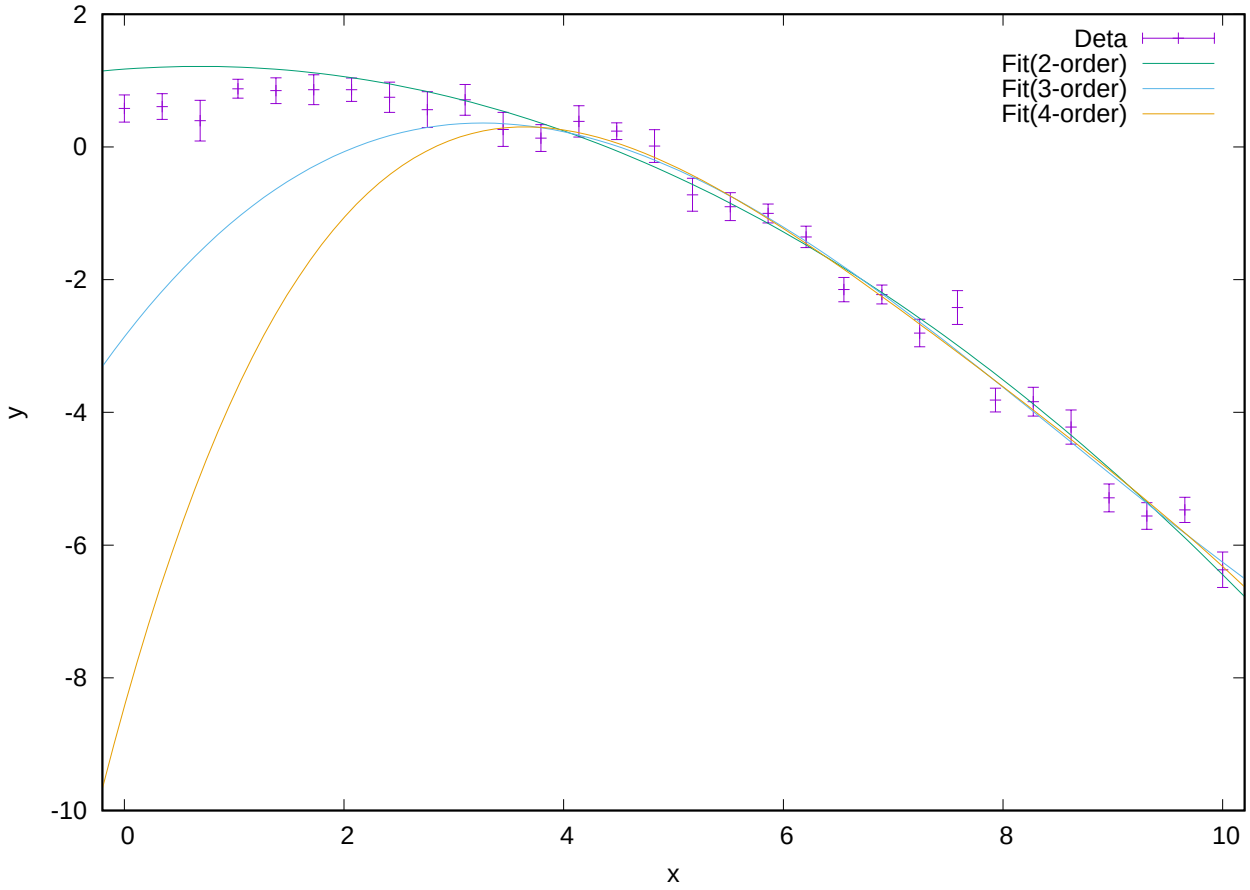

図 28　検証 2 の結果。2 次の多項式が最もよくテストデータにフィットしている。

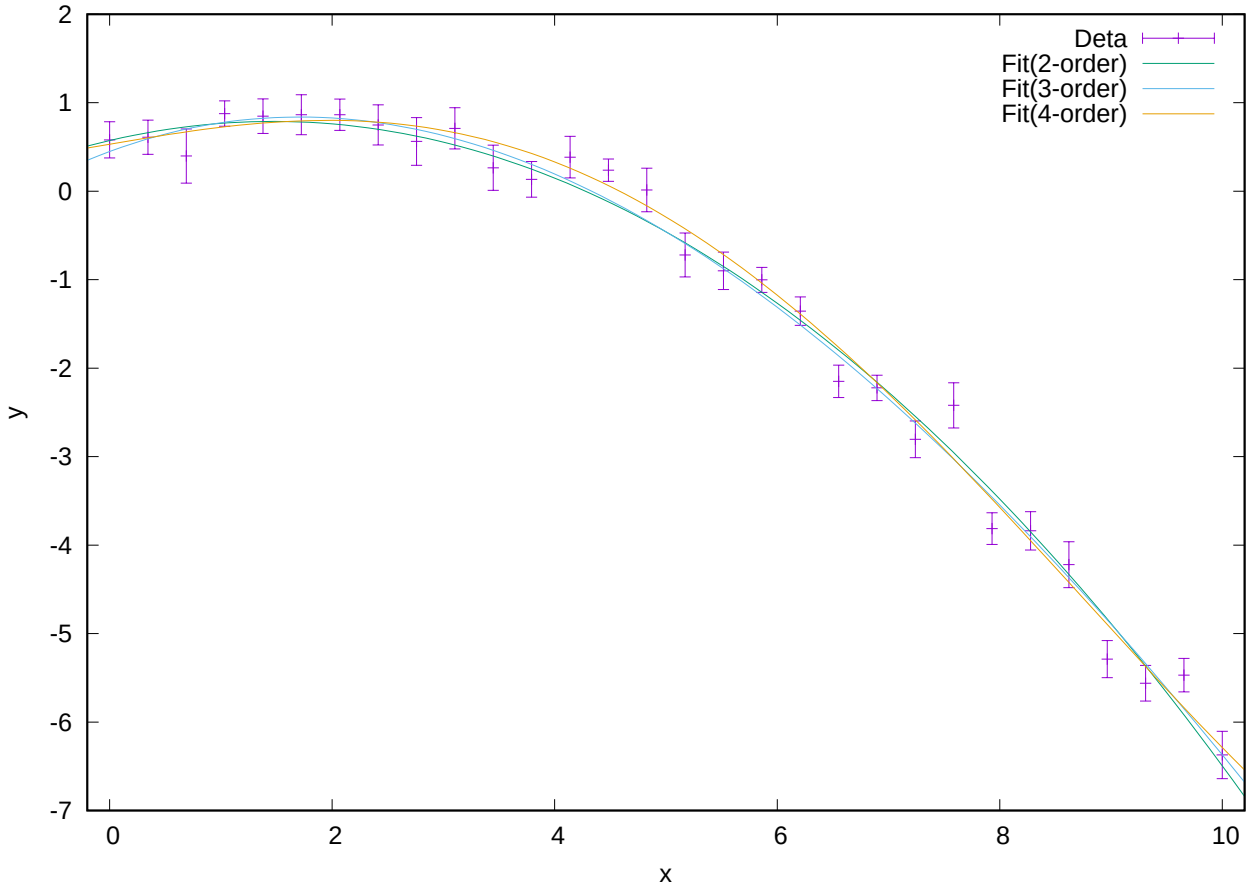

図 29　検証 3 の結果。この検証ではどの次数の多項式もテストデータにフィットしている。

以上の検証の結果から、残差等を計算するまでもなく、2 次の多項式を使うのが最も良いと結論付けることが出来る。

8.3　考察

9　応用課題 EX4-1

9.1　実験概要

9.2　実験結果

9.3　考察

10　応用課題 EX4-2

10.1　実験概要

特異値分解のサンプルプログラム (svd.c) をコンパイルし、入力 matrix2.dat を用いてプログラムを実行した。

10.2 実験結果

得られた出力をそのまま記載する。

ソースコード 8　svd.c の入力 matrix2.dat に対する実行結果

```
Matrix A:
4 3
   1.00000    2.00000    3.00000
   6.00000    4.00000    5.00000
   8.00000    9.00000    7.00000
  10.00000   11.00000   12.00000
Result of SVD U:
4 3
  -0.13801   -0.61647   -0.05283
  -0.34037    0.37028    0.81421
  -0.54626    0.53543   -0.57516
  -0.75280   -0.44293    0.05891
Result of SVD S:
3
  25.34681    2.14879    1.70929
Result of SVD Vt:
3 3
  -0.55543   -0.58526   -0.59074
   0.67915    0.09067   -0.72838
   0.47986   -0.80576    0.34712
Reconstruction of the original matrix A:
4 3
   1.00000    2.00000    3.00000
   6.00000    4.00000    5.00000
   8.00000    9.00000    7.00000
  10.00000   11.00000   12.00000
Rank (r-1) approximation of A:
4 3
   1.04333    1.92724    3.03134
   5.33218    5.12139    4.51691
   8.47175    8.20785    7.34126
   9.95168   11.08113   11.96505
```

講義 L4 の例が再現されていることが確認できた。(有効桁数は異なるが)

10.3 考察

11 応用課題 EX4-3

11.1 考察 (LAPACK による特異値分解について)

課題にある通り、svd.c 中の LAPACK の特異値分解 dgesvd の呼び出し (54 行目) では、行列の次元 (m と n)、左特異ベクトル (u) と右特異ベクトル (vt) の順番が、もともとの dgesvd のドキュメント 1 とは逆になっている。しかし応用課題 4-2 で見たように svd.c の実行結果を見ると正しい特異値分解が得られている。

なぜうまくいくかは講義で習ったように、C と Fortran では二次元配列のメモリ上での並びが違うので、C 言語で書いた 2 次元配列は Fortran に渡ると転置されるからである。

というよりも、転置されてしまうことを考慮して m と n、u と vt を逆にしているのだ。

以下ではなぜうまくいくかを具体的に考察する。

行列 A が

$$A = U\Lambda V^T$$

と特異値分解されるとする。

行列 A を C 言語の 2 次元配列で記述してこれを dgesvd に取り込む、つまり Fortran に渡すと転置され dgesvd は行列

$$A^T = V\Lambda^T U^T$$

を行列 A として受け取ったと勘違いする。dgesvd はこれを素直に特異値分解し、行列 V を dgesvd の引数 u に、行列 U^T を dgesvd の引数 vt に格納して C 言語に返す。C 言語に帰ってくるときにこれらは再び転置されるので C 言語上で dgesvd の引数 u には行列 V^T が、dgesvd の引数 vt には行列 U が入力されている。なのでこのプログラムを実行すると U が出力されるべきところに V^T が、V^T が出力されるべきところに U が表示されてしまう。

なので C 言語プログラムで dgesvd を使うときは Fortran 上で本来 U が書き込まれるところに vt を、V^T が書き込まれるところに u を書いておけば、正しく u、vt が得られることになる。

12 応用課題 EX4-4

12.1 実験概要

12.2 実験結果

12.3 考察

参考文献