Homework#3

2018008168 김지효

# Problem1

1. Purpose

Gauss-Jordan Elimination은 forward elimination을 진행하는 Gauss Elimination에서 diagonal 위쪽 요소들까지 0으로 만드는 방법이다.

LU Decomposition은 주어진 행렬 A를 lower triangular matrix L과 upper triangular matrix U로 분해하는 방법이다.

Singular Value Decomposition은 singular matrix, near singluar matrix에서도 분해할 수 있는 방법이다.

1. Implementation

Linear equation의 A, b, x와 rows, cols를 갖는 lineq\_t 구조체와, Linear equation을 할당하는 alloc\_lineq, dat 파일을 읽어 초기화하는 init\_lineq, 해제하는 free\_lineq, 출력하는 print\_lineq 및 print\_solution을 구현하였다.

// lineq.h

typedef struct

{

int rows;

int cols;

float \*\*A;

float \*\*B;

float \*x;

float \*b;

} lineq\_t;

lineq\_t \*alloc\_lineq(int, int);

lineq\_t \*init\_lineq(const char \*);

void free\_lineq(lineq\_t \*);

void print\_lineq(lineq\_t \*);

void print\_solution(lineq\_t \*);

각 방법을 수행하는 run\_gaussj, run\_ludcmp, run\_svdcmp 함수를 구현하였다. 추가적으로, ludcmp와 svdcmp는 decomposition 이후에 x를 구하는 함수를 호출하였다. (각각 lubksb, svbksb)

// Gauss-Jordan

void run\_gaussj(const char \*filename)

{

lineq\_t \*lineq = init\_lineq(filename);

gaussj(lineq->A, lineq->rows, lineq->B, 1);

// ...

}

// LU Decompistion

void run\_ludcmp(const char \*filename)

{

lineq\_t \*lineq = init\_lineq(filename);

ludcmp(lineq->A, lineq->rows, indx, d);

lubksb(lineq->A, lineq->rows, indx, lineq->b);

// ...

}

// Singular Value Decomposition

void run\_svdcmp(const char \*filename)

{

lineq\_t \*lineq = init\_lineq(filename);

svdcmp(lineq->A, lineq->rows, lineq->cols, w, v);

svbksb(lineq->A, w, v, lineq->rows, lineq->cols, lineq->b, lineq->x);

// ...

}

1. Result

각 방법론을 이용하여 세 가지 선형방정식을 푼 결과는 다음과 같다.

테이블이(가) 표시된 사진

자동 생성된 설명

1. Discussion

두번째와 세번째 선형방정식에서는 A 행렬이 nonsingular matrix이기 때문에 x값이 비슷하게 나왔다. 하지만 첫번째 선형방정식을 볼 때, Gauss-Jordan Elimination에서는 값을 계산하지 못했고, LU Decompoisition에서는 값을 계산하였지만 best solution이 아니다. Singular Value Decompoisition에서는 best solution을 계산하였다.

# Problem2

1. Purpose

Iterative improvement는 선형방정식의 solution에 대한 오차를 줄이는 과정이다.

1. Implementation

// Iterative Improvement

void run\_mprove(const char \*filename)

{

lineq\_t \*lineq = init\_lineq(filename);

for (int i = 1; i <= lineq->rows; ++i)

{

for (int j = 1; j <= lineq->cols; ++j)

{

A[i][j] = lineq->A[i][j];

}

}

for (int i = 1; i <= lineq->cols; ++i)

{

lineq->x[i] = lineq->b[i];

b[i] = lineq->b[i];

}

ludcmp(lineq->A, lineq->rows, indx, d);

lubksb(lineq->A, lineq->rows, indx, lineq->x);

for (int i = 0; i < 10000; ++i)

mprove(A, lineq->A, lineq->rows, indx, b, lineq->b);

// ...

}

1. Result

Problem1의 LU Decompoistion과 비슷하게 결과가 나왔다.



1. Discussion

이미 큰 오차 없는 값을 계산한 상황보다는, 오차가 큰 값에 대하여 값을 수정하는 과정에서 의미가 있어 보인다.

# Problem3

1. Purpose

Inverse matrix 및 determinant는 LU Decomposition을 통해 구할 수 있다.

1. Implementation

Inverse matrix는 identity matrix의 각 column에 대한 solution을 구하는 방식으로, determinant는 diagonal의 곱을 구하는 방식으로 구현하였다.

void run\_inverse(const char \*filename)

{

lineq\_t \*lineq = init\_lineq(filename);

ludcmp(lineq->A, lineq->rows, indx, d);

for (int j = 1; j <= lineq->rows; ++j)

{

for (int i = 1; i <= lineq->cols; ++i)

col[i] = 0.0;

col[j] = 1.0;

lubksb(lineq->A, lineq->rows, indx, col);

for (int i = 1; i <= lineq->cols; ++i)

y[i][j] = col[i];

}

}

void run\_determinant(const char \*filename)

{

lineq\_t \*lineq = init\_lineq(filename);

ludcmp(lineq->A, lineq->rows, indx, d);

for (int j = 1; j <= lineq->rows; ++j)

{

\*d \*= lineq->A[j][j];

}

}

1. Result

테이블이(가) 표시된 사진

자동 생성된 설명