**Homework 8**

|  |
| --- |
| 그림입니다. 원본 그림의 이름: YU_UI_RGB-10.png 원본 그림의 크기: 가로 2256pixel, 세로 3047pixel 프로그램 이름 : Adobe ImageReady |

|  |  |
| --- | --- |
| 과목명 | 프로그래밍 언어 |
| 교수님 | 김영탁 교수님 |
| 이 름 | 김주환 |
| 학 번 | 21812158 |
| 일 자 | 2021.05.03.월 |

1. **n x m 행렬 연산 프로그램**

/\*

\* 파일명 : main\_Matrix\_Operations.cpp

\* 목적 및 기본 기능

\* 주어진 행렬에 기본 연산을 실행

\* 역행렬을 구하고 검증

\* 작성자 : 김주환(21812158)

\* 작성일 : 2021년 5월 3일

\*/

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include "Matrix.h"

void main() {

FILE\* fin\_ABC, \* fout;

const char\* input\_file\_ABC = "mtrx\_AB\_6x7\_C\_7x6.txt"; // 파일 이름

const char\* output\_file = "mtrx\_operations\_results.txt";

double\*\* mA, \*\* mB, \*\* mC, \*\* mD, \*\* mE, \*\* mF, \*\* inv\_mF, \*\* mI;

int a\_row\_size, a\_col\_size;

int b\_row\_size, b\_col\_size;

int c\_row\_size, c\_col\_size;

int d\_row\_size, d\_col\_size;

int e\_row\_size, e\_col\_size, k\_size;

int f\_row\_size, f\_col\_size;

fin\_ABC = fopen(input\_file\_ABC, "r"); // 입력받을 파일 열기

if (fin\_ABC == NULL) {

printf("Error in opening input data file (%s) !!", input\_file\_ABC);

exit;

}

fout = fopen(output\_file, "w"); // 입력할 파일 열기

if (fout == NULL) {

printf("Error in opening output data file (%s) !!", output\_file);

exit;

}

mA = fGetMtrx(fin\_ABC, &a\_row\_size, &a\_col\_size); // 기본 배열 읽고 출력

printMtrx((const char\*)"mA", mA, a\_row\_size, a\_col\_size);

fprintMtrx(fout, (const char\*)"mA", mA, a\_row\_size, a\_col\_size);

mB = fGetMtrx(fin\_ABC, &b\_row\_size, &b\_col\_size);

printMtrx((const char\*)"mB", mB, b\_row\_size, b\_col\_size);

fprintMtrx(fout, (const char\*)"mB", mB, b\_row\_size, b\_col\_size);

mC = fGetMtrx(fin\_ABC, &c\_row\_size, &c\_col\_size);

printMtrx((const char\*)"mC", mC, c\_row\_size, c\_col\_size);

fprintMtrx(fout, (const char\*)"mC", mC, c\_row\_size, c\_col\_size);

d\_row\_size = a\_row\_size;

d\_col\_size = b\_col\_size;

mD = addMtrx(mA, mB, a\_row\_size, a\_col\_size); // 덧셈

printMtrx((const char\*)"mC", mC, c\_row\_size, c\_col\_size);

fprintMtrx(fout, (const char\*)"mC", mC, c\_row\_size, c\_col\_size);

e\_row\_size = a\_row\_size;

e\_col\_size = b\_col\_size;

mE = subtractMtrx(mA, mB, e\_row\_size, e\_col\_size); // 뺄셈

printMtrx((const char\*)"mE = mA - mB", mE, e\_row\_size, e\_col\_size);

fprintMtrx(fout, (const char\*)"mE = mA - mB", mE, e\_row\_size, e\_col\_size);

f\_row\_size = a\_row\_size;

f\_col\_size = c\_col\_size;

k\_size = a\_col\_size;

mF = multiplyMtrx(mA, mC, f\_row\_size, k\_size, f\_col\_size); // 곱셈

printMtrx((const char\*)"mF = mA \* mC", mF, f\_row\_size, f\_col\_size);

fprintMtrx(fout, (const char\*)"mF = mA \* mC", mF, f\_row\_size, f\_col\_size);

inv\_mF = inverseMtrx(mF, f\_row\_size); // 역행렬

printMtrx((const char\*)"inv\_mF", inv\_mF, f\_row\_size, f\_col\_size);

fprintMtrx(fout, (const char\*)"inv\_mF", inv\_mF, f\_row\_size, f\_col\_size);

mI = multiplyMtrx(mF, inv\_mF, f\_row\_size, f\_col\_size, f\_col\_size); // 검증

printMtrx((const char\*)"mI = mF \* inv\_mF", mI, f\_row\_size, f\_col\_size);

fprintMtrx(fout, (const char\*)"mI = mF \* inv\_mF", mI, f\_row\_size, f\_col\_size);

fclose(fin\_ABC);

fclose(fout);

}

/\* Matrix.cpp \*/

#define \_CRT\_SECURE\_NO\_WARNINGS

#ifndef MATRIX\_H

#define MATRIX\_H

#include <stdio.h>

#include <stdlib.h>

double\*\* fGetMtrx(FILE\* fin, int\* row\_size, int\* col\_size) {

double d = 0.0;

double\*\* dm = NULL;

fscanf(fin, "%d %d", row\_size, col\_size);

dm = (double\*\*)calloc(\*row\_size, sizeof(double\*));

for (int i = 0; i < \*row\_size; i++) {

dm[i] = (double\*)calloc(\*col\_size, sizeof(double));

}

for (int i = 0; i < \*row\_size; i++) {

for (int j = 0; j < \*col\_size; j++) {

if (fscanf(fin, "%lf", &d) != EOF) dm[i][j] = d;

}

}

return dm;

}

double\*\* addMtrx(double\*\* A, double\*\* B, int row\_size, int col\_size) {

double\*\* D = (double\*\*)calloc(row\_size, sizeof(double\*));

for (int i = 0; i < row\_size; i++) {

D[i] = (double\*)calloc(col\_size, sizeof(double));

}

for (int i = 0; i < row\_size; i++) {

for (int j = 0; j < col\_size; j++) {

D[i][j] = A[i][j] + B[i][j];

}

}

return D;

}

double\*\* subtractMtrx(double\*\* A, double\*\* B, int row\_size, int col\_size) {

double\*\* E = (double\*\*)calloc(row\_size, sizeof(double\*));

for (int i = 0; i < row\_size; i++) {

E[i] = (double\*)calloc(col\_size, sizeof(double));

}

for (int i = 0; i < row\_size; i++) {

for (int j = 0; j < col\_size; j++) {

E[i][j] = A[i][j] - B[i][j];

}

}

return E;

}

double\*\* multiplyMtrx(double\*\* A, double\*\* B, int row\_size, int k\_size, int col\_size) {

double\*\* F = (double\*\*)calloc(row\_size, sizeof(double\*));

for (int i = 0; i < row\_size; i++) {

F[i] = (double\*)calloc(col\_size, sizeof(double));

}

for (int i = 0; i < row\_size; i++) {

for (int j = 0; j < col\_size; j++) {

for (int k = 0; k < k\_size; k++) {

F[i][j] += A[i][k] \* B[k][j];

}

}

}

return F;

}

/\* Matrix.h \*/

#ifndef MATRIX\_H

#define MATRIX\_H

#include <stdio.h>

#include <stdlib.h>

double\*\* fGetMtrx(FILE\* fin, int\* row\_size, int\* col\_size);

double\*\* addMtrx(double\*\* A, double\*\* B, int row\_size, int col\_size);

double\*\* subtractMtrx(double\*\* A, double\*\* B, int row\_size, int col\_size);

double\*\* multiplyMtrx(double\*\* A, double\*\* B, int row\_size, int k\_size, int col\_size);

void printMtrx(const char\* name, double\*\* mA, int row\_size, int col\_size);

void fprintMtrx(FILE\* fout, const char\* name, double\*\* mA, int row\_size, int col\_size);

double\*\* inverseMtrx(double\*\* mA, int size\_N);

#endif

// Matrix.cpp(2)

void printMtrx(const char\* name, double\*\* dm, int row\_size, int col\_size) {

unsigned char a6 = 0xA6, a1 = 0xA1, a2 = 0xA2;

unsigned char a3 = 0xA3, a4 = 0xA4, a5 = 0xA5;

printf(name);

printf("\n");

for (int i = 0; i < row\_size; i++) {

for (int j = 0; j < col\_size; j++) {

if ((i == 0) && (j == 0)) {

printf("%c%c%7.2lf", a6, a3, dm[i][j]);

}

else if ((i == 0) && j == (col\_size - 1)) {

printf("%7.2lf%c%c", dm[i][j], a6, a4);

}

else if ((i > 0) && (i < row\_size - 1) && (j == 0)) {

printf("%c%c%7.2lf", a6, a2, dm[i][j]);

}

else if ((i > 0) && (i < row\_size - 1) && (j == (col\_size - 1))) {

printf("%7.2lf%c%c", dm[i][j], a6, a2);

}

else if ((i == row\_size - 1) && (j == 0)) {

printf("%c%c%7.2lf", a6, a6, dm[i][j]);

}

else if ((i == row\_size - 1) && (j == (col\_size - 1))) {

printf("%7.2lf%c%c", dm[i][j], a6, a5);

}

else {

printf("%7.2lf", dm[i][j]);

}

if (col\_size == 1) {

if (i == 0)

printf("%c%c", a6, a4);

else if (i == row\_size - 1)

printf("%c%c", a6, a5);

else

printf("%c%c", a6, a2);

}

}

printf("\n");

}

}

void fprintMtrx(FILE\* fout, const char\* name, double\*\* dm, int row\_size, int col\_size) {

unsigned char a6 = 0xA6, a1 = 0xA1, a2 = 0xA2;

unsigned char a3 = 0xA3, a4 = 0xA4, a5 = 0xA5;

fprintf(fout, name);

fprintf(fout, "\n");

for (int i = 0; i < row\_size; i++) {

for (int j = 0; j < col\_size; j++) {

if ((i == 0) && (j == 0)) {

fprintf(fout, "%c%c%7.2lf", a6, a3, dm[i][j]);

}

else if ((i == 0) && j == (col\_size - 1)) {

fprintf(fout, "%7.2lf%c%c", dm[i][j], a6, a4);

}

else if ((i > 0) && (i < row\_size - 1) && (j == 0)) {

fprintf(fout, "%c%c%7.2lf", a6, a2, dm[i][j]);

}

else if ((i > 0) && (i < row\_size - 1) && (j == (col\_size - 1))) {

fprintf(fout, "%7.2lf%c%c", dm[i][j], a6, a2);

}

else if ((i == row\_size - 1) && (j == 0)) {

fprintf(fout, "%c%c%7.2lf", a6, a6, dm[i][j]);

}

else if ((i == row\_size - 1) && (j == (col\_size - 1))) {

fprintf(fout, "%7.2lf%c%c", dm[i][j], a6, a5);

}

else {

fprintf(fout, "%7.2lf", dm[i][j]);

}

if (col\_size == 1) {

if (i == 0)

fprintf(fout, "%c%c", a6, a4);

else if (i == row\_size - 1)

fprintf(fout, "%c%c", a6, a5);

else

fprintf(fout, "%c%c", a6, a2);

}

}

fprintf(fout, "\n");

}

}

// Matrix.cpp(3)

double\*\* inverseMtrx(double\*\* Mtrx, int size\_N) {

int j, k, p = 0;

double pivWeight, w;

double\*\* m\_aug, \*\* inv\_m;

m\_aug = (double\*\*)calloc(size\_N, sizeof(double\*));

inv\_m = (double\*\*)calloc(size\_N, sizeof(double\*));

for (int i = 0; i < size\_N; i++) {

m\_aug[i] = (double\*)calloc(size\_N \* 2, sizeof(double));

inv\_m[i] = (double\*)calloc(size\_N, sizeof(double));

}

for (j = 0; j < size\_N; j++) {

// 역행렬을 구하고싶은 행렬

for (k = 0; k < size\_N; k++)

m\_aug[j][k] = Mtrx[j][k];

// 단위 행렬

for (k = size\_N; k < size\_N \* 2; k++)

m\_aug[j][k] = (k == (j + size\_N)) ? 1.0 : 0.0;

}

for (p = 0; p < size\_N; p++) {

if (m\_aug[p][p] != 1.0) { // 해당 위치의 값이 이미 scaling되지 않은 경우

pivWeight = m\_aug[p][p];

m\_aug[p][p] = 1.0; // scaling

for (k = p + 1; k < size\_N \* 2; k++) {

m\_aug[p][k] /= pivWeight; // 이후의 값도 나누기

}

}

// diagonalize

for (j = 0; j < size\_N; j++) {

if ((j != p) && (m\_aug[j][p] != 0.0))

{

w = m\_aug[j][p];

m\_aug[j][p] = 0.0; // 해당 열에서 pivot 위치 이외의 자리에 0.0으로 소거

for (k = p + 1; k < 2 \* size\_N; k++) {

m\_aug[j][k] -= w \* m\_aug[p][k];

}

}

}

}

// 구한 역행렬을 생성한 2차원 배열에 복사

for (j = 0; j < size\_N; j++) {

for (k = 0; k < size\_N; k++) {

inv\_m[j][k] = m\_aug[j][k + size\_N];

}

}

return inv\_m;

}

#endif

테이블이(가) 표시된 사진

자동 생성된 설명 테이블이(가) 표시된 사진

자동 생성된 설명

좌 : 실행창

우 : 파일 출력

**(5) 행렬 연산 확인 완료**

**(7) mI는 단위행렬이다.**

1. **선형시스템의 해를 Gauss-Jordan 소거법으로 구하는 프로그램 작성.**

/\*

\* 파일명 : main.cpp

\* 목적 및 기본 기능

\* 주어진 회로를 키르히호프 법칙을 사용하여 첨가행렬로 구성하고

\* 가우스-조단 소거법을 통해 해를 구한다.

\* 작성자 : 김주환(21812158)

\* 작성일 : 2021년 5월 3일

\*/

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <math.h>

#include <stdlib.h>

#include "Matrix.h"

#include "GaussJordan.h"

const char\* augMtrx\_file = "ElectronicCircuit\_B\_5x6.txt";

int main(void) {

FILE\* fin, \* fout;

int size\_N, row\_size, col\_size;

double\*\* solution;

double\*\* augMtrx = NULL;

double\*\* augtemp = NULL;

double\*\* inv\_dm, \*\*mE;

int i, j, solExist = 1, error = 0;

double d;

fin = fopen(augMtrx\_file, "r");

if (fin == NULL) {

printf("Error in opening input.txt file (%s)!!\n", augMtrx\_file);

exit(-1);

}

fout = fopen("Output.txt", "w");

if (fout == NULL) {

printf("Error in creation of output.txt file !!\n");

exit(-1);

}

fscanf(fin, "%d", &size\_N);

augMtrx = fGetAugMtrx(fin, size\_N);

fprintMtrx(fout, "Augmented Matrix", augMtrx, size\_N, size\_N + 1);

printMtrx("Augmented Matrix", augMtrx, size\_N, size\_N + 1);

solution = (double\*\*)calloc(size\_N, sizeof(double\*));

mE = (double\*\*)calloc(size\_N, sizeof(double\*));

augtemp = (double\*\*)calloc(size\_N, sizeof(double\*));

for (int i = 0; i < size\_N; i++) {

solution[i] = (double\*)calloc(size\_N, sizeof(double));

mE[i] = (double\*)calloc(size\_N, sizeof(double));

augtemp[i] = (double\*)calloc(size\_N, sizeof(double));

}

// main.cpp(2)

for (int i = 0; i < size\_N; i++) {

for (int j = 0; j < size\_N + 1; j++) {

if (j == size\_N) {

mE[i][0] = augMtrx[i][j];

}

else {

augtemp[i][j] = augMtrx[i][j];

}

}

}

diagonalize\_FileOut(fout, (double\*\*)augMtrx, size\_N, &solExist);

if (solExist) {

fprintf(fout, "The solution of the given linear system:\n");

for (i = 0; i < size\_N; i++) {

solution[i][0] = augMtrx[i][size\_N];

fprintf(fout, " x[%d] : %4f\n", i, solution[i][0]);

printf(" x[%d] : %4f\n", i, solution[i][0]);

}

}

else {

printf("No unique solution\n");

fprintf(fout, "No unique solution\n");

}

// 검증

free(solution);

augMtrx = fGetAugMtrx(fin, size\_N);

solution = (double\*\*)calloc(size\_N, sizeof(double\*));

for (int i = 0; i < size\_N; i++) {

solution[i] = (double\*)calloc(size\_N, sizeof(double));

}

fprintf(fout, "Verification\n");

printf("Verification\n");

inv\_dm = inverseMtrx(augtemp, size\_N); // 역행렬

printMtrx((const char\*)"inv\_mAug", inv\_dm, size\_N, size\_N);

fprintMtrx(fout, (const char\*)"inv\_mAug", inv\_dm, size\_N, size\_N);

solution = multiplyMatrix(inv\_dm, mE, size\_N, size\_N, 1);

printMtrx((const char\*)"X = (A^-1) x B =", solution, size\_N, 1);

fprintMtrx(fout, (const char\*)"X = (A^-1) x B =", solution, size\_N, 1);

for (int i = 0; i < size\_N; i++)

free(augMtrx[i]);

free(augMtrx);

free(solution);

fclose(fin);

fclose(fout);

}

// GaussJordan.h

#ifndef GJ

#define GJ

#include <stdio.h>

#include <math.h>

#include "Matrix.h"

#define EPSILON 0.000001

double\*\* fGetAugMtrx(FILE\* fin, int size\_N);

void pivoting(double\*\* augMtrx, int size\_N, int p, int\* piv\_found);

void diagonalize\_FileOut(FILE\* fout, double\*\* augMtrx, int size\_N, int\* solExist);

double\*\* inverseMtrx(double\*\* Mtrx, int size\_N);

#endif

// Matrix.h

#ifndef M

#define M

#include <stdio.h>

double\*\* createDynamicDoubleMatrix(int row\_size, int col\_size);

double\*\* fGetMtrx(FILE\* fin, int\* row\_size, int\* col\_size);

void deleteDynMtrx(double\*\* dm, int row\_size, int col\_size);

void printMtrx(const char\* name, double\*\* dm, int size\_row, int size\_col);

void fprintMtrx(FILE\* fout, const char\* name, double\*\* dm, int row\_size, int col\_size);

double\*\* addMatrix(double\*\* dm1, double\*\* dm2, int row\_size, int col\_size);

double\*\* subtractMatrix(double\*\* dm1, double\*\* dm2, int row\_size, int col\_size);

double\*\* multiplyMatrix(double\*\* dm1, double\*\* dm2, int row\_size, int temp, int col\_size);

#endif

// Matrix.cpp(2)

// 행렬 데이터를 지정된 파일에 출력

void fprintMtrx(FILE\* fout, const char\* name, double\*\* dm, int row\_size, int col\_size) {

unsigned char a6 = 0xA6, a1 = 0xA1, a2 = 0xA2;

unsigned char a3 = 0xA3, a4 = 0xA4, a5 = 0xA5;

fprintf(fout, name);

fprintf(fout, "\n");

for (int i = 0; i < row\_size; i++) {

for (int j = 0; j < col\_size; j++) {

if ((i == 0) && (j == 0)) {

fprintf(fout, "%c%c%7.2lf", a6, a3, dm[i][j]);

}

else if ((i == 0) && j == (col\_size - 1)) {

fprintf(fout, "%7.2lf%c%c", dm[i][j], a6, a4);

}

else if ((i > 0) && (i < row\_size - 1) && (j == 0)) {

fprintf(fout, "%c%c%7.2lf", a6, a2, dm[i][j]);

}

else if ((i > 0) && (i < row\_size - 1) && (j == (col\_size - 1))) {

fprintf(fout, "%7.2lf%c%c", dm[i][j], a6, a2);

}

else if ((i == row\_size - 1) && (j == 0)) {

fprintf(fout, "%c%c%7.2lf", a6, a6, dm[i][j]);

}

else if ((i == row\_size - 1) && (j == (col\_size - 1))) {

fprintf(fout, "%7.2lf%c%c", dm[i][j], a6, a5);

}

else {

fprintf(fout, "%7.2lf", dm[i][j]);

}

if (col\_size == 1) {

if (i == 0)

fprintf(fout, "%c%c", a6, a4);

else if (i == row\_size - 1)

fprintf(fout, "%c%c", a6, a5);

else

fprintf(fout, "%c%c", a6, a2);

}

}

fprintf(fout, "\n");

}

}

// 행렬 덧셈

double\*\* addMatrix(double\*\* dm1, double\*\* dm2, int row\_size, int col\_size) {

double\*\* dm3 = createDynamicDoubleMatrix(row\_size, col\_size);

for (int i = 0; i < row\_size; i++) {

for (int j = 0; j < col\_size; j++) {

dm3[i][j] = dm1[i][j] + dm2[i][j];

}

}

return dm3;

}

// 행렬 뺄셈

double\*\* subtractMatrix(double\*\* dm1, double\*\* dm2, int row\_size, int col\_size) {

double\*\* dm3 = createDynamicDoubleMatrix(row\_size, col\_size);

for (int i = 0; i < row\_size; i++) {

for (int j = 0; j < col\_size; j++) {

dm3[i][j] = dm1[i][j] - dm2[i][j];

}

}

return dm3;

}

// 행렬 곱셈

double\*\* multiplyMatrix(double\*\* dm1, double\*\* dm2, int row\_size, int temp, int col\_size) {

double\*\* dm3 = createDynamicDoubleMatrix(row\_size, col\_size);

for (int i = 0; i < row\_size; i++) {

for (int j = 0; j < col\_size; j++) {

for (int k = 0; k < temp; k++) {

dm3[i][j] += dm1[i][k] \* dm2[k][j];

}

}

}

return dm3;

}

#endif

// Matrix.cpp

#ifndef M

#define M

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

// 원하는 크기의 2차원 배열 동적 할당

double\*\* createDynamicDoubleMatrix(int row\_size, int col\_size) {

double\*\* ppdm = NULL;

ppdm = (double\*\*)calloc(row\_size, sizeof(double\*));

for (int i = 0; i < row\_size; i++) {

ppdm[i] = (double\*)calloc(col\_size, sizeof(double));

}

return ppdm;

}

// 파일로부터 원하는 크기의 행렬 데이터 불러와서 저장

double\*\* fGetMtrx(FILE\* fin, int\* row\_size, int\* col\_size) {

double d = 0.0;

fscanf(fin, "%d %d", row\_size, col\_size);

double\*\* dm = createDynamicDoubleMatrix(\*row\_size, \*col\_size);

for (int i = 0; i < \*row\_size; i++) {

for (int j = 0; j < \*col\_size; j++) {

if (fscanf(fin, "%lf", &d) != EOF) dm[i][j] = d;

}

}

return dm;

}

// 생성한 2차원 동적 배열 해제

void deleteDynMtrx(double\*\* dm, int row\_size, int col\_size) {

for (int i = 0; i < row\_size; i++) free(dm[i]);

free(dm);

}

// 행렬 데이터를 실행창에 출력

void printMtrx(const char\* name, double\*\* dm, int row\_size, int col\_size) {

unsigned char a6 = 0xA6, a1 = 0xA1, a2 = 0xA2;

unsigned char a3 = 0xA3, a4 = 0xA4, a5 = 0xA5;

printf(name);

printf("\n");

for (int i = 0; i < row\_size; i++) {

for (int j = 0; j < col\_size; j++) {

if ((i == 0) && (j == 0)) {

printf("%c%c%7.2lf", a6, a3, dm[i][j]);

}

else if ((i == 0) && j == (col\_size - 1)) {

printf("%7.2lf%c%c", dm[i][j], a6, a4);

}

else if ((i > 0) && (i < row\_size - 1) && (j == 0)) {

printf("%c%c%7.2lf", a6, a2, dm[i][j]);

}

else if ((i > 0) && (i < row\_size - 1) && (j == (col\_size - 1))) {

printf("%7.2lf%c%c", dm[i][j], a6, a2);

}

else if ((i == row\_size - 1) && (j == 0)) {

printf("%c%c%7.2lf", a6, a6, dm[i][j]);

}

else if ((i == row\_size - 1) && (j == (col\_size - 1))) {

printf("%7.2lf%c%c", dm[i][j], a6, a5);

}

else {

printf("%7.2lf", dm[i][j]);

}

if (col\_size == 1) {

if (i == 0)

printf("%c%c", a6, a4);

else if (i == row\_size - 1)

printf("%c%c", a6, a5);

else

printf("%c%c", a6, a2);

}

}

printf("\n");

}

}

// GaussJordan.cpp(2)

// Diagonalize

for (j = 0; j < size\_N; j++) {

if ((j != p) && (augMtrx[j][p] != 0.0)) {

w = augMtrx[j][p];

augMtrx[j][p] = 0.0;

for (k = p + 1; k <= size\_N; k++) {

augMtrx[j][k] =

augMtrx[j][k] - w \* augMtrx[p][k];

}

}

}

}

}

// 역행렬 구하기

double\*\* inverseMtrx(double\*\* Mtrx, int size\_N) {

int j, k, p = 0;

double pivWeight, w;

double\*\* m\_aug, \*\* inv\_m;

m\_aug = (double\*\*)calloc(size\_N, sizeof(double\*));

inv\_m = (double\*\*)calloc(size\_N, sizeof(double\*));

for (int i = 0; i < size\_N; i++) {

m\_aug[i] = (double\*)calloc(size\_N \* 2, sizeof(double));

inv\_m[i] = (double\*)calloc(size\_N, sizeof(double));

}

for (j = 0; j < size\_N; j++) {

// 역행렬을 구하고싶은 행렬

for (k = 0; k < size\_N; k++)

m\_aug[j][k] = Mtrx[j][k];

// 단위 행렬

for (k = size\_N; k < size\_N \* 2; k++)

m\_aug[j][k] = (k == (j + size\_N)) ? 1.0 : 0.0;

}

for (p = 0; p < size\_N; p++) {

if (m\_aug[p][p] != 1.0) { // 해당 위치의 값이 이미 scaling되지 않은 경우

pivWeight = m\_aug[p][p];

m\_aug[p][p] = 1.0; // scaling

for (k = p + 1; k < size\_N \* 2; k++) {

m\_aug[p][k] /= pivWeight; // 이후의 값도 나누기

}

}

// diagonalize

for (j = 0; j < size\_N; j++) {

if ((j != p) && (m\_aug[j][p] != 0.0))

{

w = m\_aug[j][p];

m\_aug[j][p] = 0.0; // 해당 열에서 pivot 위치 이외의 자리에 0.0으로 소거

for (k = p + 1; k < 2 \* size\_N; k++) {

m\_aug[j][k] -= w \* m\_aug[p][k];

}

}

}

}

// 구한 역행렬을 생성한 2차원 배열에 복사

for (j = 0; j < size\_N; j++) {

for (k = 0; k < size\_N; k++) {

inv\_m[j][k] = m\_aug[j][k + size\_N];

}

}

return inv\_m;

}

#endif

// GaussJordan.h

#ifndef GJ

#define GJ

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <math.h>

#include <Windows.h>

#include "Matrix.h"

#define EPSILON 0.000001

// 첨가행렬 데이터 파일로부터 불러오기

double\*\* fGetAugMtrx(FILE\* fin, int size\_N) {

double d = 0.0;

double\*\* augMtrx = createDynamicDoubleMatrix(size\_N, size\_N + 1);

if (fin == NULL) {

printf("Error in getDoubleMatrixData() - file pointer is NULL !!\n");

exit(-1);

}

for (int m = 0; m < size\_N; m++)

for (int n = 0; n <= size\_N; n++) {

if (fscanf(fin, "%lf", &d) != EOF)

augMtrx[m][n] = d;

}

return augMtrx;

}

// pivoting 과정 수행

void pivoting(double\*\* augMtrx, int size\_N, int p, int\* piv\_found) {

double xmax = 0.0, xtemp;

int j, k, max\_row;

xmax = fabs(augMtrx[p][p]); // 해당 위치 계수의 절대값 저장

max\_row = p; // 해당 위치 저장

for (j = p + 1; j < size\_N; j++) {

if (fabs(augMtrx[j][p]) > xmax) { // 저장한 절대값보다 큰 값이 존재할 경우

xmax = fabs(augMtrx[j][p]); // 더 큰 절대값을 저장

max\_row = j; // 해당 위치 저장

}

}

if (fabs(xmax) < EPSILON) // 값이 0이거나 0에 근접한 경우

\*piv\_found = 0; // 해가 없음

else { // 그 외

\*piv\_found = 1; // 해가 있음

if (max\_row != p) {

for (k = p; k <= size\_N; k++) { // swap

xtemp = augMtrx[p][k];

augMtrx[p][k] = augMtrx[max\_row][k];

augMtrx[max\_row][k] = xtemp;

}

}

}

}

// 대각화 과정 수행

void diagonalize\_FileOut(FILE\* fout, double\*\* augMtrx, int size\_N, int\* solExist) {

int j, k, p = 0;

double pivWeight, w;

// Pivoting 그리고 Scaling

for (p = 0; (\*solExist) && (p < size\_N); p++) {

pivoting(augMtrx, size\_N, p, solExist);

if (\*solExist) { // 해가 존재

// 해당 위치의 원소 scaling

if (augMtrx[p][p] != 1.0) { // 이미 scaling된 값이 아닌 경우

pivWeight = augMtrx[p][p];

augMtrx[p][p] = 1.0; // scaling

for (k = p + 1; k <= size\_N; k++) {

augMtrx[p][k] /= pivWeight; // 이후의 값도 나누기

}

}

}

else { // 해가 없음

break;

}

텍스트이(가) 표시된 사진

자동 생성된 설명테이블이(가) 표시된 사진

자동 생성된 설명

좌 : 실행창

우 : 파일 출력

**(1) 키르히호프 법칙과 옴의 법칙을 통해 정리 가능**

**(3) 문제와 같이 검증 과정을 거친 결과를 보면 가우스-조단 소거법과 역행렬을 이용한 해를 구하는 방법은 같은 결과를 가짐을 알 수 있다.**