프로그래밍언어

8. 행렬의 연산, 선형시스템, Gauss-Jordan 소거법

교수 김 영 탁

영남대학교 정보통신공학과

(Tel : +82-53-810-2497; Fax : +82-53-810-4742

http://antl.yu.ac.kr/; E-mail : ytkim@yu.ac.kr)

Outline

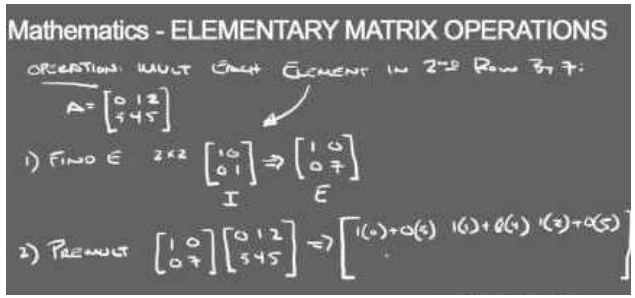
- 2차원 동적 배열 기반 행렬 (Matrix)
- 행렬의 연산
- 첨가행렬과 Gauss-Jordan 소거법을 사용한 선형시스템 해법
- Gauss-Jordan 소거법을 사용한 역행렬 계산 및 선형시스템 해법

행렬 (Matrix)과 관련 기본 연산

2차원 배열과 행렬 (Matrix)

◆ 행렬(matrix)는 자연과학에서 많은 문제를 해결하는데 사용

$$A = \begin{bmatrix} 2 & 3 & 0 \\ 8 & 9 & 1 \\ 7 & 0 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 & 7 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$



행렬 (Matrix) 관련 기본 함수

```
/* 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 (1) */
#include "Matrix.h"
#include <stdio.h>
#include <stdlib.h>

/* Dynamic Creation of Double Matrix*/
double** createDynamicDoubleMatrix(double** dM, int row_size, int col_size)
{
        dM = (double**)calloc(sizeof(double*), row_size);
        for (int m = 0; m < row_size; m++)
        {
                dM[m] = (double*)calloc(sizeof(double), col_size);
        }
        return dM;
}

void deleteDoubleMtrx(double **dM, int row_size, int col_size)
{
    for (int r = 0; r < row_size; r++)
    {
        free(dM[r]);
    }
    free(dM);
}
```

```
/* Matrix.cpp (2) */

/* Matrix File input */
double **fGetMtrx(FILE* fin, int *pRow_size, int *pCol_size)
{
   double data = 0.0;
   double** dM;
   if (fin == NULL)
   {
      printf("Error in getDoubleMatrixData() - file pointer is NULL !!\n");
      return NULL;
   }

   fscanf(fin, "%d %d", pRow_size, pCol_size);
   dM = (double**)calloc(*pRow_size, sizeof(double*));
   for (int r = 0; r < *pRow_size; r++)
   {
      dM[r] = (double*)calloc(*pCol_size, sizeof(double));
   }
   for (int r = 0; r < *pRow_size; r++)
      for (int c = 0; c < *pCol_size; c++)
      {
         if (fscanf(fin, "%lf", &data) != EOF)
            dM[r][c] = data;
      }
   return dM;
}
```

```
/** Matrix.cpp (3) */

void printMtrx(const char *name, double **mA, int row_size, int col_size)
{
   unsigned char a6 = 0xA6, a1 = 0xA1, a2 = 0xA2;
   unsigned char a3 = 0xA3, a4 = 0xA4, a5 = 0xA5;

   printf("%s = \n", name);
   for (int r = 0; r < row_size; r++) {
      for (int c = 0; c < col_size; c++)
      {
         if ((r == 0) && (c == 0))
            printf("%c%c%8.2lf", a6, a3, mA[r][c]);
         else if ((r == 0) && c == (col_size - 1))
            printf("%8.2lf%c%c", mA[r][c], a6, a4);
         else if ((r > 0) && (r < row_size - 1) && (c == 0))
            printf("%c%c%8.2lf", a6, a2, mA[r][c]);
         else if ((r > 0) && (r < row_size - 1) && (c == (col_size - 1)))
            printf("%8.2lf%c%c", mA[r][c], a6, a2);
         else if ((r == (row_size - 1)) && (c == 0))
            printf("%c%c%8.2lf", a6, a6, mA[r][c]);
         else if ((r == (row_size - 1)) && (c == (col_size - 1)))
            printf("%8.2lf%c%c", mA[r][c], a6, a5);
         else
            printf("%8.2lf", mA[r][c]);
      }
      printf("\n");
   }
}
```

/** Matrix.cpp (4) */

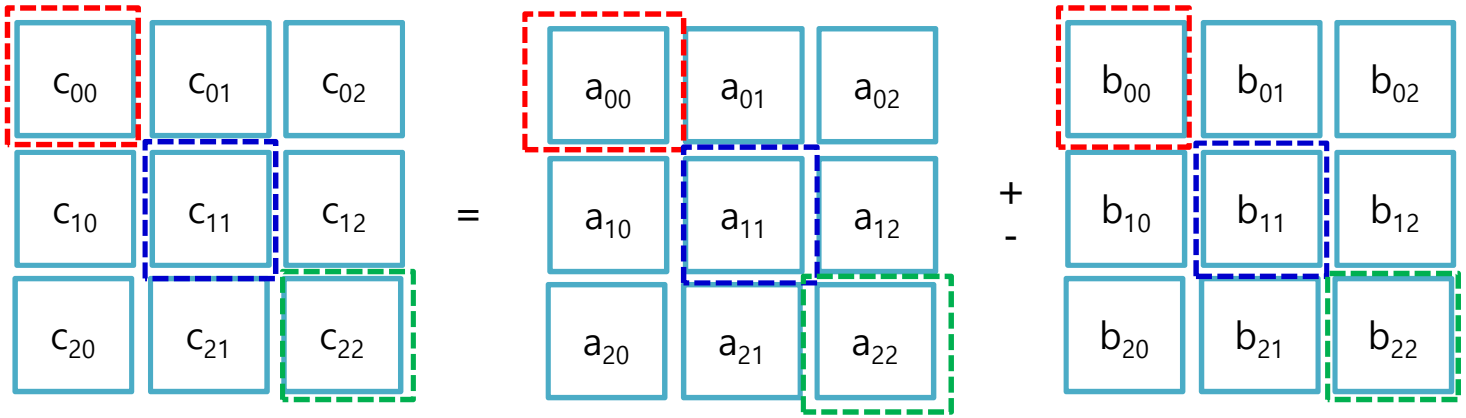
```
/* Matrix File Output */
void fprintMtrx(FILE* fout, const char* name, double** mA, int row_size, int col_size)
{
   unsigned char a6 = 0xA6, a1 = 0xA1, a2 = 0xA2;
   unsigned char a3 = 0xA3, a4 = 0xA4, a5 = 0xA5;

   fprintf(fout, "%s = \n", name);
   for (int r = 0; r < row_size; r++) {
      for (int c = 0; c < col_size; c++)
      {
         if ((r == 0) && (c == 0))
            fprintf(fout, "%c%c%8.2lf", a6, a3, mA[r][c]);
         else if ((r == 0) && c == (col_size - 1))
            fprintf(fout, "%8.2lf%c%c", mA[r][c], a6, a4);
         else if ((r > 0) && (r < row_size - 1) && (c == 0))
            fprintf(fout, "%c%c%8.2lf", a6, a2, mA[r][c]);
         else if ((r > 0) && (r < row_size - 1) && (c == (col_size - 1)))
            fprintf(fout, "%8.2lf%c%c", mA[r][c], a6, a2);
         else if ((r == (row_size - 1)) && (c == 0))
            fprintf(fout, "%c%c%8.2lf", a6, a6, mA[r][c]);
         else if ((r == (row_size - 1)) && (c == (col_size - 1)))
            fprintf(fout, "%8.2lf%c%c", mA[r][c], a6, a5);
         else
            fprintf(fout, "%8.2lf", mA[r][c]);
      }
      fprintf(fout, "\n");
   }
}
```

출력 결과	확장 완성형 코드	사용 방법
─	0xa6, 0xa1	printf(" %c%c", 0xa6, 0xa1)
│	0xa6, 0xa2	printf(" %c%c", 0xa6, 0xa2)
┌	0xa6, 0xa3	printf(" %c%c", 0xa6, 0xa3)
┐	0xa6, 0xa4	printf(" %c%c", 0xa6, 0xa4)
┘	0xa6, 0xa5	printf(" %c%c", 0xa6, 0xa5)
└	0xa6, 0xa6	printf(" %c%c", 0xa6, 0xa6)

```
┌  1.00  2.00  3.00  4.00  5.00  6.00 ┐
│  2.00  3.00  4.00  5.00  1.00  3.00 │
│  3.00  2.00  5.00  3.00  2.00  1.00 │
│  4.00  3.00  2.00  7.00  2.00  5.00 │
└  5.00  4.00  3.00  2.00  9.00  1.00 ┘
```

2차원 행렬의 덧셈과 뺄셈

◆ 행렬의 덧셈과 뺄셈

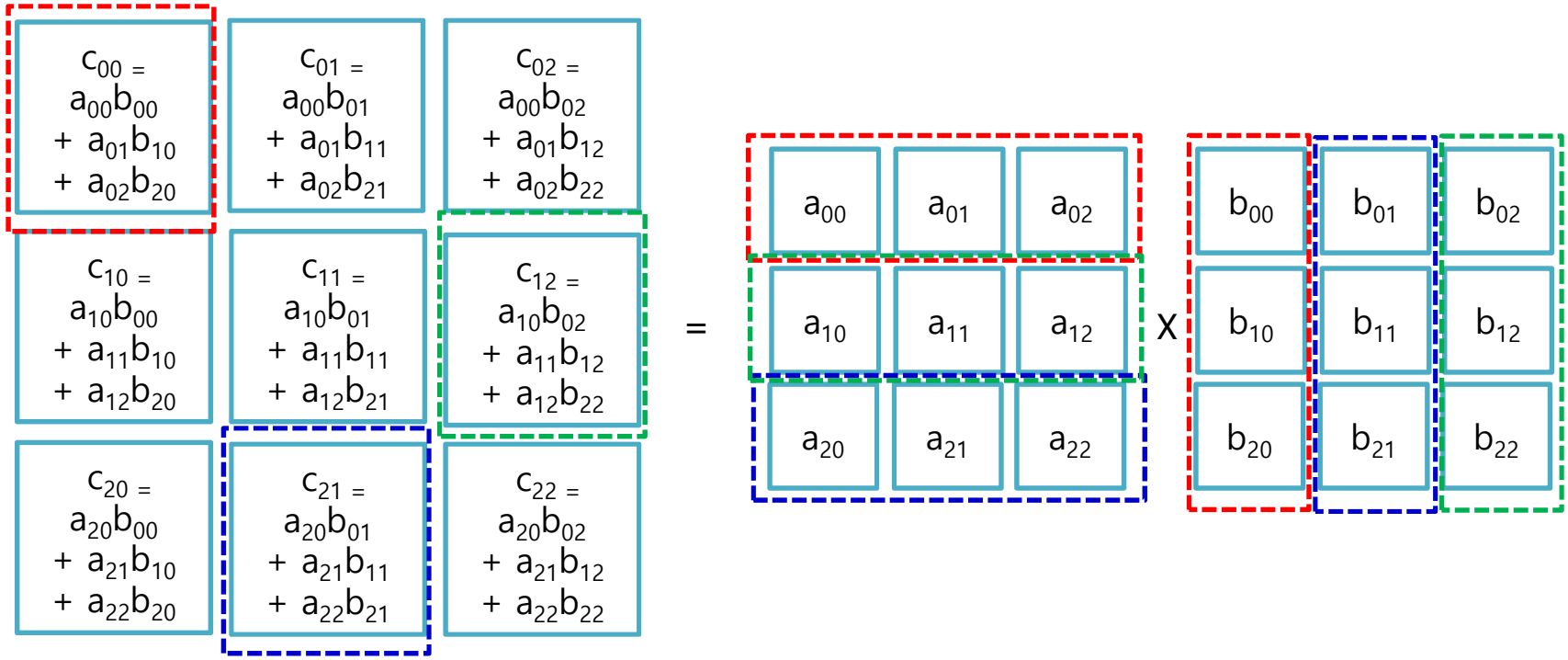


```
/** Matrix.cpp (5) */

/* Matrix Addition */
double **addMtrx(double **A, double **B, int row_size, int col_size)
{
   double** R;

   R = (double**)calloc(row_size, sizeof(double*));
   for (int r = 0; r < row_size; r++)
   {
      R[r] = (double*)calloc(col_size, sizeof(double));
   }
   for (int r=0; r< row_size; r++)
      for (int c = 0; c < col_size; c++)
      {
         R[r][c] = A[r][c] + B[r][c];
      }
   return R;
}
```

```
/** Matrix.cpp (6) */

/* Matrix Subtraction */
double** subtractMtrx(double** A, double** B, int row_size, int col_size)
{
   double** R;

   R = (double**)calloc(row_size, sizeof(double*));
   for (int r = 0; r < row_size; r++)
   {
      R[r] = (double*)calloc(col_size, sizeof(double));
   }

   for (int r = 0; r < row_size; r++)
      for (int c = 0; c < col_size; c++)
      {
         R[r][c] = A[r][c] - B[r][c];
      }
   return R;
}
```

행렬의 곱셈

◆ 행렬의 곱셈 계산



```
/** Matrix.cpp (5) */

/* Matrix Mutiplication */
double **multiplyMtrx(double** A, double** B, int row_size,  int k_size,
     int col_size)
{
   double** R;

   R = (double**)calloc(row_size, sizeof(double*));
   for (int r = 0; r < row_size; r++)
   {
      R[r] = (double*)calloc(col_size, sizeof(double));
   }

   for (int r = 0; r < row_size; r++)
      for (int c = 0; c < col_size; c++)
      {
         R[r][c] = 0.0;
         for (int k = 0; k < k_size; k++)
         {
            R[r][c] += A[r][k] * B[k][c];
         }
      }
   return R;
}
```

행렬 (Matrix)의 연산 기능 시험

행렬 연산 기능 시험

```
/** main_DynamicTwoDimensionalArray.cpp  (1) */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "Matrix.h"

void checkAddress_2DimArray_for_Matrix();
void test_2D_DynamicArray_FileIO();
void test_MatrixAdditionSubtraction();
void test_MatrixMultiplication();
```

```
/** main_DynamicTwoDimensionalArray.cpp  (2) */
int main(void)
{
        int menu;

        while (1)
        {
                printf("Testing Matrix Operations with 2-Dimensional Dynamic Array\n");
                printf(" 1: Check addresses of 2-Dim array for Matrix\n");
                printf(" 2: Test 2-D Dynamic Array Creation for Matrix with File I/O\n");
                printf(" 3: Test Matrix Addition, Subtraction\n");
                printf(" 4: Test Matrix Multiplication\n");
                printf("-1: Quit\n");
                printf("Input menu (-1 to quit) : ");
                scanf("%d", &menu);
                if (menu == -1)
                        break;
                printf("\n");
                switch (menu)
                {
                case 1:
                        checkAddress_2DimArray_for_Matrix();
                        break;
                case 2:
                        test_2D_DynamicArray_FileIO();
                        break;
                case 3:
                        test_MatrixAdditionSubtraction();
                        break;
                case 4:
                        test_MatrixMultiplication();
                        break;
```

```
/** main_DynamicTwoDimensionalArray.cpp  (3) */

                default:
                        break;
                }
        }  // end while

}

void checkAddress_2DimArray_for_Matrix()
{
        int m[3][3] = { 10, 20, 30, 40, 50, 60, 70, 80, 90 };

        printf("m       = %p\n", m);
        printf("m[0]    = %p\n", m[0]);
        printf("m[1]    = %p\n", m[1]);
        printf("m[2]    = %p\n", m[2]);
        printf("&m[0][0] = %p\n", &m[0][0]);
        printf("&m[1][0] = %p\n", &m[1][0]);
        printf("&m[2][0] = %p\n", &m[2][0]);
        printf("\n");
}
```

```
Testing Matrix Operations with 2-Dimensional Dynamic Array
 1: Check addresses of 2-Dim array for Matrix
 2: Calculate total average of 2-Dim array
 3: Test 2-D Dynamic Array Creation for Matrix with File I/O
 4: Test Matrix Addition, Subtraction
 5: Test Matrix Multiplicaton
-1: Quit
Input menu (-1 to quit) : 1

m       = 002EF7B8
m[0]    = 002EF7B8
m[1]    = 002EF7C4
m[2]    = 002EF7D0
&m[0][0] = 002EF7B8
&m[1][0] = 002EF7C4
&m[2][0] = 002EF7D0
```

```
Testing Matrix Operations with 2-Dimensional Dynamic Array
 1: Check addresses of 2-Dim array for Matrix
 2: Calculate total average of 2-Dim array
 3: Test 2-D Dynamic Array Creation for Matrix with File I/O
 4: Test Matrix Addition, Subtraction
 5: Test Matrix Multiplicaton
-1: Quit
Input menu (-1 to quit) : 3

Input Matrix_A ( 5 x 6) :
┌    1.00    2.00    3.00    4.00    5.00    6.00 ┐
|    2.00    3.00    4.00    5.00    1.00    3.00 |
|    3.00    2.00    5.00    3.00    2.00    1.00 |
|    4.00    3.00    2.00    7.00    2.00    5.00 |
└    5.00    4.00    3.00    2.00    9.00    1.00 ┘

Input Matrix_B ( 6 x 5) :
┌   13.00   15.50   17.00   14.00   15.00 ┐
|   11.50   22.00   23.00   24.00   25.00 |
|   21.00   20.50   33.00   32.00   35.00 |
|   33.00   32.00   37.50   44.00   43.00 |
|   42.00   47.00   42.00   49.50   55.00 |
└   54.00   53.00   52.00   59.00   51.20 ┘
```

```
/** main_DynamicTwoDimensionalArray.cpp  (4) */

void test_2D_DynamicArray_FileIO()
{
        const char *matrixDataFile = "mtrxInputData.txt";
        FILE *fin;
        int a_row_size, a_col_size;
        int b_row_size, b_col_size;
        double **dMA, **dMB;

        fin = fopen(matrixDataFile, "r");
        if (fin == NULL)
        {
                printf("Error in opening input.txt file !!\n");
                exit(-1);
        }

        dMA = fGetMtrx(fin, &a_row_size, &a_col_size);
        printf("Input Matrix_A ( %d x %d) : \n", a_row_size, a_col_size);
        printMtrx("dMA", dMA, a_row_size, a_col_size);
        printf("\n");

        dMB = fGetMtrx(fin, b_row_size, b_col_size);
        printf("Input Matrix_B ( %d x %d) : \n", b_row_size, b_col_size);
        printMtrx('dMB", dMB, b_row_size, b_col_size);
        printf("\n");

        deleteDoubleMtrx(dMA, a_row_size, a_col_size);
        deleteDoubleMtrx(dMB, b_row_size, b_col_size);
        fclose(fin);
}
```

```
/** main_DynamicTwoDimensionalArray.cpp  (5) */

void test_MatrixAdditionSubtraction()
{
        const char *matrixDataFile = "mtrx_nxn_InputData.txt";
        FILE *fin;

        double **dMA, **dMB, **dMC, **dMD;
        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;

        fin = fopen(matrixDataFile, "r");
        if (fin == NULL)
        {
                printf("Error in opening input.txt file !!\n");
                exit(-1);
        }

        dMA = fGetMtrx(fin, &a_row_size, &a_col_size);
        printf("Input Matrix_A ( %d x %d) : \n", a_row_size, a_col_size);
        printMtrx("dMA", dMA, a_row_size, a_col_size);
        printf("\n");

        dMB = fGetMtrx(fin, &b_row_size, &b_col_size);
        printf("Input Matrix_B ( %d x %d) : \n", b_row_size, b_col_size);
        printMtrx("dMB", dMB, b_row_size, b_col_size);
        printf("\n");
```

```
Input menu (-1 to quit) : 4

Input Matrix_A ( 6 x 6) :
┌   1.0000   2.0000   3.0000   4.0000   5.0000   6.0000 ┐
|   2.0000   3.0000   4.0000   5.0000   1.0000   3.0000 |
|   3.0000   2.0000   5.0000   3.0000   2.0000   1.0000 |
|   4.0000   3.0000   2.0000   7.0000   2.0000   5.0000 |
|   5.0000   4.0000   3.0000   2.0000   9.0000   1.0000 |
└   6.0000   7.0000   8.0000   9.0000  10.0000  11.0000 ┘

Input Matrix_B ( 6 x 6) :
┌  13.0000  15.5000  17.0000  14.0000  15.0000  16.0000 ┐
|  11.5000  22.0000  23.0000  24.0000  25.0000  26.0000 |
|  21.0000  20.5000  33.0000  32.0000  35.0000  36.0000 |
|  33.0000  32.0000  37.5000  44.0000  43.0000  42.0000 |
|  42.0000  47.0000  42.0000  49.5000  55.0000  60.0000 |
└  54.0000  53.0000  52.0000  59.0000  51.2000  70.0000 ┘

Matrix_C (6 x 6) = Matrix_A + Matrix_B :
┌  14.0000  17.5000  20.0000  18.0000  20.0000  22.0000 ┐
|  13.5000  25.0000  27.0000  29.0000  26.0000  29.0000 |
|  24.0000  22.5000  38.0000  35.0000  37.0000  37.0000 |
|  37.0000  35.0000  39.5000  51.0000  45.0000  47.0000 |
|  47.0000  51.0000  45.0000  51.5000  64.0000  61.0000 |
└  60.0000  60.0000  60.0000  68.0000  61.2000  81.0000 ┘

Matrix_D (6 x 6) = Matrix_A - Matrix_B :
┌ -12.0000 -13.5000 -14.0000 -10.0000 -10.0000 -10.0000 ┐
|  -9.5000 -19.0000 -19.0000 -19.0000 -24.0000 -23.0000 |
| -18.0000 -18.5000 -28.0000 -29.0000 -33.0000 -35.0000 |
| -29.0000 -29.0000 -35.5000 -37.0000 -41.0000 -37.0000 |
| -37.0000 -43.0000 -39.0000 -47.5000 -46.0000 -59.0000 |
└ -48.0000 -46.0000 -44.0000 -50.0000 -41.2000 -59.0000 ┘
```

```
/** main_DynamicTwoDimensionalArray.cpp  (6) */

        if ((a_row_size != b_row_size) || (a_col_size != b_col_size))
        {
                printf("Error in input matrix dimension: row_size and/or col_size are not equal !!\n");
                deleteDoubleMtrx(dMA, a_row_size, a_col_size);
                deleteDoubleMtrx(dMB, b_row_size, b_col_size);
                fclose(fin);
                return;
        }
        // MC = MA + MB
        c_row_size = a_row_size;
        c_col_size = b_col_size;
        dMC = addMatrix(dMA, dMB, c_row_size, c_col_size);
        printf("Matrix_C (%d x %d) = Matrix_A + Matrix_B : \n", c_row_size, c_col_size);
        printMtrx("dMC", dMC, c_row_size, c_col_size);
        printf("\n");

        // MD = MA - MB
        d_row_size = a_row_size;
        d_col_size = b_col_size;
        dMD = subtractMatrix(dMA, dMB, d_row_size, d_col_size);
        printf("Matrix_D (%d x %d) = Matrix_A - Matrix_B : \n", d_row_size, d_col_size);
        printMtrx("dMD", dMD, d_row_size, d_col_size);
        printf("\n");

        deleteDoubleMtrx(dMA, a_row_size, a_col_size);
        deleteDoubleMtrx(dMB, b_row_size, b_col_size);
        deleteDoubleMtrx(dMC, c_row_size, c_col_size);
        deleteDoubleMtrx(dMD, d_row_size, d_col_size);
        fclose(fin);
}
```

```
/** main_DynamicTwoDimensionalArray.cpp  (8) */

void test_MatrixMultiplication(FILE *fout)
{
        const char *matrixDataFile = "mtrxInputData.txt";
        FILE *fin;

        int a_row_size, a_col_size;
        int b_row_size, b_col_size;
        int c_row_size, c_col_size;
        double **dMA, **dMB, **dMC;

        fin = fopen(matrixDataFile, "r");
        if (fin == NULL)
        {
                printf("Error in opening input.txt file !!\n");
                exit(-1);
        }
        dMA = fGetMtrx(fin, &a_row_size, &a_col_size);
        printf("Input Matrix_A ( %d x %d) : \n", a_row_size, a_col_size);
        printMtrx("dMA", dMA, a_row_size, a_col_size);
        printf("\n");

        dMB = fGetMtrx(fin, &b_row_size, &b_col_size);
        printf("Input Matrix_B ( %d x %d) : \n", b_row_size, b_col_size);
        printMtrx('dMB", dMB, b_row_size, b_col_size);
        printf("\n");
```

```
Testing Matrix Operations with 2-Dimensional Dynamic Array
 1: Check addresses of 2-Dim array for Matrix
 2: Calculate total average of 2-Dim array
 3: Test 2-D Dynamic Array Creation for Matrix with File I/O
 4: Test Matrix Addition, Subtraction
 5: Test Matrix Multiplicaton
-1: Quit
Input menu (-1 to quit) : 5

Input Matrix_A ( 5 x 6) :
┌    1.00    2.00    3.00    4.00    5.00    6.00 ┐
|    2.00    3.00    4.00    5.00    1.00    3.00 |
|    3.00    2.00    5.00    3.00    2.00    1.00 |
|    4.00    3.00    2.00    7.00    2.00    5.00 |
└    5.00    4.00    3.00    2.00    9.00    1.00 ┘

Input Matrix_B ( 6 x 5) :
┌   13.00   15.50   17.00   14.00   15.00 ┐
|   11.50   22.00   23.00   24.00   25.00 |
|   21.00   20.50   33.00   32.00   35.00 |
|   33.00   32.00   37.50   44.00   43.00 |
|   42.00   47.00   42.00   49.50   55.00 |
└   54.00   53.00   52.00   59.00   51.20 ┘

Matrix_C (5 x 5) = Matrix_A x Matrix_B :
┌  765.00  802.00  834.00  935.50  924.20 ┐
|  513.50  545.00  620.50  674.50  668.60 |
|  404.00  436.00  510.50  540.00  560.20 |
|  713.50  752.00  809.50  894.00  872.00 |
└  672.00  767.00  781.00  854.50  912.20 ┘
```

```
/** main_DynamicTwoDimensionalArray.cpp  (9) */

        if (a_col_size != b_row_size)
        {
                printf("Error in input matrix dimension: a_col_size and b_row_size are not equal !!\n");
                deleteDoubleMtrx(dMA, a_row_size, a_col_size);
                deleteDoubleMtrx(dMB, b_row_size, b_col_size);
                fclose(fin);
                return;
        }

        // MC = MA x MB
        c_row_size = a_row_size;
        c_col_size = b_col_size;
        dMC = multiplyMatrix(dMA, dMB, a_row_size, a_col_size, b_col_size);
        printf("Matrix_C (%d x %d) = Matrix_A x Matrix_B : \n",
            c_row_size, c_col_size);
        printMtrx("dMC", dMC, c_row_size, c_col_size);
        printf("\n");

        deleteDoubleMtrx(dMA, a_row_size, a_col_size);
        deleteDoubleMtrx(dMB, b_row_size, b_col_size);
        deleteDoubleMtrx(dMC, c_row_size, c_col_size);

        fclose(fin);
}
```

```
Testing Matrix Operations with 2-Dimensional Dynamic Array
 1: Check addresses of 2-Dim array for Matrix
 2: Calculate total average of 2-Dim array
 3: Test 2-D Dynamic Array Creation for Matrix with File I/O
 4: Test Matrix Addition, Subtraction
 5: Test Matrix Multiplicaton
-1: Quit
Input menu (-1 to quit) : 5

Input Matrix_A ( 5 x 6) :
┌    1.00    2.00    3.00    4.00    5.00    6.00 ┐
|    2.00    3.00    4.00    5.00    1.00    3.00 |
|    3.00    2.00    5.00    3.00    2.00    1.00 |
|    4.00    3.00    2.00    7.00    2.00    5.00 |
└    5.00    4.00    3.00    2.00    9.00    1.00 ┘

Input Matrix_B ( 6 x 5) :
┌   13.00   15.50   17.00   14.00   15.00 ┐
|   11.50   22.00   23.00   24.00   25.00 |
|   21.00   20.50   33.00   32.00   35.00 |
|   33.00   32.00   37.50   44.00   43.00 |
|   42.00   47.00   42.00   49.50   55.00 |
└   54.00   53.00   52.00   59.00   51.20 ┘

Matrix_C (5 x 5) = Matrix_A x Matrix_B :
┌  765.00  802.00  834.00  935.50  924.20 ┐
|  513.50  545.00  620.50  674.50  668.60 |
|  404.00  436.00  510.50  540.00  560.20 |
|  713.50  752.00  809.50  894.00  872.00 |
└  672.00  767.00  781.00  854.50  912.20 ┘
```

선형시스템 해 산출을 위한
첨가행렬과 Gauss-Jordan 소거법

선형 시스템 (Linear System)과 연립 방정식

◆ 선형시스템과 연립방정식의 예

- ***A* × *X* = *B***

$$\begin{aligned} x_1 &+ x_2 &+ x_3 &= 4 \\ 2x_1 &+ 3x_2 &+ x_3 &= 9 \\ x_1 &- x_2 &- x_3 &= -2 \end{aligned}$$

계수행렬 (coefficient matrix)

미지수: Solutions to be calculated

- $A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 3 & 1 \\ 1 & -1 & -1 \end{bmatrix}$, $X = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$, $B = \begin{bmatrix} 4 \\ 9 \\ -2 \end{bmatrix}$

◆ 선형 시스템의 해 (solution) 산출

- 첨가행렬 (augmented matrix)과 Gauss-Jordan 소거법
- 계수 행렬 *A*의 역행렬 (A^{-1}) 계산 => $X = A^{-1} \times B$

첨가 행렬 (Augmented Matrix)

◆ 연립방정식

- ***A* × *X* = *B***

$$\begin{array}{rcrcrcl} x_1 & + & x_2 & + & x_3 & = & 4 \\ 2x_1 & + & 3x_2 & + & x_3 & = & 9 \\ x_1 & - & x_2 & - & x_3 & = & -2 \end{array}$$

x_1, x_2, x_3 : 미지수

◆ 첨가 행렬 (augmented matrix)

계수행렬(coefficient matrix)

- $\left[\begin{array}{ccc|c} a_{0,0} & a_{0,1} & a_{0,2} & b_0 \\ a_{1,0} & a_{1,1} & a_{1,2} & b_1 \\ a_{2,0} & a_{2,1} & a_{2,2} & b_2 \end{array}\right]$

상수

- $\left[\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 2 & 3 & 1 & 9 \\ 1 & -1 & -1 & -2 \end{array}\right]$

선형 연립 방정식의 특성

◆ 선형 연립 방정식의 기본 특성

- 선형 연립 방정식의 첨가 행렬 (augmented matrix)에 다음과 같은 연산을 수행하여도 원래의 선형 연립 방정식 특성을 유지하며, 동일한 해 (solution)를 계산할 수 있다:
  - 첨가행렬의 어떤 행 (row)의 좌우 항을 0이 아닌 상수값으로 곱하기

R_1	:	x_1	+	x_2	+	x_3	= 4
$3R_1$	:	$3x_1$	+	$3x_2$	+	$3x_3$	= 12

  - 첨가행렬의 어떤 행에 다른 행을 어떤 상수값으로 곱셈연산을 한 결과를 더하기

R_1	:	x_1	+	x_2	+	x_3	= 4
R_2	:	$2x_1$	+	$3x_2$	+	x_3	= 9
R_3	:	x_1	-	x_2	-	x_3	= -2

R_1	:	x_1	+	x_2	+	x_3	= 4
R_2	:	$2x_1$	+	$3x_2$	+	x_3	= 9
$R_3 - R_1$	:			$-2x_2$	-	$2x_3$	= -6

  - 첨가행렬의 어떤 두 행의 자리를 교환하기 (swap)

R_2	:	$2x_1$	+	$3x_2$	+	x_3	= 9
R_1	:	x_1	+	x_2	+	x_3	= 4
R_3	:	x_1	-	x_2	-	x_3	= -2

Gauss-Jordan 소거법

◆ Gauss-Jordanian Elimination

- n개의 선형 방정식으로 표현되는 선형 시스템을 대각선 위 부분 값만 존재하는 삼각 행렬로 변환
- 삼각 행렬에서 대각선 밑의 각 원소들의 값 (계수)은 0으로 설정됨
- 삼각 행렬에서 대각선의 원소들의 계수가 1로 설정될 수 있도록 scaling

$$\begin{aligned} x_1 + x_2 + x_3 &= 4 \\ x_2 - x_3 &= 1 \\ x_3 &= 1 \end{aligned}$$

- 후방대입 (back substitution)을 사용하여 각 원소의 값을 추출
  - $x_3 = 1$
  - $x_2 - x_3 = 1 \quad when \quad x_3 = 1; \Rightarrow \quad x_2 = 2$
  - $x_1 + x_2 + x_3 = 4 \quad when \quad x_3 = 1 \text{ and } x_2 = 2; \quad \Rightarrow x_1 = 1$

선형연립방정식의 해 산출을 위한 Gauss-Jordan 소거법

- 주어진 선형 시스템에 대한 첨가행렬 정리
- 첨가행렬의 계수 부분을 단위 행렬로 변환 하도록
  p = 0 ~ N-1에 대하여 Gauss-Jordan 소거 연산 반복 수행
  - **pivoting**: 아직 p번째 이후의 행 (row) 들 중에서 p번째 원소의 절대값이 가장 큰 행을 p번째 행으로 선정하고, 필요한 경우 swap
  - **scaling**: 선정된 p번째 행의 $a_{p,p}$원소 (p-번째 pivot 원소) 값이 1.0되도록 계수 조정
  - **diagonalizing**: p번째 열의 $a_{p,p}$원소 (p-번째 pivot 원소) 값만 1.0이 되게 하고, 나머지 행들의 $a_{i,p}$원소 들은 모두 0.0이 되도록 소거 연산 수행
- 첨가행렬의 계수 부분을 단위 행렬로 변환
  - $\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & y_0 \\ a_{1,0} & a_{1,1} & a_{1,2} & y_1 \\ a_{2,0} & a_{2,1} & a_{2,2} & y_2 \end{bmatrix}$ $\Rightarrow$ pivoting/scaling/diagonalization 반복 수행

$$\Rightarrow \begin{bmatrix} 1 & 0 & 0 & y_0'' \\ 0 & 1 & 0 & y_1'' \\ 0 & 0 & 1 & y_2'' \end{bmatrix}$$

Pivoting

◆ Pivoting

- Gauss-Jordan 소거법 연산 과정에서 컴퓨터 계산의 반올림 처리 오류 발생을 최소화 하기 위하여 pivot 행을 선정할 때 항상 남아 있는 행들 중에서 pivot 열 위치의 원소 절대값이 가장 큰 행을 찾고, 이를 pivot 위치의 행과 교체 (swap)
- 만약 남아 있는 행들 중에서 pivot 열 위치의 원소값이 가장 큰 pivot 대상 원소 절대값이 0이거나 0에 근접한 경우, 오차 발생 때문에 정확한 해를 구할 수 없는 상태가 되며, 이 경우 단일 해를 구할 수 없는 것으로 판단함

Scaling and Diagonalization

◆ p-번째 pivot 행에 대한 Scaling 수행

- 첨가행렬에서 pivot $augMtrx_{p,p}$ 이 선정되면, 그 p-번째 pivot 원소 값이 1.0이 되도록 p-번째 행의 전체 원소들을 $pivWeight = augMtrx_{p,p}$ 값으로 나누어 줌

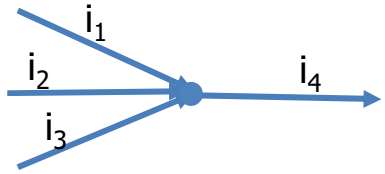
◆ Diagonalization

- p-번째 열에서 1.0으로 scaling된 $augMtrx_{p,p}$ 원소를 제외한 위, 아래 $augMtrx_{i,p}$ 원소들을 0.0으로 변경하도록 소거 연산 수행
- $augMtrx_{0,p}$ ~ $augMtrx_{p-1,p}$, $augMtrx_{p+1,p}$ ~ $augMtrx_{n-1,p}$ 가 0.0이 되도록 p-번째 행에 $augMtrx_{i,p}$ 값을 곱하여 i-번째 행에 뺄셈 연산

전자회로 분석을 위한 연립방정식 정리
- 키르히호프 (Kirchhoff) 전류, 전압 법칙

◆ 키르히호프 전류의 법칙 (KCL) –

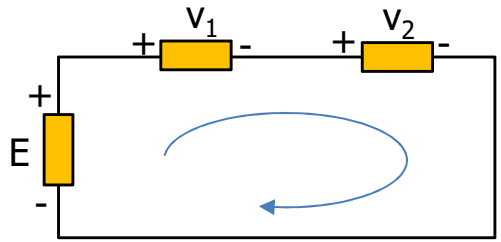
- 전기회로의 접속점에서 그 점으로 흘러드는 전류의 합은 그 점으로부터 흘러나가는 전류의 합과 같다.



$i_4 = i_1 + i_2 + i_3$

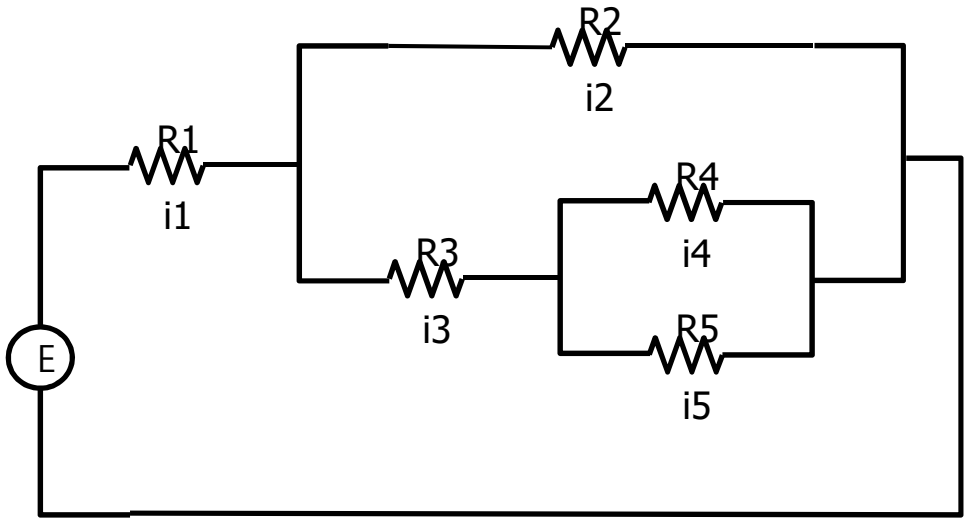
◆ 키르히호프 전압의 법칙 (KVL) –

- 전기회로에서 폐루프 (closed loop)이 구성될 때, 그 폐루프를 따라 전압 강하 (voltage drop)를 차례로 합한 최종값은 0이 된다.



$E = v_1 + v_2$

직병렬 전자회로의 연립방정식



E = 100 V
R1 = 10 Ω
R2 = 10 Ω
R3 = 5 Ω
R4 = 10 Ω
R5 = 10 Ω

(a) 5개의 저항으로 구성된 직병렬 전자회로

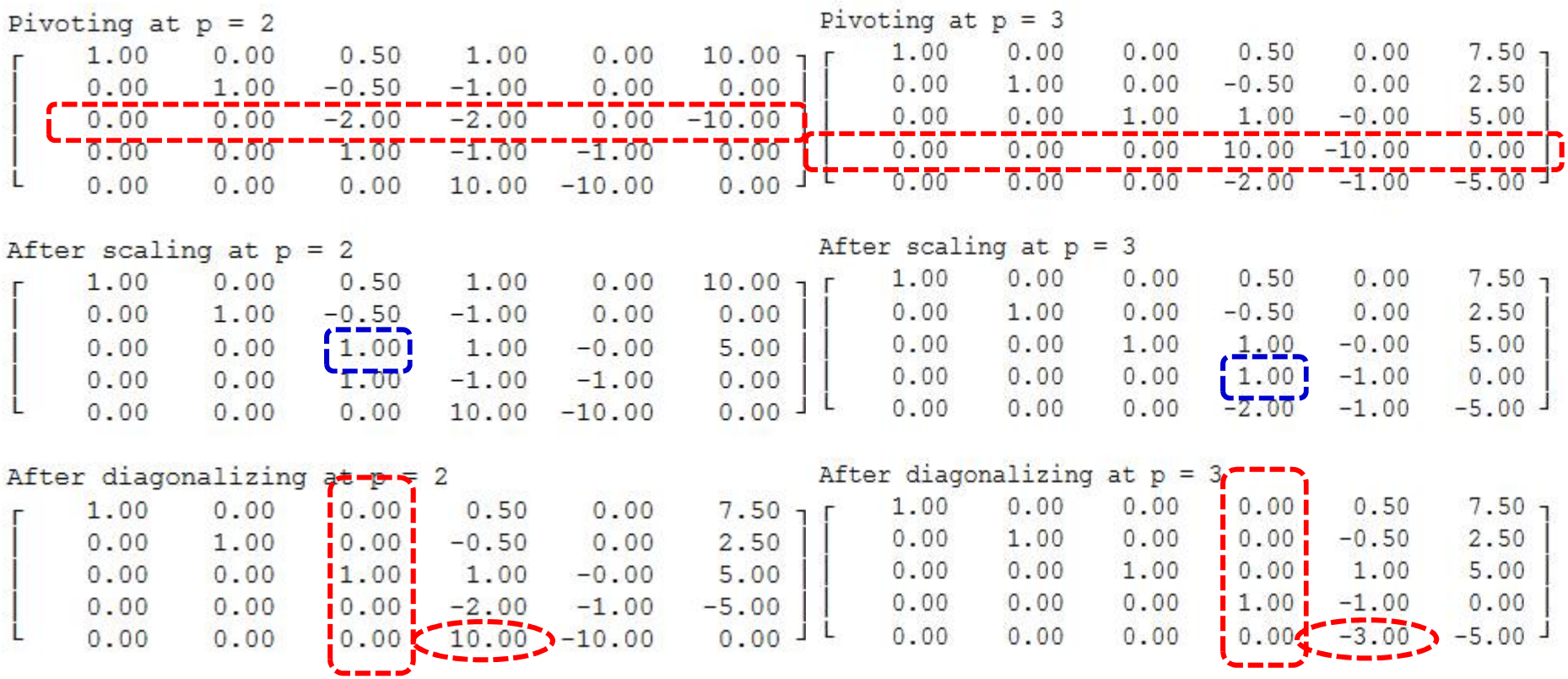
$$\begin{bmatrix} R_1 i_1 + R_2 i_2 = E \\ i_1 = i_2 + i_3 \\ i_3 = i_4 + i_5 \\ R_2 i_2 = R_3 i_3 + R_4 i_4 \\ R_4 i_4 = R_5 i_5 \end{bmatrix} \xrightarrow{Augmented\ Matrix} \begin{bmatrix} R_1 & R_2 & 0 & 0 & 0 & E \\ 1 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 & 0 \\ 0 & R_2 & -R_3 & -R_4 & 0 & 0 \\ 0 & 0 & 0 & R_4 & -R_5 & 0 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 10 & 10 & 0 & 0 & 0 & 100 \\ 1 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 & 0 \\ 0 & 10 & -5 & -10 & 0 & 0 \\ 0 & 0 & 0 & 10 & -10 & 0 \end{bmatrix}$$

Gauss-Jordan 소거법을 사용한 직병렬 회로 해석(2)

```
Augmented Matrix size_N : 5
Augmented Matrix :
┌   10.00   10.00    0.00    0.00    0.00  100.00 ┐
│    1.00   -1.00   -1.00    0.00    0.00    0.00 │
│    0.00    0.00    1.00   -1.00   -1.00    0.00 │
│    0.00   10.00   -5.00  -10.00    0.00    0.00 │
└    0.00    0.00    0.00   10.00  -10.00    0.00 ┘

Pivoting at p = 0
┌   10.00   10.00    0.00    0.00    0.00  100.00 ┐
│    1.00   -1.00   -1.00    0.00    0.00    0.00 │
│    0.00    0.00    1.00   -1.00   -1.00    0.00 │
│    0.00   10.00   -5.00  -10.00    0.00    0.00 │
└    0.00    0.00    0.00   10.00  -10.00    0.00 ┘

After scaling at p = 0
┌    1.00    1.00    0.00    0.00    0.00   10.00 ┐
│    1.00   -1.00   -1.00    0.00    0.00    0.00 │
│    0.00    0.00    1.00   -1.00   -1.00    0.00 │
│    0.00   10.00   -5.00  -10.00    0.00    0.00 │
└    0.00    0.00    0.00   10.00  -10.00    0.00 ┘

After diagonalizing at p = 0
┌    1.00    1.00    0.00    0.00    0.00   10.00 ┐
│    0.00   -2.00   -1.00    0.00    0.00  -10.00 │
│    0.00    0.00    1.00   -1.00   -1.00    0.00 │
│    0.00   10.00   -5.00  -10.00    0.00    0.00 │
└    0.00    0.00    0.00   10.00  -10.00    0.00 ┘

Pivoting at p = 1
┌    1.00    1.00    0.00    0.00    0.00   10.00 ┐
│    0.00   10.00   -5.00  -10.00    0.00    0.00 │
│    0.00    0.00    1.00   -1.00   -1.00    0.00 │
│    0.00   -2.00   -1.00    0.00    0.00  -10.00 │
└    0.00    0.00    0.00   10.00  -10.00    0.00 ┘

After scaling at p = 1
┌    1.00    1.00    0.00    0.00    0.00   10.00 ┐
│    0.00    1.00   -0.50   -1.00    0.00    0.00 │
│    0.00    0.00    1.00   -1.00   -1.00    0.00 │
│    0.00   -2.00   -1.00    0.00    0.00  -10.00 │
└    0.00    0.00    0.00   10.00  -10.00    0.00 ┘

After diagonalizing at p = 1
┌    1.00    0.00    0.50    1.00    0.00   10.00 ┐
│    0.00    1.00   -0.50   -1.00    0.00    0.00 │
│    0.00    0.00    1.00   -1.00   -1.00    0.00 │
│    0.00    0.00   -2.00   -2.00    0.00  -10.00 │
└    0.00    0.00    0.00   10.00  -10.00    0.00 ┘
```

Gauss-Jordan 소거법을 사용한 직병렬 회로 해석(3)



Gauss-Jordan 소거법을 사용한 직병렬 회로 해석(4)

```
Pivoting at p = 4
┌    1.00    0.00    0.00    0.00    0.50    7.50 ┐
│    0.00    1.00    0.00    0.00   -0.50    2.50 │
│    0.00    0.00    1.00    0.00    1.00    5.00 │
│    0.00    0.00    0.00    1.00   -1.00    0.00 │
└    0.00    0.00    0.00    0.00   -3.00   -5.00 ┘

After scaling at p = 4
┌    1.00    0.00    0.00    0.00    0.50    7.50 ┐
│    0.00    1.00    0.00    0.00   -0.50    2.50 │
│    0.00    0.00    1.00    0.00    1.00    5.00 │
│    0.00    0.00    0.00    1.00   -1.00    0.00 │
└    0.00    0.00    0.00    0.00    1.00    1.67 ┘

After diagonalizing at p = 4
┌    1.00    0.00    0.00    0.00    0.00    6.67 ┐
│    0.00    1.00    0.00    0.00    0.00    3.33 │
│    0.00    0.00    1.00    0.00    0.00    3.33 │
│    0.00    0.00    0.00    1.00    0.00    1.67 │
└    0.00    0.00    0.00    0.00    1.00    1.67 ┘
```

```
The solution of the given linear system:
 x[0] : 6.666667
 x[1] : 3.333333
 x[2] : 3.333333
 x[3] : 1.666667
 x[4] : 1.666667
```

선형시스템 해 산출을 위한
첨가행렬 기반 Gauss-Jordan 소거법 구현

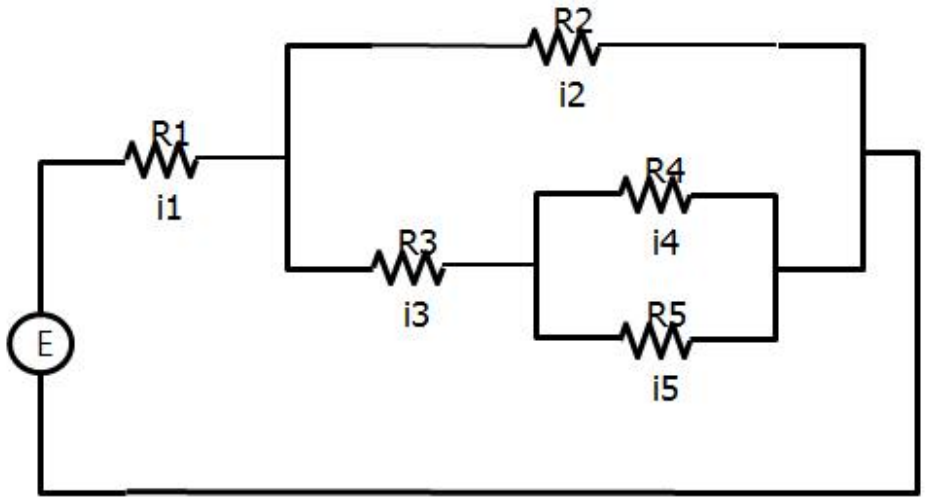
```
void pivoting(double **augMtrx, int size_N, int p, int*piv_found)
{
        double xmax = 0.0, xtemp;
        int  j, k, max_row;

        // find maximum pivot
        xmax = fabs(augMtrx[p][p]);
        max_row = p;
        for (j = p + 1; j< size_N; j++) {
                if (fabs(augMtrx[j][p]) > xmax) {
                        xmax = fabs(augMtrx[j][p]);
                        max_row = j;
                }  // end if
        }  // end for
        // swap rows if non-zero pivot was found
        if (fabs(xmax) < EPSILON) {  // EPSILON (ε) = 0.000001
                *piv_found = 0; // False
        }  else {
                *piv_found = 1; // True
                if (max_row != p) { // swap rows
                        for (k = p; k <= size_N; k++) {
                                xtemp = augMtrx[p][k];
                                augMtrx[p][k] = augMtrx[max_row][k];
                                augMtrx[max_row][k] = xtemp;
                        } // end for
                } // end if
        } // end if-else
} // end pivoting
```

```
void diagonalize_FileOut(FILE *fout, double **mtrxAug, int size_N, int *solExist)
{
      int  j, k, p = 0;  // pivot index
      double  pivWeight, w;
      // Pivoting and Scaling
      for (p = 0; (*solExist) && (p<size_N); p++) {
            fprintf(fout, "\nPivoting at p = %d\n", p);
            pivoting(mtrxAug, size_N, p, solExist);
            fprintMtrx(fout, mtrxAug, size_N, size_N + 1);
            if (*solExist) {
                  // scaling the elements in pivot row
                  if (mtrxAug[p][p] != 1.0) { // adjust pivot row, if the pivot is not 1.0
                        pivWeight = mtrxAug[p][p]; // scaling value pivot row
                        mtrxAug[p][p] = 1.0;   // make the diagonal element at pivot row into 1.0
                        for (k = p + 1; k <= size_N; k++) {
                              mtrxAug[p][k] /= pivWeight;
                        } // for
                  } // if(mtrxAug[p][p] != 1.0)
            } else  {  // end if (*solExist)
                  break;
            }

            fprintf(fout, "\nAfter scaling at p = %d\n", p);
            fprintMtrx(fout, mtrxAug, size_N, size_N + 1);
            // eliminate coefficients above and below the pivot row
```

```
            // Diagonalize
            for (j = 0; j<size_N; j++)
            {  // for j-th row
                  if ((j != p) && (mtrxAug[j][p] != 0.0)) {
                        w = mtrxAug[j][p];  // weight
                        mtrxAug[j][p] = 0.0;
                           // eliminate coefficients, just except the pivot row
                        for (k = p + 1; k <= size_N; k++) {
                              mtrxAug[j][k] =
                               mtrxAug[j][k] - w * mtrxAug[p][k];   // Rj = Rj – w * Rp
                        }  // end for
                  } // end if ( j !=p)
            } // end for (j=0; …)
            fprintf(fout, "\nAfter diagonalizing at p = %d\n", p);
            fprintMtrx(fout, mtrxAug, size_N, size_N + 1);
      } // end for (p=0; …)
}
```

Gauss-Jordan 소거법을 사용한 선형방정식 해법

```
/* Gauss-Jordan.h*/

#ifndef GAUSS_JORDAN_H
#define GAUSS_JORDAN_H

#include <stdio.h>
#include <math.h>
#include "Matrix.h"
#define EPSILON 0.000001

double **getAugMtrx(FILE *fin, int *pSize_N);
void pivoting(double **augMtrx, int size_N, int p, int*piv_found);
void diagonalize(double **mtrxAug, int size_N, int *solExist);
void diagonalize_FileOut(FILE *fout, double **mtrxAug, int size_N, int *solExist);

#endif
```



E = 100 V
R1 = 10 Ω
R2 = 10 Ω
R3 = 5 Ω
R4 = 10 Ω
R5 = 10 Ω

LinearSystem_AugMtrx.txt

```
5  6
10.0   10.0   0.0    0.0   0.0   100.0
 1.0   -1.0  -1.0    0.0   0.0     0.0
 0.0    0.0   1.0   -1.0  -1.0     0.0
 0.0   10.0  -5.0  -10.0   0.0     0.0
 0.0    0.0   0.0   10.0 -10.0     0.0
```

```
double ** fGetAugMtrx(FILE* fin, int *pAug_size)
{
    double d = 0.0;
    int row_size, col_size;
    double** dAugM;

    if (fin == NULL)
    {
        printf("Error in getDoubleMatrixData() - file pointer is NULL !!₩n");
        exit(-1);
    }

    fscanf(fin, "%d %d", &row_size, &col_size);
    dAugM = (double**)calloc(row_size, sizeof(double*));
    for (int r = 0; r < row_size; r++)
    {
        dAugM[r] = (double*)calloc(col_size, sizeof(double));
    }
    if (dAugM == NULL)
    {
        printf("Error in Creation of AugMtrx() !!₩n");
        exit(-1);
    }
    if ((row_size + 1) != col_size)
    {
        printf("Error in Augmented Matrix (row_size=%d, col_size=%d) !!", row_size, col_size);
    }
    for (int m = 0; m < row_size; m++)
        for (int n = 0; n < col_size; n++)
        {
            if (fscanf(fin, "%lf", &d) != EOF)
                dAugM[m][n] = d;
        }
    *pAug_size = row_size;
    return dAugM;
}
```

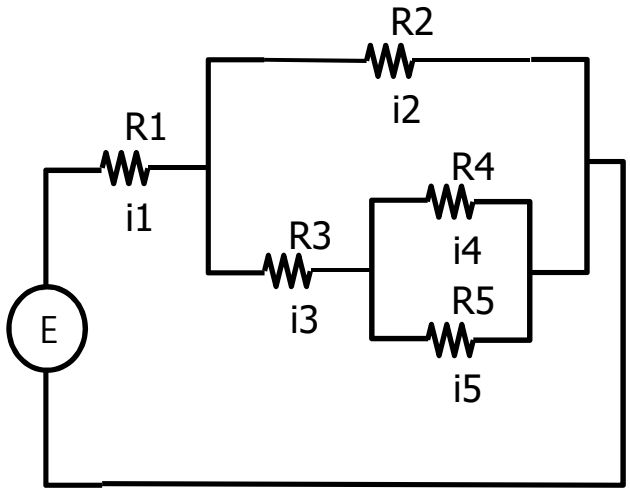
```
const char *augMatrix_inputFile = "LinearSystem_AugMtrx.txt";
void test_GaussJordanElimination_LinearSystem(FILE *fout)
{
	FILE *fin;
	int size_N, row_size, col_size;
	double *solution;
	double **augMtrx = NULL;
	int i, j, solExist = 1, error = 0;
	double d;

	fin = fopen(augMatrix_inputFile, "r");
	//fin = fopen("ElectronicCircuit_A_5x5.txt", "r");
	if (fin == NULL)
	{
		printf("Error in opening input.txt file (%s)!!\n", augMatrix_inputFile);
		exit(-1);
	}

	fscanf(fin, "%d", &size_N);
	solution = (double *)calloc(size_N, sizeof(double));
	fprintf(fout, "Augmented Matrix size_N : %d\n", size_N);
	augMtrx = getAugmentedMtrx(fin, size_N);
	fprintf(fout, "Augmented Matrix : \n");
	fprintMtrx(fout, augMtrx, size_N, size_N + 1);
	printf("Augmented Matrix : \n");
	printMtrx(augMtrx, size_N, size_N + 1);
```

```
    diagonalize_FileOut(fout, (double **)augMtrx, size_N, &solExist);
    if (solExist) {
        fprintf(fout, "The solution of the given linear system:\n");
        printf("The solution of the given linear system:\n");
        for (i = 0; i < size_N; i++) {
            solution[i] = augMtrx[i][size_N];
            fprintf(fout, " x[%d] : %4f\n", i, solution[i]);
            printf(" x[%d] : %4f\n", i, solution[i]);
        }
    } else {
        fprintf(fout, "No unique solution\n");
        printf("No unique solution\n");
    }

    for (int i = 0; i < size_N; i++)
        free(augMtrx[i]);
    free(augMtrx);
    free(solution);
    fclose(fin);
}
```

Gauss-Jordan 소거법을 사용한 직병렬 회로 해석 (1)

◆ 실행 결과

```
Testing Matrix Operations with 2-Dimensional Dynamic Array
 1: Check addresses of 2-Dim array for Matrix
 2: Calculate total average of 2-Dim array
 3: Test 2-D Dynamic Array Creation for Matrix with File I/O
 4: Test Matrix Addition, Subtraction
 5: Test Matrix Multiplicaton
 6: Test Gauss-Jordan Elimination for Linear System
-1: Quit
Input menu (-1 to quit) : 6

Augmented Matrix :
┌   10.00   10.00    0.00    0.00    0.00  100.00 ┐
|    1.00   -1.00   -1.00    0.00    0.00    0.00 |
|    0.00    0.00    1.00   -1.00   -1.00    0.00 |
|    0.00   10.00   -5.00  -10.00    0.00    0.00 |
└    0.00    0.00    0.00   10.00  -10.00    0.00 ┘
The solution of the given linear system:
 x[0] : 6.666667
 x[1] : 3.333333
 x[2] : 3.333333
 x[3] : 1.666667
 x[4] : 1.666667
```



E = 100 V
R1 = 10 Ω
R2 = 10 Ω
R3 = 5 Ω
R4 = 10 Ω
R5 = 10 Ω

행 연산 (row-operation)을 사용한 역행렬 계산 및 선형시스템 해법

선형 시스템 (Linear System)과 연립 방정식

◆ 선형시스템과 연립방정식의 예

- ***A* × *X* = *B***

$$\begin{aligned} x_1 &+ x_2 &+ x_3 &= 4 \\ 2x_1 &+ 3x_2 &+ x_3 &= 9 \\ x_1 &- x_2 &- x_3 &= -2 \end{aligned}$$

Solutions to be calculated

- $A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 3 & 1 \\ 1 & -1 & -1 \end{bmatrix}$, $X = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$, $B = \begin{bmatrix} 4 \\ 9 \\ -2 \end{bmatrix}$

◆ 선형 시스템의 해 (solution) 산출

- 첨가행렬 (augmented matrix)과 Gauss-Jordan 소거법
- **행연산을 사용하여 계수 행렬 *A*의 역행렬 (A^{-1}) 계산**
  => $X = A^{-1} \times B$

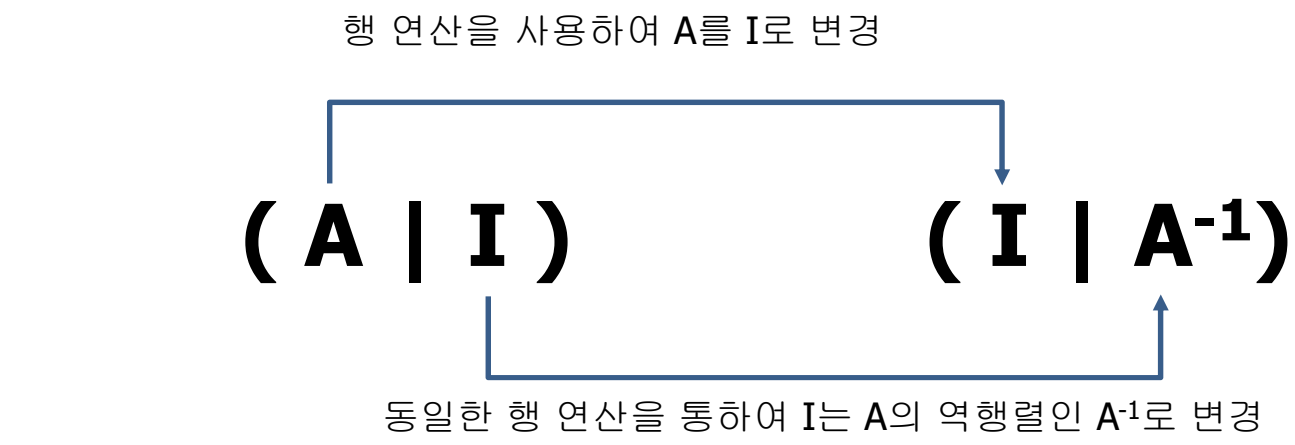
행연산을 사용한 역행렬 계산

◆ 행 연산을 사용한 역행렬 계산

- n × n 크기의 정방행렬 ***A*** 와 n × n 크기의 단위행렬 ***I*** 를 구성한 후, 가우스 소거법의 삼각행렬 산출 및 후방 대입 (backward substitution)을 사용하여 정방행렬 ***A***를 단위 행렬 ***I*** 로 변경하면, 단위 행렬 ***I*** 는 정방행렬 ***A*** 의 역행렬인 A^{-1} 로 변경됨

- $(A \mid I) = \left(\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right)$

◆ 행연산을 사용한 역행렬의 계산



Original Matrix (A) | Identity Matrix

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Identity Matrix | Inverse Matrix (B = A^{-1})

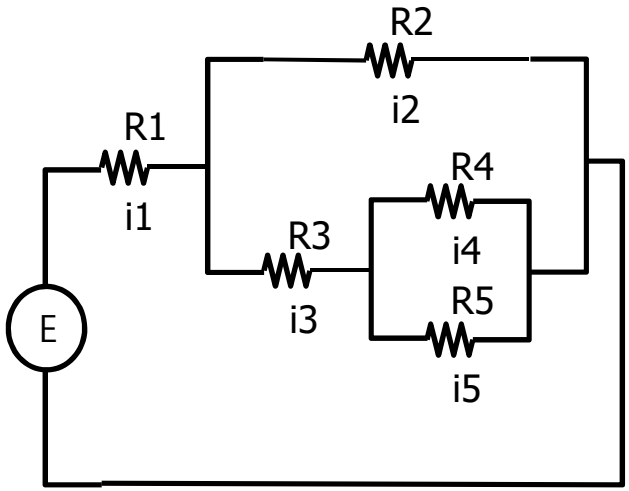
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix}$$

```
double ** invMtrxGaussJordanElim_FileOut(FILE *fout, double **mA, int size_N)
{
    int j, k, p = 0;  // pivot index
    double pivWeight, w;
    double **mAUG;  // augment matrix in N x 2N size for { A | I }
    double **mInv;
    // Part 1. Prepare the n x 2n augmented matrix mAUG[N][2N]
    mAUG = (double **)calloc(size_N, sizeof(double *));
    for (int i = 0; i < size_N; i++)
        mAUG[i] = (double *)calloc(size_N * 2 , sizeof(double));
    for (j = 0; j < size_N; j++) {
       for (k = 0; k < size_N; k++)
          mAUG[j][k] = mA[j][k];
       for (k = size_N; k < size_N * 2; k++)
          mAUG[j][k] = (k == (j + size_N)) ? 1.0 : 0.0;
    } // end for(j=0; . . . )

    //Part 2. Scaling and Elimination of coefficients of above and below
    //           the diagonal elements of mAUG[][]
    for (p = 0; p < size_N; p++) {
        // scaling
       if (mAUG[p][p] != 1.0) { // adjust pivot row, if the pivot is not 1.0
            pivWeight = mAUG[p][p]; // scaling value pivot row
            mAUG[p][p] = 1.0;   // make the diagonal element at pivot row into 1.0
            for (k = p + 1; k < size_N * 2; k++) {
                mAUG[p][k] /= pivWeight; // scaling the elements in pivot row
            }
       } // end if (mAUG[p][p] != 1.0)
```

```
        fprintf(fout, "After scaling at p = %d\n", p);
        fprintMtrx(fout, mAUG, size_N, size_N*2);
        // eliminate coefficients above and below the pivot row
        for (j = 0; j < size_N; j++) { // for j-th row
            if ((j != p) && (mAUG[j][p] != 0.0))
            {
                w = mAUG[j][p];  // weight
                mAUG[j][p] = 0.0;   // eliminate coefficients, just except the pivot row
                for (k = p + 1; k < 2 * size_N; k++) {
                    mAUG[j][k] -= w * mAUG[p][k];   // Rj = Rj – w * Rp
                }
            } // end if ( j !=p)
        } // end for (j=0; … )
        fprintf(fout, "After diagonalizing at p = %d\n", p);
        fprintMtrx(fout, mAUG, size_N, size_N*2);
    } // end for (p=0; …)

    //Part 3. Create mInv[][]  and copy the result inverse matrix
    mInv = (double**)calloc(size_N, sizeof(double *));
    for (int r = 0; r < size_N; r++)
    {
        mInv[r] = (double*)calloc(size_N, sizeof(double));
    }
    for (j = 0; j < size_N; j++) {
        for (k = 0; k < size_N; k++) {
            mInv[j][k] = mAUG[j][k + size_N];
        } // end for (k=0; … )
    } // end for (j=0; . . . )
    return mInv;
}
```

역행렬 계산을 사용한 직병렬 회로 해석 (1)

◆ 직병렬 회로의 선형 방정식 및 데이터 파일



E = 100 V
R1 = 10 Ω
R2 = 10 Ω
R3 = 5 Ω
R4 = 10 Ω
R5 = 10 Ω

Input data file for Linear System

```
5
10.0  10.0  0.0   0.0   0.0
1.0  -1.0  -1.0   0.0   0.0
0.0   0.0   1.0  -1.0  -1.0
0.0  10.0  -5.0 -10.0   0.0
0.0   0.0   0.0  10.0 -10.0

100.0 0.0 0.0 0.0 0.0
```

역행렬 계산을 사용한 직병렬 회로 해석 (2)

◆ 실행 결과

```
Matrix A:
┌    10.0000    10.0000     0.0000     0.0000     0.0000 ┐
|     1.0000    -1.0000    -1.0000     0.0000     0.0000  |
|     0.0000     0.0000     1.0000    -1.0000    -1.0000  |
|     0.0000    10.0000    -5.0000   -10.0000     0.0000  |
└     0.0000     0.0000     0.0000    10.0000   -10.0000 ┘
 Inverse A:
┌     0.0667     0.3333     0.1667    -0.0333    -0.0167 ┐
|     0.0333    -0.3333    -0.1667     0.0333     0.0167  |
|     0.0333    -0.3333     0.3333    -0.0667    -0.0333  |
|     0.0167    -0.1667    -0.3333    -0.0333     0.0333  |
└     0.0167    -0.1667    -0.3333    -0.0333    -0.0667 ┘
mE:
┌   100.0000 ┐
|     0.0000  |
|     0.0000  |
|     0.0000  |
└     0.0000 ┘
Res = Inv_A x mE:
┌     6.6667 ┐
|     3.3333  |
|     3.3333  |
|     1.6667  |
└     1.6667 ┘
```

Homework 8

Homework 8

8.1 행렬 연산 프로그램

(1) 입력 데이터가 포함된 입력 파일 (Array_2D_input.txt)과 출력 결과를 저장할 출력 데이터 파일(output.txt)을 fopen()을 사용하여 각각 입력 및 출력할 수 있도록 준비 하라. 입력 데이터 파일에는 행렬의 크기를 나타내는 정수 (integer) 2개 (행, 열 크기)와 행렬을 위한 총 행크기 (size_row) x 열크기(size_column) 개수의 double 형 데이터가 행렬 A와 행렬 B를 위하여 차례로 포함되어 있도록 구성할 것.
입력 파일 (Array_2D_input.txt)의 예는 다음과 같다.

```
6  7
1.0 2.0 3.0 4.0 5.0 6.0 7.0
2.0 3.0 4.0 5.0 1.0 3.0 5.0
3.0 2.0 5.0 3.0 2.0 1.0 0.0
4.0 3.0 2.0 7.0 2.0 5.0 3.0
5.0 4.0 3.0 2.0 9.0 1.0 2.0
6.0 7.0 8.0 9.0 10.0 11.0 5.0

6  7
13.0 15.5 17.0 14.0 15.0 16.0 17.0
11.5 22.0 23.0 24.0 25.0 26.0 27.0
21.0 20.5 33.0 32.0 35.0 36.0 37.0
33.0 32.0 37.5 44.0 43.0 42.0 41.0
42.0 47.0 42.0 49.5 55.0 60.0 65.0
54.0 53.0 52.0 59.0 51.2 70.0 60.0
```

```
7   6
1.0 2.0 3.0 4.0 5.0 6.0
7.0 8.0 9.0 1.0 2.0 3.0
2.0 3.0 4.0 5.0 1.0 3.0
5.0 3.0 2.0 5.0 3.0 2.0
1.0 0.0 4.0 3.0 2.0 7.0
2.0 5.0 3.0 5.0 4.0 3.0
2.0 9.0 1.0 2.0 6.0 7.0
```

Homework 8

8.1 행렬 연산 프로그램 (계속)

(2) 지정된 입력 파일로 부터 행렬의 크기를 나타내는 정수 size_row과 size_column를 읽고, 지정된 size_row x size_column 행렬을 저장할 수 있는 2차원 배열을 동적으로 생성한 후, 지정된 입력파일로 부터 double형 데이터를 읽어 행렬의 초기값으로 설정하는 double **fGetMtrx(FILE *fin, int *size_row, int *size_column)을 작성하라. fGetMtrx() 함수의 실행 결과로 동적으로 생성된 행렬과 그 행렬의 크기 (size_row, size_column)이 return-by-pointer 방식으로 반환되어야 한다.
fGetMtrx() 함수에서는 지정된 입력 데이터 파일 (예: Array_2D_input.txt)로부터 fscanf() 함수를 사용하여, 각 size_row x size_column 개의 실수형 (double) 데이터를 읽어 2차원 배열을 초기화하는 기능이 포함된다.
main() 함수에서는 fGetMtrx() 함수를 3번 호출하여 3개의 행렬 (mA, mB, mC)를 동적으로 생성하고 파일로 부터 초기화 실수형 데이터를 입력 받아 설정하게 된다
.

Homework 8

8.1 행렬 연산 프로그램 (계속)

(3) 입력된 3개의 행렬의 값을 화면과 파일에 출력하기 위한 printMtrx(double **dM, int size_r, int size_c) 함수와 fprintMtrx(FILE* fout, double **dM, int size_r, int size_c)함수를 각각 작성할 것. 행렬의 각 값은 최소 5자리를 확보하고, 오른쪽 정렬로 출력되도록 할 것. 이 때, 행렬을 표시하기 위하여, 확장 완성형 코드를 사용할 것.

출력 결과	확장 완성형 코드	사용 방법
ㅡ	0xa6, 0xa1	printf(" %c%c", 0xa6, 0xa1)
ㅣ	0xa6, 0xa2	printf(" %c%c", 0xa6, 0xa2)
ㄱ	0xa6, 0xa3	printf(" %c%c", 0xa6, 0xa3)
ㄴ	0xa6, 0xa4	printf(" %c%c", 0xa6, 0xa4)
ㄷ	0xa6, 0xa5	printf(" %c%c", 0xa6, 0xa5)
ㄹ	0xa6, 0xa6	printf(" %c%c", 0xa6, 0xa6)

8.1 행렬 연산 프로그램 (계속)

(4) 이 행렬 연산 함수들은 연산 결과를 저장할 2차원 배열을 동적으로 생성하고, 연산 결과를 저장하며, 그 주소를 반환하여야 한다:

```
double **addMtrx(double **mA, double **mB, int size_row, int size_col);
double **subtractMtrx(double **mA, double **mB, int size_row, int size_col);
double **multiplyMtrx(double **mA, double **mC, int size_row_a, int size_col_a, int
    size_col_c);
```

(5) 행렬 연산을 총괄하는 main() 함수에서, 행렬 mA, mB, mC를 입력 데이터 파일로 부터 읽고 동적으로 생성하며, 행렬 mA와 mB의 덧셈, 뺄셈을 계산하여 각각 행렬 mD, mE에 저장하고, mA와 mC의 곱셈을 계산하여 mF에 저장하도록 하라. 이 기능을 수행하기 위하여 fGetMtrx(), fprintfMtrx(), addMtrx(), subtractMtrx(), multiplyMtrx() 함수 호출하며, 행렬 준비, 행렬 연산 및 결과 출력을 실행하고, 그 결과를 확인할 것.

(6) 역행렬을 계산하는 함수 double **inverseMtrx(double **mF, int size_N)를 작성하라. 역행렬 계산을 하기 위한 행렬 mF는 정방행렬 (square matrix)이며, 행의 크기와 열의 크기가 동일하다. inverseMtrx() 함수에서는 역행렬 결과를 저장할 2차원 배열을 동적으로 생성하며, 역행렬 결과를 저장하고, 그 주소를 반환하여 준다.

(7) main() 함수에서 행렬 mF의 역행렬을 inverseMtrx() 함수를 사용하여 계산하고, inv_mF에 저장하도록 하라. 다음으로 행렬 mF와 행렬 inv_mF를 multiplyMtrx() 함수를 사용하여 계산한 후, mI에 저장하고, fprintMtrx() 함수를 사용하여 파일로 출력하라. 출력된 결과로 부터 mI가 단위 행렬인 것을 확인하라.

8.1 행렬 연산 프로그램 (계속)

- Matrix.h 헤더파일 (예시)

```
/* 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
```

8.1 행렬 연산 프로그램 (계속)

- main() 함수 (예시)

```
/* main - Matrix Operations (1) */

#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";
    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");
    if (fin_ABC == NULL)
    {
        printf("Error in opening input data file (%s) !!", input_file_ABC);
        exit;
    }

    mA = fGetMtrx(fin_ABC, &a_row_size, &a_col_size);
    printMtrx((const char*)"mA", mA, a_row_size, a_col_size);
```

```
/* main - Matrix Operations (2) */

    mB = fGetMtrx(fin_ABC, &b_row_size, &b_col_size);
    printMtrx((const char*)"mB", mB, b_row_size, b_col_size);

    mC = fGetMtrx(fin_ABC, &c_row_size, &c_col_size);
    printMtrx((const char*)"mB", mB, b_row_size, b_col_size);

    d_row_size = a_row_size;
    d_col_size = b_col_size;
    mD = addMtrx(mA, mB, a_row_size, a_col_size);
    printMtrx((const char*)"mD = mA + mB", mD, d_row_size, d_col_size);

    e_row_size = a_row_size;
    e_col_size = b_col_size;
    mE = subtractMtrx(mA, mB, e_row_size, e_col_size);
    printMtrx((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);
    printMtrx((const char*)"mF = mA * mC", mF, f_row_size, f_col_size);

    inv_mF = invMtrxGaussJordanElim(mF, f_row_size);
    printMtrx((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);
    printMtrx((const char*)"mI = mF * inv_mF", mI, f_row_size, f_col_size);

    fclose(fin_ABC);
}
```

8.1 행렬 연산 프로그램 (계속)

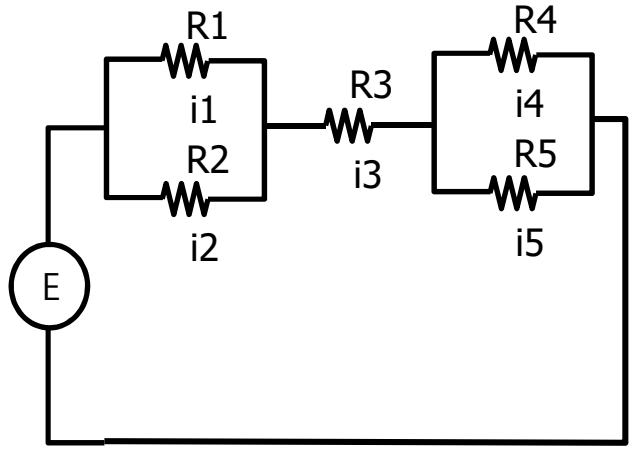
- 실행 결과

```
mA =
    1.00    2.00    3.00    4.00    5.00    6.00    7.00
    2.00    3.00    4.00    5.00    1.00    3.00    5.00
    3.00    2.00    5.00    3.00    2.00    1.00    0.00
    4.00    3.00    2.00    7.00    2.00    5.00    3.00
    5.00    4.00    3.00    2.00    9.00    1.00    2.00
    6.00    7.00    8.00    9.00   10.00   11.00    5.00
mB =
   13.00   15.50   17.00   14.00   15.00   16.00   17.00
   11.50   22.00   23.00   24.00   25.00   26.00   27.00
   21.00   20.50   33.00   32.00   35.00   36.00   37.00
   33.00   32.00   37.50   44.00   43.00   42.00   41.00
   42.00   47.00   42.00   49.50   55.00   60.00   65.00
   54.00   53.00   52.00   59.00   51.20   70.00   60.00
mB =
   13.00   15.50   17.00   14.00   15.00   16.00   17.00
   11.50   22.00   23.00   24.00   25.00   26.00   27.00
   21.00   20.50   33.00   32.00   35.00   36.00   37.00
   33.00   32.00   37.50   44.00   43.00   42.00   41.00
   42.00   47.00   42.00   49.50   55.00   60.00   65.00
   54.00   53.00   52.00   59.00   51.20   70.00   60.00
mD = mA + mB =
   14.00   17.50   20.00   18.00   20.00   22.00   24.00
   13.50   25.00   27.00   29.00   26.00   29.00   32.00
   24.00   22.50   38.00   35.00   37.00   37.00   37.00
   37.00   35.00   39.50   51.00   45.00   47.00   44.00
   47.00   51.00   45.00   51.50   64.00   61.00   67.00
   60.00   60.00   60.00   68.00   61.20   81.00   65.00
mE = mA - mB =
  -12.00  -13.50  -14.00  -10.00  -10.00  -10.00  -10.00
   -9.50  -19.00  -19.00  -19.00  -24.00  -23.00  -22.00
  -18.00  -18.50  -28.00  -29.00  -33.00  -35.00  -37.00
  -29.00  -29.00  -35.50  -37.00  -41.00  -37.00  -38.00
  -37.00  -43.00  -39.00  -47.50  -46.00  -59.00  -63.00
  -48.00  -46.00  -44.00  -50.00  -41.20  -59.00  -55.00
```

```
mF = mA * mC =
   72.00  132.00   86.00  100.00  100.00  131.00
   73.00  115.00   77.00   84.00   79.00   94.00
   46.00   51.00   64.00   65.00   41.00   62.00
   82.00  111.00   87.00  101.00   91.00  103.00
   64.00   80.00  108.00   85.00   76.00  135.00
  158.00  219.00  209.00  211.00  173.00  237.00
inv_mF =
   -0.00    0.09    0.15    0.07    0.06   -0.14
    0.00    0.03   -0.01   -0.05   -0.02    0.02
   -0.06   -0.03   -0.16   -0.05   -0.03    0.12
    0.04   -0.04    0.09    0.00   -0.02   -0.02
   -0.04   -0.09   -0.20    0.05   -0.02    0.11
    0.04    0.04    0.12    0.01    0.04   -0.10
mI = mF * inv_mF =
    1.00   -0.00    0.00   -0.00    0.00   -0.00
    0.00    1.00    0.00    0.00    0.00   -0.00
    0.00    0.00    1.00   -0.00    0.00   -0.00
    0.00    0.00    0.00    1.00    0.00   -0.00
    0.00   -0.00    0.00    0.00    1.00    0.00
    0.00    0.00    0.00    0.00    0.00    1.00
```

8.2 선형시스템의 해 (solution)를 Gauss-Jordan 소거법으로 구하는 프로그램 작성.

(1) 5개의 저항으로 구성된 직•병렬 전자회로에서 각 저항을 통하여 흐르는 전류의 양을 계산하는 선형 연립방정식을 정리하라.
(2) 위에서 정리한 연립방정식의 계수 행렬을 입력데이터 파일 (inputData.txt)에 저장하라.
(3) 선형시스템의 연립방정식에 해당하는 값 (계수 행렬)은 입력 데이터 파일 (inputData.txt)에서 읽어 Gauss-Jordan 소거법으로 해를 구할 수 있도록 첨가행렬 (augmented matrix)을 구성하기 위한 fgetDoubleAugMtrx() 함수를 작성하고, 해를 구하기 위한 pivoting() 함수와 diagonalize() 함수를 작성하라.
(4) main() 함수에서는 입력 데이터 파일 open, fgetDoubleAugMtrx() 함수를 호출하여 첨가행렬을 준비하고, diagonalize() 함수와 pivoting() 함수를 호출하여 해를 구한 뒤, 그 결과를 출력 파일 (output.txt)에 저장하도록 하라.
(5) 위 (1)에서 주어진 선형연립방정식의 계수 행렬 A의 역행렬을 계산하고, AX = B로 부터 X = $A^{-1}B$를 계산하는 방법을 사용하여 해를 구하라. 그리고, 이렇게 산출된 해를 위 (3)에서 구한 해와 비교하라.



E = 100 V
R1 = 10 Ω
R2 = 20 Ω
R3 = 30 Ω
R4 = 40 Ω
R5 = 50 Ω

```
/** main.cpp (1) */
#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,     FILE *fout;
    int size_N, row_size, col_size;
    double *solution;
    double **augMtrx = NULL;
    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);
    }
```

```
/** main.cpp (2) */

    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));

    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] = augMtrx[i][size_N];
            fprintf(fout, " x[%d] : %4f\n", i, solution[i]);
            printf(" x[%d] : %4f\n", i, solution[i]);
        }
    }
    else {
        printf("No unique solution\n");
        fprintf(fout, "No unique solution\n");
    }

    for (int i = 0; i < size_N; i++)
        free(augMtrx[i]);
    free(augMtrx);
    free(solution);
    fclose(fin);
    fclose(fout);
}
```

```
Augmented Matrix =
┌  10.00    0.00   30.00   40.00    0.00  100.00 ┐
└                                                 ┘
x[0] : 1.132075
x[1] : 0.566038
x[2] : 1.698113
x[3] : 0.943396
x[4] : 0.754717
```