Московский Государсвенный Университет
имени М.В.Ломоносова
Факультет Вычислительной математики и Кибернетики

Компьютерный практикум по учебному курсу "Введение в численные методы" Задание №1

Отчет о выполненном задании

студента 203 учебной группы факультета ВМК МГУ
Журавского Максима Игоревича

Москва, 2017

Оглавление

Глава 1

Введение

Целью данной работы является реализация численный методов нахождения:

- решения заданных систем линейных алгебраических уравнений методами Гаусса, в т.ч. методом Гаусса с выбором главного элемента, и методом верхней релаксации.
- определителя заданных матриц
- матрицы, обратной данной

Кроме того, в работе исследуется вопросы устойчивости метода Гаусса при больших размерах матрицы коэффициентов и скорости сходимости к точному решению задачи итераций метода верхней релаксации при изменении итерационного параметра ω.

Глава 2

Алгоритмы решения

2.1 Метод Гаусса

Решения системы линейных алгебраических уравнений методом Гаусса производится в два этапа, именуемые "прямым"и "обратным"ходом, в результате которых из соответствующей исследуемой системе линейных алгебраических уравнений расширенной матрицы находится решение системы. В результате "прямого хода"матрица коэффициентов, входящая в расширенную матрицу, путем линейных пребразований строк и их перестановок последовательно приводится к верхнему треугольному виду. На этапе "обратного хода"выполняется восстановление решения путем прохода по строкам матрицы в обратном направлении.

В отличие от классического метода Гаусса в метода Гаусса с выбором главного элемента на каждой итерации выбирается максимальный из всех элементов матрицы, что позволяет уменьшить погрешность вычислений. Более подробно оба метода описаны в [1].

Задача вычисления определителя сводится к задаче приведения исследуемой матрицы к верхнему треугольному виду и последующим вычислением произведения её диагональных элементов. Для приведения матрицы к диагональному виду используется классический метод Гаусса.

Задача нахождения обратной матрицы решается с помощью метода Гаусса-Жордана. Над дополненной матрицей, составленной из столбцов исходной матрицы и единичной того же порядка, производятся преобразования метода Гаусса, в результате которых исходная матрица принимает вид единичной матрицы, а на месте единичной образуется матрица, обратная исходной. Теоретическое обоснование может быть найдено в [2].

2.2 Метод верхней релаксации

Метод вехний релакцации является стационарным итерационным методом, в котором каждый следующий вектор приближения вычисляется по формуле:

$$(D + \omega T_{\text{н}}) \frac{y_{k+1} - y_k}{\omega} + A y_k = f$$

где

ω - итерационный параметр,

y_k - k-й вектор приближения,

y_{k+1} - $(k+1)$-й вектор приближения,

A - матрица коэффициентов,

f - правая часть системы,

$T_{\text{н}}$ - нижняя диагональная матрица матрицы A,

D - матрица диагональных элементов матрицы A.

По теореме Самарского [3] для положительно определенных матриц A метод верхней релаксации сходится, если $0 < \omega < 2$. Однако, оптимальное значение ω в каждом случае находится экспериментально и зависит от матрицы A. В данной реализации метода допускается возможность изменения итерационного параметра с помощью флагов компиляции. Значением по умолчанию является $\omega = \frac{4}{3}$.

Глава 3

Программа

В качестве языка программорования системы был выбран язык С ввиду его гибкости и высоко производительности. Программа реализована модульно. Основная часть программы реализована в файле **main.c**. Исходный код модулей, отвечающих за реализацию метода Гаусса, метода Гаусса с выбором главного члена и метода верхней релаксации, расположен в файлах **gauss.c**, **modified-gauss.c** и **iteration.c** соответственно. Функции из приложения №2 находятся в файле **functions.c**. Ниже приведено содержание каждого из файлов.

main.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <math.h>

/* Constants */
typedef enum {
    SOR, GAUSS, MODIFIED
} Method;

#ifndef OMEGA
    #define OMEGA 1.3333333333333333
#endif
#define PRECISION 0.00000001


/* Prototypes */
extern double **matrix_create(unsigned n, unsigned m);
extern double **matrix_read(unsigned n);
extern double *matrix_read_vector(unsigned n);
extern int matrix_print(double **matrix, unsigned n);
extern int matrix_print_vector(double *vector, unsigned n);
extern void matrix_destroy(double **matrix, unsigned n);
extern double matrix_determinant(double **matrix, unsigned n);
extern double **matrix_inverse(double **matrix, unsigned n);

extern double *matrix_gauss_solve(double **matrix, const double *f,
    unsigned n);
extern double *matrix_modified_solve(double **matrix, const double *f,
    unsigned n);
```

```c
extern double *matrix_iteration_solve(double **a, const double *f, unsigned n, double omega, const double *start, double precision);

extern unsigned functions_init(unsigned task);
extern double **matrix_fill();
extern double *(*matrix_fill_vector)(double x);


/*
 * Entry Point.
 */
int main(int argc, char *argv[]) {
    // read matrix
    unsigned n;
    double **matrix, *f;

    // parse options
    if (argc == 2) {
        // use standard input stream
        scanf("%u", &n);
        if ((matrix = matrix_read(n)) == NULL || (f = matrix_read_vector(n)) == NULL) {
            fprintf(stderr, "> error: reading failed with message: %s\n", strerror(errno));
            exit(1);
        }
    } else if (argc == 4) {
        // use formula
        unsigned long formula = strtoul(argv[2], NULL, 10);
        if (errno != 0 || formula > 4) {
            fprintf(stderr, "> error: invalid option\n");
            exit(2);
        }
        double x = atof(argv[3]);
        n = functions_init((unsigned) formula);
        matrix = matrix_fill();
        f = matrix_fill_vector(x);
    } else {
        fprintf(stderr, "> error: wrong number arguments: %s gauss|mod|sor [formula x]\n",
            argv[0]);
        exit(2);
    }

    // choose method
    Method method;
    double start[n];
    if (strcmp(argv[1], "gauss") == 0) {
        // Gaussian elimination method
        method = GAUSS;
    } else if (strcmp(argv[1], "mod") == 0) {
        // Modified Gaussian elimination method
        method = MODIFIED;
    } else if (strcmp(argv[1], "sor") == 0) {
        // SOR method
        method = SOR;
        // create first approximation vector
        for (unsigned i = 0; i < n; i++) {
```

```
            start[i] = 0;
        }
    } else {
        fprintf(stderr, "> error: unknown method\n");
        exit(2);
    }

    // print processed matrix
    printf("> matrix:\n");
    matrix_print(matrix, n);
    putchar('\n');

    // calculate determinant
    double determinant = matrix_determinant(matrix, n);
    printf("> determinant:\n %.10g\n", determinant);
    putchar('\n');

    // process if posssible if possible
    if (determinant == 0) {
        printf("> inverse matrix:\n does not exist\n\n");
        printf("> solution:\n cannot be found using Gaussian elimination method\n\n");
    } else {
        // calculate inverse matrix
        double **inverse;
        if ((inverse = matrix_inverse(matrix, n)) == NULL) {
            fprintf(stderr, "> error: invertion failed\n");
            exit(1);
        }
        printf("> inverse matrix:\n");
        matrix_print(inverse, n);
        matrix_destroy(inverse, n);
        putchar('\n');

        // calculate solution using Gaussian elimination method
        double *solution;
        switch (method) {
            case GAUSS:
                solution = matrix_gauss_solve(matrix, f, n);
                break;
            case MODIFIED:
                solution = matrix_modified_solve(matrix, f, n);
                break;
            case SOR:
                solution = matrix_iteration_solve(matrix, f, n, OMEGA, start, PRECISION);
                break;
            default:
                fprintf(stderr, "> error: unknown method option\nterminationg...\n");
                exit(1);
        }

        if (solution == NULL) {
            fprintf(stderr, "> error: failed to calculate solution: %s\n", strerror(errno));
            exit(1);
        }
```

```c

        printf("> solution:\n");
        matrix_print_vector(solution, n);
        putchar('\n');

        // free memory
        free(solution);
    }

    // free memory
    free(f);
    matrix_destroy(matrix, n);

    // completion message
    printf("> program sucessfully finished!\n");
}
```

gauss.c:

```c
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include <math.h>
#include <stdbool.h>

int matrix_forward(double **matrix, unsigned n, unsigned m, bool to_swap);
void matrix_back(double **matrix, unsigned n, unsigned m);
void matrix_normalize(double **matrix, unsigned n, unsigned m);

/*
 * Function allocates memory for a matrix of size n * m on the heap and
   returns a handle to it.
 */
double **matrix_create(unsigned n, unsigned m) {
    double **matrix;
    if ((matrix = malloc(n * sizeof(matrix[0]))) == NULL) {
        // failed to allocate memory
        return NULL;
    }
    for (unsigned i = 0; i < n; i++) {
        if ((matrix[i] = malloc(m * sizeof(matrix[i][0]))) == NULL) {
            // failed to allocate memory
            for (unsigned j = 0; j < i; j++) {
                free(matrix[j]);
            }
            free(matrix);
            return NULL;
        }
    }
    return matrix;
}

/*
 * Function releases memory occupied by a matrix of size n * n.
 */
void matrix_destroy(double **matrix, unsigned n) {
    for (unsigned i = 0; i < n; i++) {
```

```c
        free(matrix[i]);
    }
    free(matrix);
}

/*
 * Function reads a matrix of size n * n and returns a handle to it.
 */
double **matrix_read(unsigned n) {
    double **matrix;
    if ((matrix = matrix_create(n, n)) == NULL) {
        return NULL;
    }
    for (unsigned i = 0; i < n; i++) {
        for (unsigned j = 0; j < n; j++) {
            if (scanf("%lf", &matrix[i][j]) == EOF) {
                // failed to read
                matrix_destroy(matrix, n);
                errno = EINVAL;
                return NULL;
            }
        }
    }
    return matrix;
}

/*
 * Function reads a matrix of size n * 1 and returns a handle to it.
 */
double *matrix_read_vector(unsigned n) {
    double *vector;
    if ((vector = malloc(n * sizeof(vector[0]))) == NULL) {
        // failed to allocate memory
        return NULL;
    }
    for (unsigned i = 0; i < n; i++) {
        if (scanf("%lf", &vector[i]) == EOF) {
            // falied to read
            free(vector);
            errno = EINVAL;
            return NULL;
        }
    }
    return vector;
}

/*
 * Function prints a matrix of size n * n.
 */
int matrix_print(double **matrix, unsigned n) {
    for (unsigned i = 0; i < n; i++) {
        for (unsigned j  = 0; j < n; j++) {
            // substitute negative zero with zero
            if (matrix[i][j] == 0) {
                matrix[i][j] = +0.0;
            }
            // print matrix row
            if (printf("%15.10g ", matrix[i][j]) == 0) {
```

```c
                // failed to print
                return -1;
            }
        }
        // put 'end of line' at the end
        if (putchar('\n') == EOF) {
            // failed to print
            return -1;
        }
    }
    return 0;
}

/*
 * Function prints a matrix of size n * 1.
 */
int matrix_print_vector(double *vector, unsigned n) {
    for (unsigned i = 0; i < n; i++) {
        // substitute negative zero with zero
        if (vector[i] == 0) {
            vector[i] = +0.0;
        }
        // print vector
        if (printf("%15.10g ", vector[i]) == 0) {
            // failed to print
            return -1;
        }
    }
    // put 'end of line' at the end
    if (putchar('\n') == EOF) {
        return -1;
    }
    return 0;
}

/*
 * Function solves matrix equation Ax = f using Gaussian Elimination method
    .
 */
double *matrix_gauss_solve(double **matrix, const double *f, unsigned n) {
   // create augmented matrix
   double **aug = matrix_create(n, n + 1);
   if (aug == NULL) {
       return NULL;
   }
   for (unsigned i = 0; i < n; i++) {
       for (unsigned j = 0; j < n; j++) {
           aug[i][j] = matrix[i][j];
       }
       aug[i][n] = f[i];
   }

   // perform forward elimination, normalization and back substitution
   matrix_forward(aug, n, n + 1, true);
   matrix_normalize(aug, n, n + 1);
   matrix_back(aug, n, n + 1);

   // calculate the result
```

```c
    double *result;
    if ((result = malloc(n * sizeof(result[0]))) == NULL) {
        matrix_destroy(aug, n);
        return NULL;
    }
    for (unsigned i = 0; i < n; i++) {
        result[i] = aug[i][n];
    }

    // free memory
    matrix_destroy(aug, n);
    return result;
}

/*
 * Function returns the number of the row with the greatest primary element.
 */
unsigned matrix_find_greatest(double **matrix, unsigned current, unsigned n) {
    unsigned result = current;
    for (unsigned j = current + 1; j < n; j++) {
        if (matrix[j][current] != 0 && (!matrix[result][current] || fabs(matrix[result][current]) < fabs(matrix[j][current])) ) {
            result = j;
        }
    }
    return result;
}

/*
 * Function swap specified rows in a matrix.
 */
void matrix_swap_rows(double **matrix, unsigned i, unsigned j) {
    double *temp = matrix[i];
    matrix[i] = matrix[j];
    matrix[j] = temp;
}

/*
 * Function subtracts current line from the following ones.
 * retruns -1 if division by 0 was about to occur.
 */
int matrix_subtract(double **matrix, unsigned current, unsigned n, unsigned m) {
    for (unsigned i = current + 1; i < n; i++) {
        if (matrix[current][current] == 0) {
            return -1;
        }
        double multiplier = matrix[i][current] / matrix[current][current];
        if (multiplier != 0) {
            for (unsigned j = current; j < m; j++) {
                matrix[i][j] -= multiplier * matrix[current][j];
            }
        }
    }
    return 0;
}
```

```c

/*
 * Function performs Forward Elimination of the augmented matrix n * m (n
   >= m).
 * Flag signal whether rows should be swapped or not.
 */
int matrix_forward(double **matrix, unsigned n, unsigned m, bool to_swap) {
    for (unsigned i = 0; i < n; i++) {
        // subtract greatest from the rest
        if (to_swap) {
            unsigned greatest = matrix_find_greatest(matrix, i, n);
            if (i != greatest) {
                matrix_swap_rows(matrix, i, greatest);
            }
        }
        if (matrix_subtract(matrix, i, n, m) == -1) {
            return -1;
        }
    }

    return 0;
}

/*
 * Function normalizes upper triangular n * m matrix.
 */
void matrix_normalize(double **matrix, unsigned n, unsigned m) {
    for (unsigned i = 0; i < n; i++) {
        for (unsigned j = i + 1; j < m; j++) {
            matrix[i][j] /= matrix[i][i];
        }
        matrix[i][i] = 1;
    }
}

/*
 * Function performs back substitution.
 */
void matrix_back(double **matrix, unsigned n, unsigned m) {
    for (unsigned i = n - 1; i > 0; i--) {
        for (unsigned prev = 0; prev < i; prev++) {
            // subtract current row from previous
            double multiplier = matrix[prev][i];
            if (multiplier == 0) {
                continue;
            }
            for (unsigned j = i; j < m; j++) {
                matrix[prev][j] -= matrix[i][j] * multiplier;
            }
        }
    }
}

/*
 * Function returns determinant of an n * n matrix.
 */
double matrix_determinant(double **matrix, unsigned n) {

```

```
    // calculate upper triangular matrix
    double **aug = matrix_create(n, n);
    if (aug == NULL) {
        // failed to allocate matrix
        return 0;
    }
    for (unsigned i = 0; i < n; i++) {
        for (unsigned j = 0; j < n; j++) {
            aug[i][j] = matrix[i][j];
        }
    }
    if (matrix_forward(aug, n, n, false) == -1) {
        matrix_destroy(aug, n);
        return 0;
    }

    // calculate determinant
    double result = 1;
    for (unsigned i = 0; i < n; i++) {
        result *= aug[i][i];
    }

    // free memory
    matrix_destroy(aug, n);
    return result;
}

/*
 *  Function returns a handle to the inverse of a given n * n matrix.
 */
double **matrix_inverse(double **matrix, unsigned n) {
    // create augmented matrix
    double **aug = matrix_create(n, 2 * n);
    if (aug == NULL) {
        return NULL;
    }
    for (unsigned i = 0; i < n; i++) {
        for (unsigned j = 0; j < n; j++) {
            aug[i][j] = matrix[i][j];
        }
        for (unsigned j = n; j < n * 2; j++) {
            aug[i][j] = (i == (j - n) ? 1 : 0);
        }
    }

    // perform forward elimination and back substitution
    matrix_forward(aug, n, 2 * n, false);
    matrix_normalize(aug, n, 2 * n);
    matrix_back(aug, n, 2 * n);

    // copy the result
    double **result = matrix_create(n, n);
    if (result == NULL) {
        return NULL;
    }
    for (unsigned i = 0; i < n; i++) {
        for (unsigned j = 0; j < n; j++) {
            result[i][j] = aug[i][j + n];
```

```c
        }
    }

    // free memory
    matrix_destroy(aug, n);

    return result;
}
```

modified-gauss.c:

```c
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdbool.h>

extern double **matrix_create(unsigned n, unsigned m);
extern void matrix_destroy(double **matrix, unsigned n);
extern void matrix_swap_rows(double **matrix, unsigned i, unsigned j);

void matrix_modified_forward(double **matrix, unsigned *transform, unsigned n, unsigned m);
void matrix_modified_back(double **matrix, const unsigned *transform, unsigned n, unsigned m);
void matrix_modified_normalize(double **matrix, const unsigned *transform, unsigned n, unsigned m);

typedef struct {
  unsigned i;
  unsigned j;
} Point;

/*
 * Function solves matrix equation Ax = f using modified Gaussian Elimination method.
 */
double *matrix_modified_solve(double **matrix, const double *f, unsigned n) {
    // create augmented matrix and a transformation vector
    double **aug = matrix_create(n, n + 1);
    if (aug == NULL) {
        return NULL;
    }
    for (unsigned i = 0; i < n; i++) {
        for (unsigned j = 0; j < n; j++) {
            aug[i][j] = matrix[i][j];
        }
        aug[i][n] = f[i];
    }
    unsigned transform[n];
    for (unsigned i = 0; i < n; i++) {
      transform[i] = i;
    }

    // perform forward elimination and normalization
    matrix_modified_forward(aug, transform, n, n + 1);
```

```c

    // calculate result
    double *result;
    if ((result = malloc(n * sizeof(result[0]))) == NULL) {
        matrix_destroy(aug, n);
        return NULL;
    }
    for (unsigned i = 0; i < n; i++) {
        // find effective x
        unsigned col = 0;
        for (unsigned j = 0; j < n; j++) {
            if (transform[j] == i) {
                col = j;
                break;
            }
        }
        // write out the result
        result[col] = aug[i][n];
    }

    // free memory
    matrix_destroy(aug, n);
    return result;
}

/* Function returns the number of the row and column of the greatest element */
Point matrix_modified_find_greatest(double **matrix, const unsigned *transform,
     unsigned current, unsigned n) {
     Point result = {current, 0};
     bool isFound = false;
   for (unsigned i = current; i < n; i++) {
       for (unsigned j = 0; j < n; j++) {
       if (transform[j] < current) {
         continue;
       }
           if (matrix[i][j] != 0 &&
           (!isFound || fabs(matrix[result.i][result.j]) < fabs(matrix[i][j])) ) {
                 result.i = i;
         result.j = j;
                  isFound = true;
           }
       }
   }
     return result;
}

/* Function performs Forward Elimination of the augmented matrix n * m */
void matrix_modified_forward(double **matrix, unsigned *transform, unsigned n, unsigned m) {
     for (unsigned i = 0; i < n; i++) {
         // find greatest from others and put it to front
         Point greatest = matrix_modified_find_greatest(matrix, transform, i, n);
         if (i != greatest.i) {
             matrix_swap_rows(matrix, i, greatest.i);
```

```
        }
        if (i != transform[greatest.j]) {
            unsigned col = 0;
            for (unsigned j = 0; j < n; j++) {
                if (transform[j] == i) {
                    col = j;
                    break;
                }
            }
        unsigned temp = transform[greatest.j];
        transform[greatest.j] = transform[col];
            transform[col] = temp;
        }

    // divide current row
    for (unsigned j = 0; j < m; j++) {
            if (j == greatest.j) {
                continue;
            }
      matrix[i][j] /= matrix[i][greatest.j];
    }
        matrix[i][greatest.j] = 1;

        // reduce other rows
        const unsigned col = greatest.j;
        for (unsigned row = 0; row < n; row++) {
            if (row == i) {
                continue;
            }
        double multiplier = matrix[row][col];
            if (multiplier != 0) {
                for (unsigned j = 0; j < m; j++) {
                    matrix[row][j] -= multiplier * matrix[i][j];
                }
            }
        }
    }
}
```

iteration.c:

```
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <stdio.h>

extern double **matrix_create(unsigned n, unsigned m);
extern void matrix_destroy(double **matrix, unsigned n);

/*
 * Function normalizes lower triangular n * m matrix.
 */
void normalize_reversed(double **matrix, unsigned n, unsigned m) {
    for (unsigned i = 0; i < n; i++) {
        for (unsigned j = 0; j < i; j++) {
            matrix[i][j] /= matrix[i][i];
        }
```

```
        matrix[i][m - 1] /= matrix[i][i];
        matrix[i][i] = 1;
        }
}

/*
 * Function performs back substitution.
 */
void back_reversed(double **matrix, unsigned n, unsigned m) {
        for (unsigned i = 0; i < n; i++) {
                for (unsigned next = i + 1; next < n; next++) {
                        // subtract current row from previous
                        double multiplier = matrix[next][i];
                        if (multiplier == 0) {
                                continue;
                        }
                        for (unsigned j = 0; j < m; j++) {
                                matrix[next][j] -= matrix[i][j] * multiplier;
                        }
                }
        }
}


/*
 * Function calculates next element of the iteration sequence.
 */
void matrix_iteration_next(const double **a, double **b, const double *f, unsigned n, double omega, double *current) {
    // create new right part in the augmented B matrix
    for (unsigned i = 0; i < n; i++) {
        b[i][n] = f[i];
        for (unsigned j = i; j < n; j++) {
            if (i == j) {
                b[i][n] -= (1 - 1 / omega) * a[i][j] * current[i];
            } else {
                b[i][n] -= a[i][j] * current[j];
            }
        }
    }

    // solve matrix equation with the new f vector: Bx = f
    normalize_reversed(b, n, n + 1);
    back_reversed(b, n, n + 1);

    // copy the result
    for (unsigned i = 0; i < n; i++) {
        current[i] = b[i][n];
    }
}

/*
 * Function calculates vector difference.
 */
double matrix_vector_diff(const double *a, const double *b, unsigned n) {
    double result = 0;
    for (unsigned i = 0; i < n; i++) {
        result += fabs(a[i] - b[i]);
```

```
    }

    return result;
}

/*
 * Functions calculates an estimate solution of the matrix equation
 * Ax = f using succesive over-relaxation method. Function returns a handle to the result.
 */
double *matrix_iteration_solve(const double **a, const double *f, unsigned n, double omega, const double *start, double precision) {
    // allocate vectors for current and previous elements of iteration sequence
    double *current, *previous;
    if ((current = malloc(sizeof(current[0]) * n)) == NULL) {
        return NULL;
    }
    if ((previous = malloc(sizeof(previous[0]) * n)) == NULL) {
        free(current);
        return NULL;
    }

    // create matrix B
    double **aug;
    if ((aug = matrix_create(n, n + 1)) == NULL) {
        free(current);
        free(previous);
        return NULL;
    }

    // start iteration process
    unsigned iterationCount = 0;
    memcpy(current, start, n * sizeof(start[0]));
    do {
        // construct B matrix
        for (unsigned i = 0; i < n; i++) {
            for (unsigned j = 0; j < n; j++) {
                if (i < j) {
                    aug[i][j] = 0;
                } else if (i == j) {
                    aug[i][j] = (1 / omega) * a[i][j];
                } else {
                    aug[i][j] = a[i][j];
                }
            }
        }

        // move result of the previous iteration and perform iteration
        memcpy(previous, current, n * sizeof(current[0]));
        matrix_iteration_next(a, aug, f, n, omega, current);
        iterationCount++;
    } while(matrix_vector_diff(current, previous, n) > precision);

    printf("> log: %d iterations made\n", iterationCount);

    // release memory
    matrix_destroy(aug, n);
```

```
  free(previous);

  return current;
}
```

functions.c:

```
#include <stdlib.h>
#include <math.h>
#include <assert.h>

/*
 * Functions for tasks from Appendix 2.
 * Tasks 1-4.
 */
extern double **matrix_create(unsigned n, unsigned m);

static unsigned M = 0;
static unsigned n = 0;
double *(*matrix_fill_vector)(double x) = NULL;

static double *f1(double x) {
  double *vector;
  if ((vector = malloc(n * sizeof(vector[0]))) == NULL) {
    return NULL;
  }
  for (unsigned i = 0; i < n; i++) {
    vector[i] = n * exp(x / i) * cos(x);
  }

  return vector;
}

static double *f2(double x) {
  double *vector;
  if ((vector = malloc(n * sizeof(vector[0]))) == NULL) {
    return NULL;
  }
  for (unsigned i = 0; i < n; i++) {
    vector[i] = fabs(x - (double) n / 10) * i * sin(x);
  }

  return vector;
}

static double *f3(double x) {
  double *vector;
  if ((vector = malloc(n * sizeof(vector[0]))) == NULL) {
    return NULL;
  }
  for (unsigned i = 0; i < n; i++) {
    vector[i] = x * exp(x / i) * cos(x / i);
  }

  return vector;
}

```

```
static double *f4(double x) {
  double *vector;
  if ((vector = malloc(n * sizeof(vector[0]))) == NULL) {
    return NULL;
  }
  for (unsigned i = 0; i < n; i++) {
    vector[i] = n * exp(x / i) * cos(x);
  }

  return vector;
}

/*
 * Function initializes the module for using formula-determined matrices.
 */
unsigned functions_init(unsigned task) {
  switch(task) {
    case 1:
      M = 1;
      n = 50;
      matrix_fill_vector = f1;
      break;
    case 2:
      M = 2;
      n = 40;
      matrix_fill_vector = f2;
      break;
    case 3:
      M = 3;
      n = 30;
      matrix_fill_vector = f3;
      break;
    case 4:
      M = 4;
      n = 100;
      matrix_fill_vector = f4;
      break;
    default:
      assert(0);
  }

  return n;
}

/*
 * Function fills the matrix A using the specified formula.
 */
double **matrix_fill() {
  const double q = 1.001 - 2 * M * 0.001;

  double **matrix;
  if ((matrix = matrix_create(n, n)) == NULL) {
    return NULL;
  }
  for (unsigned i = 0; i < n; i++) {
    for (unsigned j = 0; j < n; j++) {
      if (i == j) {
        matrix[i][j] = pow(q - 1, i + j);
```

```
        } else {
          matrix[i][j] = pow(q, i + j) + 0.1 * (j - i);
        }
      }
    }

  return matrix;
}
```

Глава 4

Тестирование

Тестирование правильности решения систем линейных алгебраических уравнений программой проводилось как с помощью проверки результатов работы программы путем подстановки их в исходную систему, так и с помощью сравнения результат работ двух методов. Тестирование нахождения обратных матриц и определителя проверялось вручную для матриц, размером меньше $3x3$, и с помощью пакета Maple для матриц больше порядка. Ниже приведены условия тестов в виде расширенных матриц систем алгебраических уравнений и результаты работы программы. Первые три теста взяты из варианта №13 тестовых заданий.

Тест #1

$$\left(\begin{array}{cccc|c} 3 & -2 & 2 & -2 & 8 \\ 2 & -1 & 2 & 0 & 4 \\ 2 & 1 & 4 & 8 & -1 \\ 1 & 3 & -6 & 2 & 3 \end{array}\right)$$

Результат работы метода Гаусса:
> determinant:
24
> inverse matrix:
-0.5 1.333333333 -0.1666666667 0.1666666667
-5 8.666666667 -1.333333333 0.3333333333
-2 3.5 -0.5 0
1.75 -3.166666667 0.5833333333 -0.08333333333
> solution:
2 -3 -1.5 0.5

Результат работы метода Гаусса с выбором главного элемента:
> solution:
2 -3 -1.5 0.5

Результат работы метода верхней релаксации:
> log: 235 iterations made
> solution:
-nan -nan -nan -nan

Тест #2

$$\left(\begin{array}{cccc|c} 2 & 3 & 1 & 2 & 4 \\ 4 & 3 & 1 & 1 & 5 \\ 1 & -7 & -1 & -2 & 7 \\ 2 & 5 & 1 & 1 & 1 \end{array}\right)$$

Результат работы метода Гаусса:
> determinant:
2
> inverse matrix:
-1 2 -1 -2
-1 1.5 -1 -1.5
8 -14.5 9 16.5
-1 3 -2 -4
> solution:
-3 -5 39 -7

Результат работы метода Гаусса с выбором главного элемента:
> solution:
-3 -5 39 -7

Результат работы метода верхней релаксации:
> log: 271 iterations made
> solution:
-nan -nan -nan -nan

Тест #3

$$\left(\begin{array}{cccc|c} 1 & -1 & 1 & -1 & 0 \\ 4 & -1 & 0 & -1 & 0 \\ 2 & 1 & -2 & 1 & 0 \\ 5 & 1 & 0 & -4 & 0 \end{array}\right)$$

Результат работы метода Гаусса:
> determinant: 0 > inverse matrix: does not exist > solution: cannot be found using Gaussian elimination method

Тест #4

$$\left(\begin{array}{ccc|c} 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 3 \\ 1 & 0 & 0 & 4 \end{array}\right)$$

Результат работы метода Гаусса:
> determinant:
-1
> inverse matrix:
0 0 1

0 1 0
1 0 -1
$>$ solution:
4 3 -2

Результат работы метода Гаусса с выбором главного элемента:
$>$ solution:
4 3 -2

Результат работы метода верхней релаксации:
$>$ log: 1 iterations made
$>$ solution:
2.6666666667 4 -nan

Тест #5

$$\left(\begin{array}{cc|c} 1 & 1 & 0 \\ 1 & 2 & 1 \end{array}\right)$$

Результат работы метода Гаусса:
$>$ determinant:
1
$>$ inverse matrix:
2 -1
-1 1
$>$ solution:
-1 1

Результат работы метода Гаусса с выбором главного элемента:
$>$ solution:
-1 1

Результат работы метода верхней релаксации:
$>$ log: 19 iterations made
$>$ solution:
-1.000000003 1.000000001

Тест #6

$$\left(\begin{array}{ccc|c} 3 & -1 & 1 & 3 \\ 1 & 1 & 1 & 5 \\ 4 & -1 & 4 & 5 \end{array}\right)$$

Результат работы метода Гаусса:
$>$ determinant:
10
$>$ inverse matrix:
0.5 0.3 -0.2
0 0.8 -0.2
-0.5 -0.1 0.4

$>$ solution:
2 3 0

Результат работы метода Гаусса с выбором главного элемента:
$>$ solution:
2 3 0

Результат работы метода верхней релаксации:
$>$ log: 25 iterations made
$>$ solution:
2.000000001 3.000000001 0

Тест #7

$$\left(\begin{array}{ccc|c} 2 & -1 & 1 & 6 \\ 1 & 2 & -1 & -1 \\ 1 & -1 & 2 & 11 \end{array}\right)$$

Результат работы метода Гаусса:
$>$ determinant:
6
$>$ inverse matrix:
0.5 0.1666666667 -0.1666666667
-0.5 0.5 0.5
-0.5 0.1666666667 0.8333333333
$>$ solution:
1 2 6

Результат работы метода Гаусса с выбором главного элемента:
$>$ solution:
1 2 6

Результат работы метода верхней релаксации:
$>$ log: 58 iterations made
$>$ solution:
0.9999999987 2.000000002 6.000000002

Ввиду того, что лишь в последних двух тестах матрицы были положительно определенными, результаты работы метода верхней релаксации в большинстве других были далеки от истинных. В целом, при $\omega = \frac{4}{3}$ метод верхней релаксации получал решение, достаточно близкое к точному менее, чем за 100 итераций.

Глава 5

Заключение

В ходе работы мной были освоены методы решения систем линейных алгебраических уравнений, а именно: метод верхней релаксации и метод Гаусса. Из результатов легко убедиться в том, что ввиду ограниченной применимости метода верхней релаксации, порой он оказывается неприменимым. Тем не менее, при аккуратном подборе итерационного параметра, он может оказаться даже быстрей метода Гаусса. Преимуществом метода Гаусса же является простота его реализиции и широкая применимость в задачах, связанных с исследованием матриц, в том числе нахождением обратной матрицы и детерминанта.

Литература

[1] Самарский А.А. Введение в численные методы. СПб., 2009. С. 91.

[2] Белоусов И.В. Матрицы и определители. Кишинев, 2006. С. 68.

[3] Самарский А.А. Введение в численные методы. СПб., 2009. С. 102.