

**Московский авиационный институт
(национальный исследовательский университет)**

**Институт №8 «Информационные технологии и прикладная
математика»**

Кафедра 806 «Вычислительная математика и программирование»

Лабораторные работы по курсу «Численные методы»

Студент: Г. С. Будайчиев
Преподаватель: Д. Е. Пивоваров
Группа: М8О-303Б-21
Дата:
Оценка:
Подпись:

Москва, 2024

1.1 LU - разложение матриц

1 Постановка задачи

Реализовать алгоритм LU - разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.

Вариант: 3

$$\begin{cases} 9x_1 - 5x_2 - 6x_3 + 3x_4 = -8 \\ x_1 - 7x_2 + x_3 = 38 \\ 3x_1 - 4x_2 + 9x_3 = 47 \\ 6x_1 - x_2 + 9x_3 + 8x_4 = -8 \end{cases}$$

2 Результаты работы

```
Solving system:
3.94746e-16
-5
3
-5

Determinant:
-4239

InverseMatrix:
0.111347      -0.149328      0.132578      -0.0417551
0.0113234     -0.167728      0.0304317     -0.00424628
-0.032083     -0.02477       0.0804435     0.0120311
-0.0460014    0.118896      -0.186129     0.142251
```

Рис. 1: Вывод программы в консоли

3 Исходный код

Общий файл для всех подзадач 1 лабораторной работы:

```
1  #ifndef MATRIX_H
2  #define MATRIX_H
3
4  #include <vector>
5  #include <map>
6  #include <utility>
7  #include <memory>
8  #include <cmath>
9  #include <string>
10 #include <fstream>
11 #include <complex>
12
13 namespace numeric {
14
15     template<class T>
16     class EigenResult {
17     public:
18         std::vector<T> eigenValues;
19         std::vector<std::vector<T>> eigenVectors;
20
21         explicit EigenResult(size_t size) : eigenValues(size), eigenVectors(size, std::vector<T>(size)) {}
22     };
23
24     template<class T, template<typename> class Container = std::vector>
25     class AbstractMatrix {
26     protected:
27         size_t _rows;
28         size_t _cols;
29     public:
30         virtual ~AbstractMatrix() = default;
31
32         virtual size_t rows() const;
33
34         virtual size_t cols() const;
35
36         virtual Container<T>& operator[](size_t i) = 0;
37
38         virtual const Container<T>& operator[](size_t i) const = 0;
39
40     };
41
42     template<class T, template<typename> class Container>
43     size_t AbstractMatrix<T, Container>::rows() const {
44         return _rows;
45     }
```

```

46
47 template<class T, template<typename> class Container>
48 size_t AbstractMatrix<T, Container>::cols() const {
49     return _cols;
50 }
51
52 template<class T>
53 class Matrix : public AbstractMatrix<T, std::vector> {
54 protected:
55     std::vector<std::vector<T>> data;
56     size_t _rows;
57     size_t _cols;
58 public:
59     explicit Matrix(const std::vector<std::vector<T>>& data);
60
61     explicit Matrix(size_t rows, size_t cols);
62
63     Matrix(const Matrix& other);
64
65     Matrix<T>& operator=(const Matrix& other);
66
67     Matrix<T>& operator+=(const Matrix<T>& rhs);
68
69     Matrix<T>& operator-=(const Matrix<T>& rhs);
70
71     Matrix<T>& operator*=(const Matrix<T>& rhs);
72
73     size_t rows() const override;
74
75     size_t cols() const override;
76
77     std::vector<T>& operator[](size_t i);
78
79     const std::vector<T>& operator[](size_t i) const;
80
81     Matrix<T> transpose() const;
82
83     ~Matrix() override;
84
85     static Matrix<T> eye(size_t size);
86 };
87
88 template<class T>
89 class SquareMatrix : public AbstractMatrix<T, std::vector> {
90 private:
91     std::vector<std::vector<T>> data;
92     size_t size;
93 public:
94     explicit SquareMatrix(const std::vector<std::vector<T>>& data);

```

```

95
96     explicit SquareMatrix(size_t size);
97
98     SquareMatrix(const SquareMatrix& other);
99
100    SquareMatrix<T>& operator=(const SquareMatrix& other);
101
102    SquareMatrix<T>& operator+=(const SquareMatrix<T>& rhs);
103
104    SquareMatrix<T>& operator-=(const SquareMatrix<T>& rhs);
105
106    SquareMatrix<T>& operator*=(const SquareMatrix<T>& rhs);
107
108    size_t rows() const override;
109
110    size_t cols() const override;
111
112    std::vector<T>& operator[](size_t i);
113
114    const std::vector<T>& operator[](size_t i) const;
115
116    SquareMatrix<T> transpose() const;
117
118    explicit operator Matrix<T>() const;
119
120    ~SquareMatrix() override;
121
122 };
123
124 template<class T>
125 SquareMatrix<T>::operator Matrix<T>() const {
126     Matrix<T> result(this->size, this->size);
127     for (size_t i = 0; i < this->size; ++i) {
128         for (size_t j = 0; j < this->size; ++j) {
129             result[i][j] = this->data[i][j];
130         }
131     }
132     return result;
133 }
134
135 template<class T>
136 SquareMatrix<T>& SquareMatrix<T>::operator=(const SquareMatrix& other) {
137     if (this != &other) {
138         data = other.data;
139         size = other.size;
140     }
141     return *this;
142 }
143

```

```

144
145     template<class T>
146     SquareMatrix<T>::SquareMatrix(const SquareMatrix& other) : data(other.data), size(
        other.cols()) {
147
148     }
149
150     template<class T>
151     Matrix<T>& Matrix<T>::operator=(const Matrix& other) {
152         if (this != &other) {
153             data = other.data;
154             _rows = other._rows;
155             _cols = other._cols;
156         }
157         return *this;
158     }
159
160     template<class T>
161     Matrix<T>::Matrix(const Matrix& other) : data(other.data), _rows(other._rows),
        _cols(other._cols) {
162
163     }
164
165     template<typename T>
166     class Row {
167     private:
168         std::map<size_t, T> row_data;
169
170     public:
171         T& operator[](size_t col) {
172             return row_data[col];
173         }
174
175         const T& operator[](size_t col) const {
176             auto it = row_data.find(col);
177             if (it != row_data.end()) {
178                 return it->second;
179             }
180             static const T defaultValue{};
181             return defaultValue;
182         }
183     };
184
185     template<class T>
186     class SparseMatrix : public AbstractMatrix<T, Row> {
187     private:
188         size_t _rows;
189         size_t _cols;
190

```

```

191         std::map<size_t, Row<T>> data;
192
193
194     public:
195         explicit SparseMatrix(size_t rows, size_t cols);
196
197         SparseMatrix(const SparseMatrix& other);
198
199         SparseMatrix<T>& operator=(const SparseMatrix& other);
200
201         SparseMatrix<T>& operator+=(const SparseMatrix<T>& rhs);
202
203         SparseMatrix<T>& operator-=(const SparseMatrix<T>& rhs);
204
205         SparseMatrix<T>& operator*=(const SparseMatrix<T>& rhs);
206
207         size_t rows() const override;
208
209         size_t cols() const override;
210
211         Row<T>& operator[](size_t row) override;
212
213         const Row<T>& operator[](size_t row) const override;
214
215         SparseMatrix<T> transpose() const;
216
217         ~SparseMatrix() override;
218     };
219
220     template<class T>
221     Matrix<T>::Matrix(size_t rows, size_t cols) : data(rows, std::vector<T>(cols)),
222         _rows(rows), _cols(cols) {
223     }
224
225     template<class T>
226     Matrix<T>::Matrix(const std::vector<std::vector<T>>& data)
227         : data(data), _rows(data.size()), _cols(data.begin()->size()) {
228     }
229
230
231     template<class T>
232     size_t Matrix<T>::rows() const {
233         return _rows;
234     }
235
236     template<class T>
237     size_t Matrix<T>::cols() const {
238         return _cols;

```

```

239     }
240
241     template<class T>
242     std::vector<T>& Matrix<T>::operator[](size_t i) {
243         return data[i];
244     }
245
246     template<class T>
247     const std::vector<T>& Matrix<T>::operator[](size_t i) const {
248         return data[i];
249     }
250
251     template<class T>
252     Matrix<T> Matrix<T>::transpose() const {
253         Matrix<T> transposedMatrix(_cols, _rows);
254         for (size_t i = 0; i < rows(); ++i) {
255             for (size_t j = 0; j < cols(); ++j) {
256                 transposedMatrix[j][i] = (*this)[i][j];
257             }
258         }
259         return transposedMatrix;
260     }
261
262     template<class T>
263     Matrix<T>::~Matrix() = default;
264
265     template<class T>
266     Matrix<T> Matrix<T>::eye(size_t size) {
267         Matrix<T> matrix(size, size);
268         for (int i = 0; i < size; ++i) {
269             matrix[i][i] = 1;
270         }
271         return matrix;
272     }
273
274     template<class T>
275     Matrix<T> operator+(const Matrix<T>& lhs, const Matrix<T>& rhs) {
276         if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols()) {
277             throw std::invalid_argument("Matrix dimensions must match for addition.");
278         }
279
280         Matrix<T> result(lhs.rows(), lhs.cols());
281         for (size_t i = 0; i < lhs.rows(); ++i) {
282             for (size_t j = 0; j < lhs.cols(); ++j) {
283                 result[i][j] = lhs[i][j] + rhs[i][j];
284             }
285         }
286         return result;
287     }

```



```

288
289 template<class T>
290 std::vector<T> operator*(const std::vector<T>& vec, T scalar) {
291     std::vector<T> result(vec.size());
292     for (size_t i = 0; i < vec.size(); ++i) {
293         result[i] = vec[i] * scalar;
294     }
295     return result;
296 }
297
298 template<class T>
299 std::vector<T> operator*(T scalar, const std::vector<T>& vec) {
300     return vec * scalar;
301 }
302
303 template<class T>
304 Matrix<T> operator-(const Matrix<T>& lhs, const Matrix<T>& rhs) {
305     if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols()) {
306         throw std::invalid_argument("Matrix dimensions must match for subtraction."
307                                     );
308     }
309     Matrix<T> result(lhs.rows(), lhs.cols());
310     for (size_t i = 0; i < lhs.rows(); ++i) {
311         for (size_t j = 0; j < lhs.cols(); ++j) {
312             result[i][j] = lhs[i][j] - rhs[i][j];
313         }
314     }
315     return result;
316 }
317
318 template<class T>
319 Matrix<T> operator*(const Matrix<T>& lhs, const Matrix<T>& rhs) {
320     if (lhs.cols() != rhs.rows()) {
321         throw std::invalid_argument(
322             "The number of columns in the first matrix must match the number of rows
323              in the second.");
324     }
325     Matrix<T> result(lhs.rows(), rhs.cols());
326     for (size_t i = 0; i < lhs.rows(); ++i) {
327         for (size_t j = 0; j < rhs.cols(); ++j) {
328             T sum = T();
329             for (size_t k = 0; k < lhs.cols(); ++k) {
330                 sum += lhs[i][k] * rhs[k][j];
331             }
332             result[i][j] = sum;
333         }
334     }

```

```

335     return result;
336 }
337
338 template<class T>
339 Matrix<T> operator*(const Matrix<T>& matrix, T scalar) {
340     Matrix<T> result(matrix.rows(), matrix.cols());
341     for (size_t i = 0; i < matrix.rows(); ++i) {
342         for (size_t j = 0; j < matrix.cols(); ++j) {
343             result[i][j] = matrix[i][j] * scalar;
344         }
345     }
346     return result;
347 }
348
349 template<class T>
350 Matrix<T> operator*(T scalar, const Matrix<T>& matrix) {
351     return matrix * scalar;
352 }
353
354 template<class T>
355 Matrix<T> operator/(const Matrix<T>& matrix, T scalar) {
356     if (scalar == 0) {
357         throw std::invalid_argument("Zero division");
358     }
359     Matrix<T> result(matrix.rows(), matrix.cols());
360     for (size_t i = 0; i < matrix.rows(); ++i) {
361         for (size_t j = 0; j < matrix.cols(); ++j) {
362             result[i][j] = matrix[i][j] / scalar;
363         }
364     }
365     return result;
366 }
367
368 template<class T>
369 std::vector<T> operator*(const Matrix<T>& matrix, const std::vector<T>& vec) {
370     if (matrix.cols() != vec.size()) {
371         throw std::invalid_argument("The number of columns in the matrix must match
372                                     the size of the vector.");
373     }
374     std::vector<T> result(matrix.rows(), T());
375     for (size_t i = 0; i < matrix.rows(); ++i) {
376         for (size_t k = 0; k < matrix.cols(); ++k) {
377             result[i] += matrix[i][k] * vec[k];
378         }
379     }
380     return result;
381 }
382

```

```

383 template<class T>
384 std::vector<T> operator*(const std::vector<T>& vec, const Matrix<T>& matrix) {
385     if (vec.size() != matrix.rows()) {
386         throw std::invalid_argument("The size of the vector must match the number
           of rows in the matrix.");
387     }
388
389     std::vector<T> result(matrix.cols(), T());
390     for (size_t j = 0; j < matrix.cols(); ++j) {
391         for (size_t i = 0; i < matrix.rows(); ++i) {
392             result[j] += vec[i] * matrix[i][j];
393         }
394     }
395     return result;
396 }
397
398
399
400
401 template<class T>
402 Matrix<T>& Matrix<T>::operator+=(const Matrix<T>& rhs) {
403     if (_rows != rhs._rows || _cols != rhs._cols) {
404         throw std::invalid_argument("Matrix dimensions must match for addition.");
405     }
406     for (size_t i = 0; i < _rows; ++i) {
407         for (size_t j = 0; j < _cols; ++j) {
408             data[i][j] += rhs.data[i][j];
409         }
410     }
411     return *this;
412 }
413
414 template<class T>
415 Matrix<T>& Matrix<T>::operator-=(const Matrix<T>& rhs) {
416     if (_rows != rhs._rows || _cols != rhs._cols) {
417         throw std::invalid_argument("Matrix dimensions must match for subtraction."
           );
418     }
419     for (size_t i = 0; i < _rows; ++i) {
420         for (size_t j = 0; j < _cols; ++j) {
421             data[i][j] -= rhs.data[i][j];
422         }
423     }
424     return *this;
425 }
426
427 template<class T>
428 Matrix<T>& Matrix<T>::operator*=(const Matrix<T>& rhs) {
429     if (_cols != rhs._rows) {

```

```

430         throw std::invalid_argument(
431             "The number of columns in the first matrix must match the number of rows
              in the second for multiplication.");
432     }
433
434     Matrix<T> result(_rows, rhs._cols);
435     for (size_t i = 0; i < _rows; ++i) {
436         for (size_t j = 0; j < rhs._cols; ++j) {
437             for (size_t k = 0; k < _cols; ++k) {
438                 result.data[i][j] += data[i][k] * rhs.data[k][j];
439             }
440         }
441     }
442
443     *this = std::move(result);
444     return *this;
445 }
446
447 template<class T>
448 SquareMatrix<T>::SquareMatrix(const std::vector<std::vector<T>>& data) : data(data)
    , size(data.size()) {
449     if (data.size() != data.begin()->size()) {
450         throw std::invalid_argument("Not square matrix");
451     }
452 }
453
454 template<class T>
455 SquareMatrix<T>::SquareMatrix(size_t size) : size(size), data(size, std::vector<T>(
    size)) {
456 }
457
458 template<class T>
459 size_t SquareMatrix<T>::rows() const {
460     return size;
461 }
462
463 template<class T>
464 size_t SquareMatrix<T>::cols() const {
465     return size;
466 }
467
468 template<class T>
469 std::vector<T>& SquareMatrix<T>::operator[](size_t i) {
470     return data[i];
471 }
472
473 template<class T>
474 const std::vector<T>& SquareMatrix<T>::operator[](size_t i) const {
475     return data[i];

```

```

476     }
477
478     template<class T>
479     SquareMatrix<T> SquareMatrix<T>::transpose() const {
480         SquareMatrix<T> transposedMatrix(size);
481         for (size_t i = 0; i < rows(); ++i) {
482             for (size_t j = 0; j < cols(); ++j) {
483                 transposedMatrix[j][i] = (*this)[i][j];
484             }
485         }
486         return transposedMatrix;
487     }
488
489     template<class T>
490     SquareMatrix<T>::~SquareMatrix() = default;
491
492     template<class T>
493     SquareMatrix<T> operator+(const SquareMatrix<T>& lhs, const SquareMatrix<T>& rhs) {
494         if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols()) {
495             throw std::invalid_argument("Matrix dimensions must match for addition.");
496         }
497
498         SquareMatrix<T> result(lhs.cols());
499         for (size_t i = 0; i < lhs.rows(); ++i) {
500             for (size_t j = 0; j < lhs.cols(); ++j) {
501                 result[i][j] = lhs[i][j] + rhs[i][j];
502             }
503         }
504         return result;
505     }
506
507     template<class T>
508     SquareMatrix<T> operator-(const SquareMatrix<T>& lhs, const SquareMatrix<T>& rhs) {
509         if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols()) {
510             throw std::invalid_argument("Matrix dimensions must match for subtraction.");
511         }
512
513         SquareMatrix<T> result(lhs.cols());
514         for (size_t i = 0; i < lhs.rows(); ++i) {
515             for (size_t j = 0; j < lhs.cols(); ++j) {
516                 result[i][j] = lhs[i][j] - rhs[i][j];
517             }
518         }
519         return result;
520     }
521
522     template<class T>
523     SquareMatrix<T> operator*(const SquareMatrix<T>& lhs, const SquareMatrix<T>& rhs) {

```

```

524     if (lhs.cols() != rhs.rows()) {
525         throw std::invalid_argument(
526             "The number of columns in the first matrix must match the number of rows
              in the second.");
527     }
528
529     SquareMatrix<T> result(rhs.cols());
530     for (size_t i = 0; i < lhs.rows(); ++i) {
531         for (size_t j = 0; j < rhs.cols(); ++j) {
532             T sum = T();
533             for (size_t k = 0; k < lhs.cols(); ++k) {
534                 sum += lhs[i][k] * rhs[k][j];
535             }
536             result[i][j] = sum;
537         }
538     }
539     return result;
540 }
541
542 template<class T>
543 SquareMatrix<T>& SquareMatrix<T>::operator+=(const SquareMatrix<T>& rhs) {
544     if (size != rhs.size || size != rhs.size) {
545         throw std::invalid_argument("Matrix dimensions must match for addition.");
546     }
547     for (size_t i = 0; i < size; ++i) {
548         for (size_t j = 0; j < size; ++j) {
549             data[i][j] += rhs[i][j];
550         }
551     }
552     return *this;
553 }
554
555 template<class T>
556 SquareMatrix<T>& SquareMatrix<T>::operator-=(const SquareMatrix<T>& rhs) {
557     if (size != rhs.size || size != rhs.size) {
558         throw std::invalid_argument("Matrix dimensions must match for subtraction."
              );
559     }
560     for (size_t i = 0; i < size; ++i) {
561         for (size_t j = 0; j < size; ++j) {
562             data[i][j] -= rhs[i][j];
563         }
564     }
565     return *this;
566 }
567
568 template<class T>
569 SquareMatrix<T>& SquareMatrix<T>::operator*=(const SquareMatrix<T>& rhs) {
570     if (size != rhs.size) {

```

```

571         throw std::invalid_argument(
572             "The number of columns in the first matrix must match the number of rows
              in the second for multiplication.");
573     }
574
575     SquareMatrix<T> result(size);
576     for (size_t i = 0; i < size; ++i) {
577         for (size_t j = 0; j < rhs.size; ++j) {
578             for (size_t k = 0; k < size; ++k) {
579                 result.data[i][j] += data[i][k] * rhs.data[k][j];
580             }
581         }
582     }
583
584     *this = std::move(result);
585     return *this;
586 }
587
588
589 template<class T>
590 SparseMatrix<T>::SparseMatrix(size_t rows, size_t cols) : _rows(rows), _cols(cols)
    {
591 }
592
593 template<class T>
594 size_t SparseMatrix<T>::rows() const {
595     return _rows;
596 }
597
598 template<class T>
599 size_t SparseMatrix<T>::cols() const {
600     return _cols;
601 }
602
603 template<class T>
604 Row<T>& SparseMatrix<T>::operator[](size_t row) {
605     return data[row];
606 }
607
608 template<class T>
609 const Row<T>& SparseMatrix<T>::operator[](size_t row) const {
610     auto it = data.find(row);
611     if (it != data.end()) {
612         return it->second;
613     }
614     else {
615         static const Row<T> emptyRow{};
616         return emptyRow;
617     }

```

```

618     }
619
620     template<typename T>
621     T operator*(const std::vector<T>& v1, const std::vector<T>& v2) {
622         if (v1.size() != v2.size()) {
623             throw std::invalid_argument("Vectors must be of the same length.");
624         }
625
626         T result = 0;
627         for (size_t i = 0; i < v1.size(); ++i) {
628             result += v1[i] * v2[i];
629         }
630         return result;
631     }
632
633
634     template<class T>
635     SparseMatrix<T> SparseMatrix<T>::transpose() const {
636         SparseMatrix<T> transposedMatrix(cols(), rows());
637         for (size_t i = 0; i < rows(); ++i) {
638             for (size_t j = 0; j < cols(); ++j) {
639                 transposedMatrix[j][i] = (*this)[i][j];
640             }
641         }
642         return transposedMatrix;
643     }
644
645     template<class T>
646     SparseMatrix<T>::~SparseMatrix() = default;
647
648     template<class T>
649     SparseMatrix<T> operator+(const SparseMatrix<T>& lhs, const SparseMatrix<T>& rhs) {
650         if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols()) {
651             throw std::invalid_argument("Matrix dimensions must match for addition.");
652         }
653
654         SparseMatrix<T> result(lhs.rows(), lhs.cols());
655         for (size_t i = 0; i < lhs.rows(); ++i) {
656             for (size_t j = 0; j < lhs.cols(); ++j) {
657                 result[i][j] = lhs[i][j] + rhs[i][j];
658             }
659         }
660         return result;
661     }
662
663     template<class T>
664     SparseMatrix<T> operator-(const SparseMatrix<T>& lhs, const SparseMatrix<T>& rhs) {
665         if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols()) {

```



```

666         throw std::invalid_argument("Matrix dimensions must match for subtraction."
667                                     );
668     }
669     SparseMatrix<T> result(lhs.rows(), lhs.cols());
670     for (size_t i = 0; i < lhs.rows(); ++i) {
671         for (size_t j = 0; j < lhs.cols(); ++j) {
672             result[i][j] = lhs[i][j] - rhs[i][j];
673         }
674     }
675     return result;
676 }
677
678 template<class T>
679 SparseMatrix<T> operator*(const SparseMatrix<T>& lhs, const SparseMatrix<T>& rhs) {
680     if (lhs.cols() != rhs.rows()) {
681         throw std::invalid_argument(
682             "The number of columns in the first matrix must match the number of rows
              in the second.");
683     }
684
685     SparseMatrix<T> result(lhs.rows(), rhs.cols());
686     for (size_t i = 0; i < lhs.rows(); ++i) {
687         for (size_t j = 0; j < rhs.cols(); ++j) {
688             T sum = T();
689             for (size_t k = 0; k < lhs.cols(); ++k) {
690                 sum += lhs[i][k] * rhs[k][j];
691             }
692             result[i][j] = sum;
693         }
694     }
695     return result;
696 }
697
698 template<typename T>
699 Matrix<T> outerProduct(const std::vector<T>& v1, const std::vector<T>& v2) {
700     Matrix<T> matrix(v1.size(), v2.size());
701     for (size_t i = 0; i < v1.size(); ++i) {
702         for (size_t j = 0; j < v2.size(); ++j) {
703             matrix[i][j] = v1[i] * v2[j];
704         }
705     }
706     return matrix;
707 }
708
709 template<class T>
710 SparseMatrix<T>::SparseMatrix(const SparseMatrix& other) : data(other.data), _rows(
    other._rows),
711     _cols(other._cols) {

```

```

712 }
713
714
715 template<class T>
716 SparseMatrix<T>& SparseMatrix<T>::operator=(const SparseMatrix& other) {
717     if (this != &other) {
718         data = other.data;
719         _rows = other._rows;
720         _cols = other._cols;
721     }
722     return *this;
723 }
724
725 template<class T>
726 SparseMatrix<T>& SparseMatrix<T>::operator+=(const SparseMatrix<T>& rhs) {
727     if (_rows != rhs._rows || _cols != rhs._cols) {
728         throw std::invalid_argument("Matrix dimensions must match for addition.");
729     }
730     for (size_t i = 0; i < _rows; ++i) {
731         for (size_t j = 0; j < _cols; ++j) {
732             data[i][j] += rhs[i][j];
733         }
734     }
735     return *this;
736 }
737
738 template<class T>
739 SparseMatrix<T>& SparseMatrix<T>::operator-=(const SparseMatrix<T>& rhs) {
740     if (_rows != rhs._rows || _cols != rhs._cols) {
741         throw std::invalid_argument("Matrix dimensions must match for addition.");
742     }
743     for (size_t i = 0; i < _rows; ++i) {
744         for (size_t j = 0; j < _cols; ++j) {
745             data[i][j] -= rhs[i][j];
746         }
747     }
748     return *this;
749 }
750
751 template<class T>
752 SparseMatrix<T>& SparseMatrix<T>::operator*=(const SparseMatrix<T>& rhs) {
753     if (_cols != rhs._rows) {
754         throw std::invalid_argument(
755             "The number of columns in the first matrix must match the number of rows
              in the second for multiplication.");
756     }
757
758     SparseMatrix<T> result(_rows, rhs._cols);
759     for (size_t i = 0; i < _rows; ++i) {

```

```

760         for (size_t j = 0; j < rhs._cols; ++j) {
761             for (size_t k = 0; k < _cols; ++k) {
762                 result[i][j] += data[i][k] * rhs[k][j];
763             }
764         }
765     }
766
767     *this = std::move(result);
768     return *this;
769 }
770
771 template<class T>
772 class LUMatrix {
773 private:
774     void init(const Matrix<T>& matrix);
775
776 public:
777     Matrix<T> L;
778     Matrix<T> U;
779
780     explicit LUMatrix(const Matrix<T>& matrix);
781
782     Matrix<T> solve();
783
784     T determinant();
785 };
786
787 template<class T>
788 T LUMatrix<T>::determinant() {
789     T deter = 1;
790     for (size_t i = 0; i < U.rows(); ++i) {
791         deter *= U[i][i];
792     }
793
794     return deter;
795 }
796
797 template<class T>
798 Matrix<T> LUMatrix<T>::solve() {
799     Matrix<T> result(U.rows(), U.cols() - U.rows());
800     for (int k = 0; k < U.cols() - U.rows(); ++k) {
801         for (int i = U.rows() - 1; i >= 0; --i) {
802             T sum = 0.0;
803             for (int j = i + 1; j < U.rows(); ++j) {
804                 sum += U[i][j] * result[j][k];
805             }
806             result[i][k] = (U[i][U.rows() + k] - sum) / U[i][i];
807         }
808     }

```

```

809
810     return result;
811 }
812
813 template<class T>
814 void LUMatrix<T>::init(const Matrix<T>& matrix) {
815     U = matrix;
816     for (size_t i = 0; i < matrix.rows(); ++i) {
817         T max = 0;
818         size_t row = i;
819         for (size_t k = i; k < matrix.rows(); ++k) {
820             if (std::fabs(matrix[k][i]) > max) {
821                 max = std::fabs(matrix[k][i]);
822                 row = k;
823             }
824         }
825
826         std::swap(U[i], U[row]);
827         std::swap(L[i], L[row]);
828
829         L[i][i] = 1;
830
831         for (size_t j = i + 1; j < matrix.rows(); ++j) {
832             double factor = U[j][i] / double(U[i][i]);
833             L[j][i] = factor;
834             for (size_t k = i; k < matrix.cols(); ++k) {
835                 U[j][k] -= factor * U[i][k];
836             }
837         }
838     }
839 }
840
841 template<class T>
842 LUMatrix<T>::LUMatrix(const Matrix<T>& matrix) : L(matrix.rows(), matrix.cols()), U
    (matrix.rows(), matrix.cols()) {
843     init(matrix);
844 }
845
846 template<class T>
847 Matrix<T> inputMatrix(const std::string& path) {
848     std::ifstream fin(path);
849     size_t n, m;
850     fin >> n >> m;
851     Matrix<T> matrix(n, m);
852     for (size_t i = 0; i < n; ++i) {
853         for (size_t j = 0; j < m; ++j) {
854             fin >> matrix[i][j];
855         }
856     }

```

```

857     return matrix;
858 }
859
860 template<class T>
861 std::vector<T> inputVector(const std::string& path) {
862     std::ifstream fin(path);
863     size_t n;
864     fin >> n;
865     std::vector<T> vec(n);
866     for (size_t i = 0; i < n; ++i) {
867         fin >> vec[i];
868     }
869     return vec;
870 }
871
872 template<class T>
873 void printMatrix(const AbstractMatrix<T>& matrix) {
874     for (size_t i = 0; i < matrix.rows(); ++i) {
875         for (size_t j = 0; j < matrix.cols(); ++j) {
876             std::cout << matrix[i][j] << "\t";
877         }
878         std::cout << "\n";
879     }
880 }
881
882 template<class T>
883 void printVector(const std::vector<T>& vector) {
884     for (size_t i = 0; i < vector.size(); ++i)
885         std::cout << vector[i] << " ";
886 }
887
888 template<class T>
889 std::vector<T> tridiagonalSolve(const AbstractMatrix<T>& matrix, const std::vector<
    T>& b) {
890     size_t n = matrix.rows();
891     if (matrix.cols() != n || b.size() != n) {
892         throw std::invalid_argument("Matrix must be square and the size of vector b
            must match.");
893     }
894
895     std::vector<T> C(n, 0);
896     std::vector<T> D(n, 0);
897     std::vector<T> x(n);
898
899     C[0] = matrix[0][1] / matrix[0][0];
900     D[0] = b[0] / matrix[0][0];
901
902     for (size_t i = 1; i < n; ++i) {
903         T m = 1 / (matrix[i][i] - matrix[i][i - 1] * C[i - 1]);

```

```

904         C[i] = i < n - 1 ? matrix[i][i + 1] * m : 0;
905         D[i] = (b[i] - matrix[i][i - 1] * D[i - 1]) * m;
906     }
907
908     x[n - 1] = D[n - 1];
909
910     for (int i = n - 2; i >= 0; --i) {
911         x[i] = D[i] - C[i] * x[i + 1];
912     }
913
914     return x;
915 }
916
917 template<class T>
918 double norm(const AbstractMatrix<T>& matrix) {
919     double norm = 0;
920     for (size_t i = 0; i < matrix.rows(); ++i) {
921         double currentSum = 0;
922         for (size_t j = 0; j < matrix.cols(); ++j) {
923             currentSum += fabs(matrix[i][j]);
924         }
925         norm = fmax(currentSum, norm);
926     }
927
928     return norm;
929 }
930
931 template<class T>
932 double norm(const std::vector<T>& vector) {
933     double norm = 0;
934     for (size_t i = 0; i < vector.size(); ++i) {
935         norm += pow(vector[i], 2);
936     }
937     return sqrt(norm);
938 }
939
940 template<class T>
941 std::vector<T> diffVector(const std::vector<T>& lhs, const std::vector<T>& rhs) {
942     if (lhs.size() != rhs.size())
943         throw std::invalid_argument("invalid args");
944     std::vector<T> result(lhs.size());
945     for (size_t i = 0; i < lhs.size(); ++i) {
946         result[i] = lhs[i] - rhs[i];
947     }
948     return result;
949 }
950
951 template<class T>

```

```

952 std::vector<T> iterationSolve(const AbstractMatrix<T>& matrix, const std::vector<T
    >& b, T eps) {
953     std::size_t n = matrix.rows();
954     std::vector<T> beta(n, T());
955     Matrix<T> alpha(n, n);
956     for (std::size_t i = 0; i < n; ++i) {
957         beta[i] = b[i] / matrix[i][i];
958         for (std::size_t j = 0; j < n; ++j) {
959             if (i == j) {
960                 alpha[i][j] = 0;
961             }
962             else {
963                 alpha[i][j] = -matrix[i][j] / matrix[i][i];
964             }
965         }
966     }
967
968     std::vector<T> x = beta;
969     std::vector<T> x_next(n, T());
970
971     bool continueIteration = true;
972
973     double a = norm(alpha);
974
975     std::cout << "\nNorm of matrix:\n";
976     std::cout << a << "\n";
977
978     size_t iter = 0;
979     while (continueIteration) {
980         continueIteration = false;
981
982         for (std::size_t i = 0; i < n; ++i) {
983             T sum = beta[i];
984             for (std::size_t j = 0; j < n; ++j) {
985                 sum += alpha[i][j] * x[j];
986             }
987             x_next[i] = sum;
988         }
989
990         if (a < 1) {
991             if (a / (1 - a) * norm(diffVector(x_next, x)) > eps) {
992                 continueIteration = true;
993             }
994         }
995         else {
996             if (norm(diffVector(x_next, x)) > eps) {
997                 continueIteration = true;
998             }
999         }

```

```

1000
1001     x = x_next;
1002     ++iter;
1003 }
1004 std::cout << "\nCount of iterations: " << iter << "\n";
1005 return x;
1006 }
1007
1008 template<class T>
1009 std::vector<T> SeidelSolve(const AbstractMatrix<T>& matrix, const std::vector<T>& b
    , T eps) {
1010     std::size_t n = matrix.rows();
1011     std::vector<T> beta(n, T());
1012     Matrix<T> alpha(n, n);
1013     Matrix<T> CMatrix(n, n);
1014     for (std::size_t i = 0; i < n; ++i) {
1015         beta[i] = b[i] / matrix[i][i];
1016         for (std::size_t j = 0; j < n; ++j) {
1017             if (i == j) {
1018                 alpha[i][j] = 0;
1019             }
1020             else {
1021                 alpha[i][j] = -matrix[i][j] / matrix[i][i];
1022             }
1023         }
1024     }
1025
1026     for (size_t i = 0; i < n; ++i) {
1027         for (size_t j = i; j < n; ++j) {
1028             CMatrix[i][j] = alpha[i][j];
1029         }
1030     }
1031     std::vector<T> x = beta;
1032     std::vector<T> x_next(n, T());
1033
1034     bool continueIteration = true;
1035
1036     double a = norm(alpha);
1037     double c = norm(CMatrix);
1038     std::cout << "\nNorm of matrix:\n";
1039     std::cout << a << "\n";
1040     std::cout << "\nNorm of C matrix:\n";
1041     std::cout << c << "\n";
1042
1043     size_t iter = 0;
1044     while (continueIteration) {
1045         continueIteration = false;
1046
1047         x_next = x;

```



```

1048     for (std::size_t i = 0; i < n; ++i) {
1049         T sum = beta[i];
1050         for (size_t j = 0; j < n; ++j) {
1051             sum += alpha[i][j] * x_next[j];
1052         }
1053         x_next[i] = sum;
1054     }
1055
1056     if (a < 1) {
1057         if (c / (1 - a) * norm(diffVector(x_next, x)) > eps) {
1058             continueIteration = true;
1059         }
1060     }
1061     else {
1062         if (norm(diffVector(x_next, x)) > eps) {
1063             continueIteration = true;
1064         }
1065     }
1066
1067     x = x_next;
1068     ++iter;
1069 }
1070 std::cout << "\nCount of iterations: " << iter << "\n";
1071 return x;
1072 }
1073
1074 template<class T>
1075 void applyRotation(AbstractMatrix<T>& matrix, std::vector<std::vector<T>>&
    eigenVectors, size_t p, size_t q, T c, T s) {
1076     size_t n = matrix.rows();
1077     for (size_t i = 0; i < n; ++i) {
1078         T mpi = matrix[i][p];
1079         T mqi = matrix[i][q];
1080         matrix[i][p] = c * mpi + s * mqi;
1081         matrix[i][q] = -s * mpi + c * mqi;
1082
1083         T epi = eigenVectors[i][p];
1084         T eqi = eigenVectors[i][q];
1085         eigenVectors[i][p] = c * epi + s * eqi;
1086         eigenVectors[i][q] = -s * epi + c * eqi;
1087     }
1088
1089     for (size_t j = 0; j < n; ++j) {
1090         T mpj = matrix[p][j];
1091         T mqj = matrix[q][j];
1092         matrix[p][j] = c * mpj + s * mqj;
1093         matrix[q][j] = -s * mpj + c * mqj;
1094     }
1095 }

```

```

1096
1097
1098 template<class T>
1099 EigenResult<T> findEigenvaluesAndEigenvectors(Matrix<T>& inputMatrix, double eps) {
1100     size_t n = inputMatrix.rows();
1101     EigenResult<T> result(n);
1102     auto& eigenVectors = result.eigenVectors;
1103     auto& eigenValues = result.eigenValues;
1104     Matrix<T> matrix = inputMatrix;
1105
1106     for (size_t i = 0; i < n; ++i) {
1107         eigenVectors[i][i] = 1;
1108     }
1109
1110     double offDiagonalNorm;
1111     do {
1112         offDiagonalNorm = 0.0;
1113         for (size_t p = 0; p < n; ++p) {
1114             for (size_t q = p + 1; q < n; ++q) {
1115                 offDiagonalNorm += matrix[p][q] * matrix[p][q];
1116             }
1117         }
1118
1119         if (sqrt(offDiagonalNorm) < eps)
1120             break;
1121
1122         for (size_t p = 0; p < n; ++p) {
1123             for (size_t q = p + 1; q < n; ++q) {
1124                 T apq = matrix[p][q];
1125                 if (fabs(apq) > eps) {
1126                     T app = matrix[p][p];
1127                     T aqq = matrix[q][q];
1128                     T tau = (aqq - app) / (2 * apq);
1129                     T t = (tau / fabs(tau)) * (1.0 / (fabs(tau) + sqrt(1.0 + tau *
1130                         tau)));
1131                     T c = 1 / sqrt(1 + t * t);
1132                     T s = t * c;
1133
1134                     applyRotation(matrix, eigenVectors, p, q, c, s);
1135                 }
1136             }
1137         } while (true);
1138
1139         for (size_t i = 0; i < n; ++i) {
1140             eigenValues[i] = matrix[i][i];
1141         }
1142
1143         return result;

```

```

1144     }
1145
1146     template<class T>
1147     class QRMatrix {
1148     public:
1149         Matrix<T> Q;
1150         Matrix<T> R;
1151
1152         explicit QRMatrix(const Matrix<T>& matrix) : Q(Matrix<T>::eye(matrix.rows())),
1153             R(matrix) {
1154             init();
1155         }
1156     private:
1157         void init() {
1158             size_t m = R.rows();
1159             size_t n = R.cols();
1160
1161             for (size_t j = 0; j < n - 1; ++j) {
1162                 T norm_x = 0;
1163                 for (size_t i = j; i < m; ++i) {
1164                     norm_x += R[i][j] * R[i][j];
1165                 }
1166                 norm_x = std::sqrt(norm_x);
1167
1168                 std::vector<T> v(m, 0);
1169                 T alpha = R[j][j] > 0 ? -norm_x : norm_x;
1170                 for (size_t i = j; i < m; ++i) {
1171                     v[i] = R[i][j] - ((i == j) ? alpha : 0);
1172                 }
1173
1174                 Matrix<T> H = Matrix<T>::eye(n) - 2.0 * (outerProduct(v, v) / (v * v));
1175
1176                 R = H * R;
1177                 Q *= H;
1178             }
1179         }
1180     };
1181
1182     template<class T>
1183     EigenResult<std::complex<T>> findEigenvaluesAndEigenvectorsByQR(Matrix<T>&
1184         inputMatrix, double eps1, double eps2) {
1185         bool continueIteration = true;
1186         Matrix<T> A = inputMatrix;
1187         std::vector<std::complex<T>> prevEigenvalues(inputMatrix.cols());
1188
1189         while (continueIteration) {
1190             QRMatrix<T> QR(A);
1191             A = QR.R * QR.Q;

```

```

1191
1192     for (int i = 0; i < A.cols(); ++i) {
1193         T underDiagonal = 0;
1194         for (int j = i + 1; j < A.rows(); ++j) {
1195             underDiagonal += A[j][i] * A[j][i];
1196         }
1197         underDiagonal = std::sqrt(underDiagonal);
1198         if (i < A.cols() - 1 && underDiagonal > eps1) {
1199             T a = A[i][i], b = A[i][i + 1], c = A[i + 1][i], d = A[i + 1][i +
1200                 1];
1201             T tr = a + d;
1202             T det = a * d - b * c;
1203             T s = std::sqrt(std::abs(tr * tr / 4 - det));
1204             if (std::abs(std::complex<T>(tr / 2, s) - prevEigenvalues[i]) < eps2
1205                 ) {
1206                 continueIteration = false;
1207             }
1208             prevEigenvalues[i] = std::complex<T>(tr / 2, s);
1209         }
1210         else if (i < A.cols() - 1) {
1211             if (underDiagonal < eps1) {
1212                 continueIteration = false;
1213             }
1214         }
1215     }
1216
1217     size_t i = 0;
1218     EigenResult<std::complex<T>> result(A.cols());
1219     while (i < A.cols()) {
1220         T underDiagonal = 0;
1221         for (int j = i + 1; j < A.rows(); ++j) {
1222             underDiagonal += A[j][i] * A[j][i];
1223         }
1224         underDiagonal = std::sqrt(underDiagonal);
1225         if (i < A.cols() - 1 && underDiagonal > eps1) {
1226             T a = A[i][i], b = A[i][i + 1], c = A[i + 1][i], d = A[i + 1][i + 1];
1227             T tr = a + d;
1228             T det = a * d - b * c;
1229             T s = std::sqrt(std::abs(tr * tr / 4 - det));
1230             result.eigenValues[i] = std::complex<T>(tr / 2, s);
1231             result.eigenValues[i + 1] = std::complex<T>(tr / 2, -s);
1232             i += 2;
1233         }
1234         else {
1235             result.eigenValues[i] = std::complex<T>(A[i][i], 0);
1236             i++;
1237         }
1238     }

```

```

1238
1239     return result;
1240 }
1241 } // numeric
1242
1243 #endif //MATRIX_H

```

Коэффициенты перед иксами системы:

```

1 9 -5 -6 3
2 1 -7 1 0
3 3 -4 9 0
4 6 -1 9 8

```

Вектор b:

```

1 -8
2 38
3 47
4 -8

```

```

1 #include <iostream>
2 #include "Matrix.h"
3 using namespace std;
4 using namespace numeric;
5
6 int main() {
7     Matrix<double> matrix = inputMatrix<double>("input1Matrix.txt");
8     vector<double> b = inputVector<double>("input1Vector.txt");
9     Matrix<double> LinearSystem(matrix.rows(), matrix.rows() + 1);
10    Matrix<double> InverseSystem(matrix.rows(), matrix.rows() + matrix.rows());
11    for (size_t i = 0; i < matrix.rows(); ++i) {
12        for (size_t j = 0; j < matrix.cols(); ++j) {
13            LinearSystem[i][j] = matrix[i][j];
14            InverseSystem[i][j] = matrix[i][j];
15        }
16    }
17    for (size_t i = 0; i < b.size(); ++i) {
18        LinearSystem[i][LinearSystem.rows()] = b[i];
19    }
20    for (size_t i = 0; i < matrix.cols(); ++i) {
21        InverseSystem[i][i + matrix.rows()] = 1;
22    }
23    LUMatrix<double> LinearSystemLU(LinearSystem);
24    LUMatrix<double> LU(matrix);
25    LUMatrix<double> InverseSystemLU(InverseSystem);
26    cout << "\nSolving system:\n";
27    printMatrix(LinearSystemLU.solve());
28    cout << "\nDeterminant:\n";
29    cout << LinearSystemLU.determinant();
30    cout << "\nInverseMatrix:\n";

```

```

31 | printMatrix(InverseSystemLU.solve());
32 | cout << "\nU:\n";
33 | printMatrix(LU.U);
34 | cout << "\nL:\n";
35 | printMatrix(LU.L);
36 | cout << "\nMatrix:\n";
37 | printMatrix(LU.L * LU.U);
38 | return 0;
39 | }

```

1.2 Метод прогонки

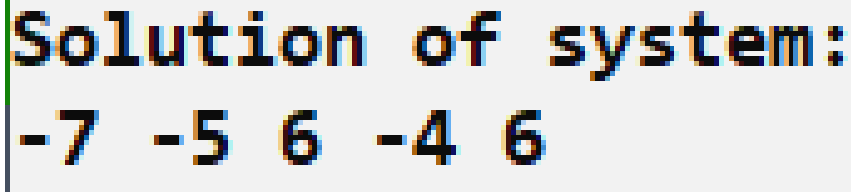
4 Постановка задачи

Реализовать метод прогонки в виде программы, задавая в качестве входных данных ненулевые элементы матрицы системы и вектор правых частей. Используя разработанное программное обеспечение, решить СЛАУ с трехдиагональной матрицей.

Вариант: 3

$$\begin{cases} 13x_1 - 5x_2 = -66 \\ -4x_1 + 9x_2 - 5x_3 = -47 \\ -x_2 - 12x_3 - 6x_4 = -43 \\ 6x_3 + 20x_4 - 5x_5 = -74 \\ 4x_4 + 5x_5 = 14 \end{cases}$$

5 Результаты работы



```
Solution of system:
-7 -5 6 -4 6
```

Рис. 2: Вывод программы в консоли

6 Исходный код

Коэффициенты перед искоми системы:

```
1 | 13 -5 0 0 0
2 | -4 9 -5 0 0
3 | 0 -1 -12 -6 0
4 | 0 0 6 20 -5
5 | 0 0 0 4 5
```

Вектор b:

```
1 | -66
2 | -47
3 | -43
4 | -74
5 | 14
```

```
1 | #include <iostream>
2 | #include "Matrix.h"
3 | using namespace std;
4 | using namespace numeric;
5 |
6 | int main() {
7 |     Matrix<double> matrix = inputMatrix<double>("input2Matrix.txt");
8 |     vector<double> b = inputVector<double>("input2Vector.txt");
9 |     cout << "\nMatrix:\n";
10 |    printMatrix(matrix);
11 |    cout << "\nVector b:\n";
12 |    printVector(b);
13 |    cout << "\nSolution of system:\n";
14 |    printVector(tridiagonalSolve(matrix, b));
15 |    return 0;
16 | }
```


1.3 Метод простых итераций. Метод Зейделя

7 Постановка задачи

Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности.

Вариант: 3

$$\begin{cases} -23x_1 - 7x_2 + 5x_3 + 2x_4 = -26 \\ -7x_1 - 21x_2 + 4x_3 + 9x_4 = -55 \\ 9x_1 + 5x_2 - 31x_3 - 8x_4 = -58 \\ x_2 - 2x_3 + 10x_4 = -24 \end{cases}$$

8 Результаты работы

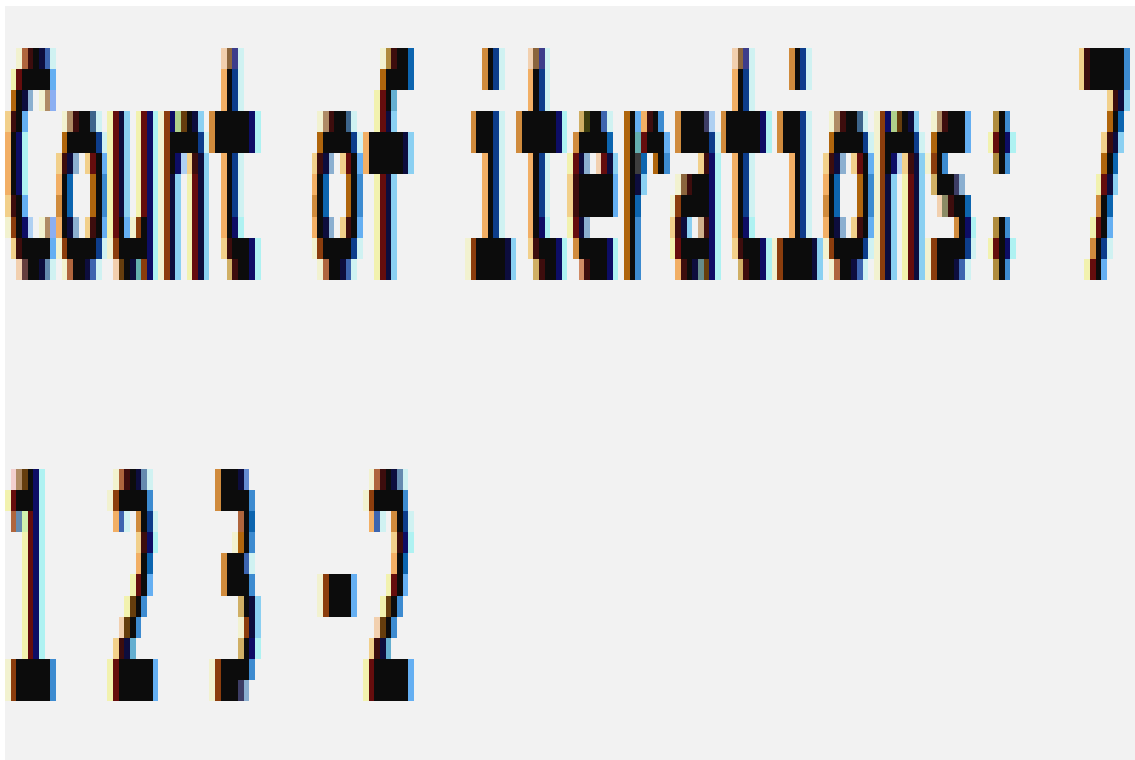


Рис. 3: Вывод программы в консоли

9 Исходный код

Коэффициенты перед иксами системы:

```
1 || -23 -7 5 2
2 || -7 -21 4 9
3 || 9 5 -31 -8
4 || 0 1 -2 10
```

Вектор b:

```
1 || -26
2 || -55
3 || -58
4 || -24
```

```
1 | #include <iostream>
2 | #include "Matrix.h"
3 | using namespace std;
4 | using namespace numeric;
5 |
6 | int main() {
7 |     Matrix<double> matrix = inputMatrix<double>("input3Matrix.txt");
8 |     vector<double> b = inputVector<double>("input3Vector.txt");
9 |     cout << "\nMatrix:\n";
10 |    printMatrix(matrix);
11 |    cout << "\nVector b:\n";
12 |    printVector(b);
13 |    cout << "\nSolution of system by iterations\n";
14 |    printVector(iterationSolve<double>(matrix, b, 0.001));
15 |    cout << "\nSolution of system by Seidel\n";
16 |    printVector(SeidelSolve<double>(matrix, b, 0.001));
17 |
18 |
19 |    return 0;
20 | }
```

1.4 Метод вращений

10 Постановка задачи

Реализовать метод вращений в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.

Вариант: 3

$$\begin{pmatrix} 5 & 5 & 3 \\ 5 & -4 & 1 \\ 3 & 1 & 2 \end{pmatrix}$$

11 Результаты работы

```
Eigen values  
-6.23937 8.70547 0.533906  
Eigen vectors  
0.415191 -0.830359 0.371646  
-0.908818 -0.360255 0.210393  
-0.0408142 -0.425112 -0.90422
```

Рис. 4: Вывод программы в консоли

12 Исходный код

Матрица:

```
1 | 5 5 3
2 | 5 -4 1
3 | 3 1 2
```

```
1 | #include <iostream>
2 | #include "Matrix.h"
3 | using namespace std;
4 | using namespace numeric;
5 |
6 | int main() {
7 |     Matrix<double> matrix = inputMatrix<double>("input4Matrix.txt");
8 |     cout << "\nMatrix:\n";
9 |     printMatrix(matrix);
10 |    cout << "\nEigen values\n";
11 |    auto res = findEigenvaluesAndEigenvectors(matrix, 0.001);
12 |    printVector(res.eigenValues);
13 |    cout << "\nEigen vectors\n";
14 |    for(const auto& x: res.eigenVectors) {
15 |        printVector(x);
16 |        cout << "\n";
17 |    }
18 |    return 0;
19 | }
```

1.5 QR – разложение матриц

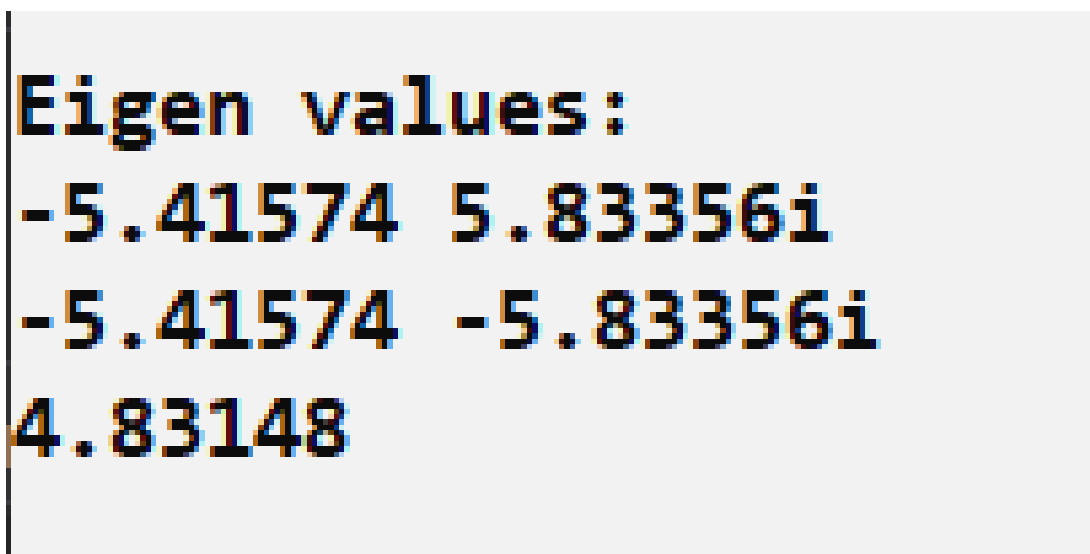
13 Постановка задачи

Реализовать алгоритм QR – разложения матриц в виде программы. На его основе разработать программу, реализующую QR – алгоритм решения полной проблемы собственных значений произвольных матриц, задавая в качестве входных данных матрицу и точность вычислений. С использованием разработанного программного обеспечения найти собственные значения матрицы.

Вариант: 3

$$\begin{pmatrix} 5 & -5 & -6 \\ -1 & -8 & -5 \\ 2 & 7 & -3 \end{pmatrix}$$

14 Результаты работы

A screenshot of a console window with a light gray background. The text is displayed in a monospaced font. The first line is "Eigen values:". The second line is "-5.41574 5.83356i". The third line is "-5.41574 -5.83356i". The fourth line is "4.83148".

```
Eigen values:  
-5.41574 5.83356i  
-5.41574 -5.83356i  
4.83148
```

Рис. 5: Вывод программы в консоли

15 Исходный код

Матрица:

```
1 5 -5 -6
2 -1 -8 -5
3 2 7 -3

1 #include <iostream>
2 #include "Matrix.h"
3 using namespace std;
4 using namespace numeric;
5
6 int main() {
7     Matrix<double> matrix = inputMatrix<double>("input5Matrix.txt");
8     cout << "\nMatrix:\n";
9     printMatrix(matrix);
10    QRMatrix<double> QRDecomposition(matrix);
11    cout << "\nEigen values:\n";
12    auto res = findEigenvaluesAndEigenvectorsByQR(matrix, 0.001, 0.0001);
13    for(auto c: res.eigenValues) {
14        if(c.imag() == 0) {
15            cout << c.real() << "\n";
16        } else {
17            cout << c.real() << " " << c.imag() << "i\n";
18        }
19    }
20    return 0;
21 }
```