

# Cuprins

<b>Fișiere comune</b> . . . . .	<b>2</b>
<b>Alocarea dinamica a memoriei. Tipuri specifice.</b> . . . . .	<b>16</b>
Laborator 1 . . . . .	16
Laborator 2 . . . . .	19
<b>Tablouri</b> . . . . .	<b>22</b>
Laborator 3 . . . . .	22
<b>Liste liniare simplu înlănțuite. Stive și cozi</b> . . . . .	<b>25</b>
Laborator 4 . . . . .	25
Laborator 5 . . . . .	26

## Fișiere comune

utils.h

```
1  #pragma once
2  #include <initializer_list>
3  #include <iostream>
4
5  constexpr size_t getSize(std::initializer_list<double> l) {
6      size_t n = 0;
7      auto it = l.begin();
8      auto end = l.end();
9      while (it++ != end) ++n;
10     return n;
11 }
12
13 constexpr void getSize(std::initializer_list<
14     std::initializer_list<double>> list,
15     int& s1, int& s2) {
16     constexpr int invalid = -1;
17     s1 = 0;
18     s2 = invalid;
19
20     for (const auto& l: list) {
21         int i = getSize(l);
22         ++s1;
23         if (s2 != invalid && i != s2) {
24             //std::cerr << i << "!=" << s2 << "\n";
25             throw std::logic_error("Fixed line size expected");
26         }
27         s2 = i;
28     }
29     //std::cerr << "size: " << s1 << ", " << s2 << "\n";
30 }
31 constexpr int MAX_SZ = 256;
32 inline size_t readSize(const char* name, int a = 1, int b = MAX_SZ) {
33     int res;
34     do {
35         std::cout << name << ": ";
36         std::cin >> res;
37     } while (res < a || res > b);
38     return res;
39 }
```

```

40
41
42 inline void assert(bool cond, const char* msg) {
43     if (!cond) throw std::logic_error(msg);
44 }
45 template<typename T>
46 inline T read(const char* name) {
47     T res;
48     std::cout << name << ": ";
49     std::cin >> res;
50     return res;
51 }

```

## matrix.h

```

1  #pragma once
2  #include "utils.h"
3
4  #include <iostream>
5  #include <cmath>
6  #include <cstring>
7  #include <utility>
8
9  struct Mat {
10     int m, n;
11     double *data;
12     Mat() : m(0), n(0), data(nullptr) {}
13     Mat(int m, int n) : m(m), n(n) { data = new double[m*n]; }
14     //template<int m, int n>
15     Mat(std::initializer_list<std::initializer_list<double>> list) {
16         getSize(list, m, n);
17         data = new double[m*n];
18
19         auto it = begin();
20         for (const auto& l : list) {
21             for (const auto& v : l) *(it++) = v;
22         }
23     }
24     Mat(Mat&& rhs) noexcept :
25         m(rhs.m), n(rhs.n),
26         data(std::exchange(rhs.data, nullptr)) {
27         //rhs.n = rhs.m = 0;
28     }
29
30     Mat(const Mat& rhs) : Mat(rhs.m, rhs.n) {
31         std::memcpy(data, rhs.data, sizeof(double) *m*n);
32     }
33     Mat& operator=(const Mat& rhs) {
34         setSize(rhs.m, rhs.n);
35         std::memcpy(data, rhs.data, sizeof(double) *m*n);

```

```

36         return *this;
37     }
38     Mat& operator=(Mat&& rhs) noexcept {
39         this->~Mat();
40         m = rhs.m;
41         n = rhs.n;
42         data = std::exchange(rhs.data, nullptr);
43
44         return *this;
45     }
46
47     ~Mat() {
48         delete[] data;
49     }
50     double& at(int i, int j) {
51         assert(i < m && j < n, "out of range");
52         return data[i * n + j];
53     }
54     double at(int i, int j) const { return data[i * n + j]; }
55     double* begin() { return data; }
56     const double* begin() const { return data; }
57     double* end() { return data + m*n; }
58     const double* end() const { return data + m*n; }
59
60     void setSize(int m, int n) {
61         if (this->m == m && this->n == n) return;
62         this->~Mat();
63         new (this) Mat(m, n);
64     }
65
66     static Mat read() {
67         Mat res(readSize("m"), readSize("n"));
68         for (auto& i : res)
69             std::cin >> i;
70         return res;
71     }
72     void print(const char* name) const {
73         std::cout << name << " = Mat " << m << "x" << n << "{\n";
74
75         for (int i = 0; i < m; ++i) {
76             for (int j = 0; j < n; ++j) {
77                 std::cout << at(i, j) << " ";
78             }
79             std::cout << "\n";
80         }
81         std::cout << "}\n";
82     }
83
84     friend std::ostream& operator<<(std::ostream& s, Mat& m) {
85         s << "Mat " << m.m << "x" << m.n << "{\n";

```

```

86     for (int i = 0; i < m.m; ++i) {
87         for (int j = 0; j < m.n; ++j) {
88             s << m.at(i, j) << " ";
89         }
90         s << "\n";
91     }
92     return s << "}\n";
93 }
94
95 double norm1() const { return normImpl<Type::Row>(n, m); }
96 double normInf() const { return normImpl<Type::Col>(m, n); }
97 double normF() const {
98     double res = 0;
99     for (int i = 0; i < m; ++i)
100         for (int j = 0; j < n; ++j)
101             res += at(i, j) * at(i, j);
102
103     return std::sqrt(res);
104 }
105 bool isStrictlyRowDiagonallyDominant() const {
106     return isStrictlyDiagonallyDominantImpl<Type::Row>();
107 }
108
109 bool isStrictlyColDiagonallyDominant() const {
110     return isStrictlyDiagonallyDominantImpl<Type::Col>();
111 }
112
113 private:
114
115     enum class Type { Row, Col };
116     template<Type type>
117     bool isStrictlyDiagonallyDominantImpl() const {
118         assert(m == n, "Matrix must be square");
119         for (int i = 0; i < m; ++i) {
120             double val = std::abs(at(i, i));
121             double sum = -val;
122             for (int j = 0; j < m; ++j) {
123                 sum += std::abs(type == Type::Col? at(j, i): at(i, j));
124             }
125             if (sum >= val) {
126                 std::cout << "(" << sum << "," << val << ")";
127                 return false;
128             }
129         }
130         return true;
131     }
132     template<Type type>
133     double normImpl(int sz1, int sz2) const {
134         double max = -1;
135         for (int j = 0; j < sz1; ++j) {

```

```

136         double x = 0;
137         for (int i = 0; i < sz2; ++i) {
138             x += std::abs(type == Type::Col? at(j, i) : at(i, j));
139         }
140         if (x > max) max = x;
141     }
142     return max;
143 }
144 };
145
146 Mat& add(const Mat& a, const Mat& b, Mat& res) {
147     assert(a.m == b.m && a.n == b.n, "Sizes don't match, can't add");
148     res.setSize(a.m, a.n);
149     for (int i = 0; i < a.m; ++i)
150         for (int j = 0; j < a.n; ++j)
151             res.at(i, j) = a.at(i, j) + b.at(i, j);
152     return res;
153 }
154
155 Mat& mul(double a, const Mat& b, Mat& res) {
156     res.setSize(b.m, b.n);
157
158     for (int i = 0; i < res.m; ++i)
159         for (int j = 0; j < res.n; ++j)
160             res.at(i, j) = a * b.at(i, j);
161     return res;
162 }
163
164 Mat& neg(const Mat& a, Mat& res) {
165     return mul(-1, a, res);
166 }
167
168 Mat& sub(const Mat& a, const Mat& b, Mat& res) {
169     return add(a, neg(b, res), res);
170 }
171
172 Mat& mul(const Mat& a, const Mat& b, Mat& res) {
173     assert(a.n == b.m, "Sizes don't match, can't multiply");
174     res.setSize(a.m, b.n);
175
176     for (int i = 0; i < res.m; ++i)
177         for (int j = 0; j < res.n; ++j) {
178             res.at(i, j) = 0;
179             for (int k = 0; k < a.n; ++k)
180                 res.at(i, j) += a.at(i, k) * b.at(k, j);
181         }
182     return res;
183 }
184
185 Mat& trans(const Mat& a, Mat& res) {
186     assert(a.data != res.data, "Can't transpose inplace");
187     res.setSize(a.n, a.m);

```

```

186
187     for (int i = 0; i < res.m; ++i)
188         for (int j = 0; j < res.n; ++j)
189             res.at(i, j) = a.at(j, i);
190     return res;
191 }

```

## vector.h

```

1  #pragma once
2  #include "utils.h"
3
4  #include <iostream>
5  #include <utility>
6  #include <cmath>
7
8  struct Vec {
9      double *_begin, *_end;
10
11      constexpr double* begin() { return _begin; }
12      constexpr const double* begin() const { return _begin; }
13
14      constexpr double* end() { return _end; }
15      constexpr const double* end() const { return _end; }
16
17      constexpr Vec() : _begin(nullptr), _end(nullptr) {}
18      explicit Vec(size_t n) : _begin(new double[n]), _end(_begin+n) {}
19      Vec(std::initializer_list<double> list) : Vec(getSize(list)) {
20          auto it = _begin;
21          for (const auto& v : list) *(it++) = v;
22      }
23      Vec(const Vec& rhs) : Vec(rhs.size()) {
24          auto it = _begin;
25          for (const auto& v : rhs) *(it++) = v;
26      }
27      Vec(Vec&& rhs) noexcept
28          : _begin(std::exchange(rhs._begin, nullptr)),
29            _end(std::exchange(rhs._end, nullptr)) {}
30
31      Vec& operator=(const Vec& rhs) {
32          if (size() != rhs.size()) {
33              this->~Vec();
34              new (this) Vec(rhs.size());
35          }
36          auto it = _begin;
37          for (const auto& v : rhs) *(it++) = v;
38          return *this;
39      }
40      Vec& operator=(Vec&& rhs) noexcept {
41          this->~Vec();

```

```

42     _begin = std::exchange(rhs._begin, nullptr);
43     _end = std::exchange(rhs._end, nullptr);
44     return *this;
45 }
46
47 ~Vec() { delete[] _begin; }
48
49 constexpr size_t size() const { return _end - _begin; }
50
51 constexpr double& operator[](size_t i) { return _begin[i]; }
52 constexpr double operator[](size_t i) const { return _begin[i]; }
53
54 void setSize(size_t n) {
55     if (size() == n) return;
56     *this = Vec(n);
57 }
58
59 friend std::ostream& operator<<(std::ostream& s, const Vec& v) {
60     s << "(";
61     double* it = v._begin;
62     for (double* end = v._end - 1; it < end; ++it)
63         s << *it << ", ";
64
65     if (it < v._end) s << *it;
66
67     return s << ")";
68 }
69 static Vec read() {
70     Vec res(readSize("n"));
71     for (auto& v: res) std::cin >> v;
72     return res;
73 }
74 double norm() const;
75 };
76 void assertSizes(const Vec& a, const Vec& b) {
77     assert(a.size() == b.size(), "Sizes don't match");
78 }
79
80 Vec& add(const Vec& a, const Vec& b, Vec& res) {
81     assertSizes(a, b);
82     res.setSize(a.size());
83     auto aIt = a.begin();
84     auto bIt = b.begin();
85     for (auto& v: res) v = *(aIt++) + *(bIt++);
86     return res;
87 }
88
89 Vec& mul(double a, const Vec& b, Vec& res) {
90     res.setSize(b.size());
91     auto it = b.begin();

```



```

92     for (auto& v : res) v = a * (*(it++));
93     return res;
94 }
95
96 Vec& neg(const Vec& b, Vec& res) { return mul(-1, b, res); }
97
98 Vec& sub(const Vec& a, const Vec& b, Vec& res) {
99     return add(a, neg(b, res), res);
100 }
101 double dot(const Vec& a, const Vec& b) {
102     double res = 0;
103     assertSizes(a, b);
104     auto bIt = b.begin();
105     for (auto& v : a) res += v * (*(bIt++));
106     return res;
107 }
108 double norm(const Vec& a) {
109     return std::sqrt(dot(a, a));
110 }
111 double Vec::norm() const {
112     return ::norm(*this);
113 }

```

## list.h

```

1  #include "utils.h"
2
3  #include <type_traits>
4  #include <utility>
5
6
7  // We should never check for (bot == nullptr),
8  // so we don't update it when the list becomes empty.
9  template<typename T>
10 struct List {
11     struct Node {
12         T val;
13         Node* next;
14     };
15     Node* top;
16     Node* bot;
17     constexpr List(Node* top = nullptr, Node* bot = nullptr)
18         : top(top), bot(bot) {}
19     List(const List&) = delete;
20     List& operator=(const List&) = delete;
21     ~List() {
22         while (top) {
23             Node* n = top;
24             top = top->next;
25             delete n;

```

```

26     }
27 }
28 List(List&& rhs) noexcept
29     : top(std::exchange(rhs.top, nullptr)),
30     bot(std::exchange(rhs.bot, nullptr)) {}
31 List& operator=(List&& rhs) noexcept {
32     this->~List();
33     this->top = std::exchange(rhs.top, nullptr);
34     this->bot = std::exchange(rhs.bot, nullptr);
35 }
36
37 T& first() { return top->val; }
38 T& last() { return bot->val; }
39 T pop_front() {
40     assert(top, "Empty list");
41     Node* n = top;
42     top = top->next;
43     T res = n->val;
44     delete n;
45     return res;
46 }
47
48 void push_front(T val) {
49     Node *n = new Node{ val, top };
50     top = n;
51 }
52 void push_back(T val) {
53     Node *n = new Node{ val, nullptr };
54     if (top == nullptr) {
55         top = n;
56     }
57     else {
58         bot->next = n;
59     }
60     bot = n;
61 }
62
63 bool empty() const { return top == nullptr; }
64
65 bool operator==(const List& rhs) {
66     for (auto it = top, rit = rhs.top;
67          it != nullptr;
68          it = it->next, rit = rit->next) {
69         if (it->val != rit->val) return false;
70     }
71     return true;
72 }
73
74 static List read(const char* msg) {
75     List q;

```

```

76         std::cout << msg << ":\n";
77         int len1 = readSize("n");
78         for (int i = 0; i < len1; ++i) {
79             T s;
80             std::cin >> s;
81             q.push_front(s);
82         }
83         return std::move(q);
84     }
85
86     friend std::ostream& operator<<(std::ostream& s, const List& st) {
87         if constexpr (std::is_same_v<T, char>) {
88             const Node* it = st.top;
89             while (it != nullptr) {
90                 s << it->val;
91                 it = it->next;
92             }
93             return s;
94         }
95         else {
96             if (st.empty()) return s << "{}";
97             s << "{";
98             const Node* it = st.top;
99             while (it->next != nullptr) {
100                 s << it->val << ", ";
101                 it = it->next;
102             }
103
104             return s << it->val << "}";
105         }
106     }
107
108     void remove(T& val) {
109         remove_if([&] (T& t) { return t == val; });
110     }
111     // removes all elements that satisfy p
112     template<class P>
113     void remove_if(P p) {
114         apply_on(p, [] (Node* n) { delete n; } );
115     }
116     //applies f() on all nodes that satisfy the predicate p()
117     template<class P, class F>
118     void apply_on(P p, F f) {
119         for (;;) {
120             auto* n = top;
121             if (!n) {
122                 return;
123             }
124             if (!p(n->val)) break;
125             top = n->next;

```

```

126         f(n);
127     }
128     auto* prev = top;
129     auto* it = prev->next;
130
131     while (it) {
132         if (p(it->val)) {
133             prev->next = it->next;
134             if (prev->next == nullptr) {
135                 bot = prev;
136             }
137             f(it);
138             it = prev;
139         } else {
140             prev = it;
141             it = it->next;
142         }
143     }
144 }
145 };
146
147 // keeps in l all the elements that don't satisfy the predicate p
148 // and returns pair of:
149 // - a reference to the original list
150 // - a list containing the elements that satisfy p
151 template<typename T, typename P>
152 constexpr auto partition_split(List<T>& l, P p) {
153     struct res_t {
154         List<T>& notSatisfying;
155         List<T> satisfying;
156     } res = { l, {} };
157     List<T>& sat = res.satisfying;
158     auto insertNode = [&sat](auto* n) {
159         n->next = sat.top;
160         sat.top = n;
161         if (sat.bot == nullptr) {
162             sat.bot = n;
163         }
164     };
165     l.apply_on(p, insertNode);
166     return res;
167 }

```

## stack.h

```

1  #include "utils.h"
2
3  template<typename T>
4  struct Stack {
5      struct Node {

```

```

6         T val;
7         Node* next;
8     };
9
10    Stack(const Stack&) = delete;
11    Stack& operator=(const Stack&) = delete;
12    Node* top;
13    Stack(Node* top = nullptr) : top(top) {}
14    ~Stack() {
15        while (top) {
16            Node* n = top;
17            top = top->next;
18            delete n;
19        }
20    }
21    void push(T val) {
22        Node *n = new Node{ val, top };
23        top = n;
24    }
25    void quickPop() {
26        Node* n = top;
27        top = top->next;
28        delete n;
29    }
30    T pop() {
31        assert(top, "Empty stack");
32        Node* n = top;
33        top = top->next;
34        T res = n->val;
35        delete n;
36        return res;
37    }
38    bool empty() const { return top == nullptr; }
39    friend std::ostream& operator<<(std::ostream& s, const Stack& st) {
40        if (st.empty()) return s << "{}";
41        s << "{";
42        const Node* it = st.top;
43        while (it->next != nullptr) {
44            s << it->val << ", ";
45            it = it->next;
46        }
47
48        return s << it->val << "}";
49    }
50 };

```

## queue.h

```

1  #include "utils.h"
2

```

```

3  template<class T>
4  struct Queue {
5      struct Node {
6          T val;
7          Node* next;
8      };
9      Node* top;
10     Node* bot;
11     Queue(Node* top = nullptr, Node* bot = nullptr) : top(top), bot(bot) {}
12     /*
13     Queue(const Queue&) = delete;
14     Queue& operator=(const Queue&) = delete;
15     ~Queue() {
16         while (top) {
17             Node* n = top;
18             top = top->next;
19             delete n;
20         }
21     }*/
22     void push(T val) {
23         Node *n = new Node{ val, nullptr };
24         if (bot == nullptr) {
25             top = n;
26         }
27         else {
28             bot->next = n;
29         }
30         bot = n;
31     }
32
33     T pop() {
34         assert(top, "Empty queue");
35         Node* old = top;
36         T res = old->val;
37         top = top->next;
38         if (top == nullptr)
39             bot = nullptr;
40         delete old;
41         return res;
42     }
43     bool empty() const { return top == nullptr; }
44     static Queue read(const char* msg) {
45         Queue q;
46         std::cout << msg << ":\n";
47         int len1 = readSize("n");
48         for (int i = 0; i < len1; ++i) {
49             T s;
50             std::cin >> s;
51             q.push(s);
52         }

```

```

53     return std::move(q);
54 }
55
56 friend std::ostream& operator<<(std::ostream& s, const Queue& st) {
57     if (st.empty()) return s << "{}";
58     s << "{";
59     const Node* it = st.top;
60     while (it->next != nullptr) {
61         s << it->val << ", ";
62         it = it->next;
63     }
64
65     return s << it->val << "}";
66 }
67 };

```

# Alocarea dinamică a memoriei.

## Tipuri specifice.

### Laborator 1

16. Scrieți funcții pentru implemetarea operațiilor specifice pe matrice de numere reale cu  $m$  linii și  $n$  coloane: suma, diferența și produsul al două matrice, produsul dintre o matrice și un scalar real, transpusa unei matrice, norme matriceale specifice<sup>1</sup>, citirea de la tastatură a componentelor unei matrice, afișarea componentelor matricei. Pentru cazul particular al unei matrice patratice de ordin  $n$ , să se testeze dacă aceasta satisface criteriul de dominanță pe linii<sup>2</sup> sau pe coloane<sup>3</sup>. Se vor folosi tablouri bidimensionale alocate static.

```
1  #include "utils.h"
2
3  #include <iostream>
4  #include <cmath>
5
6  struct Mat {
7      double data[MAX_SZ][MAX_SZ] {};
8      int m, n;
9
10     Mat() : m(0), n(0) {}
11     Mat(int m, int n) : m(m), n(n) {}
12
13     static Mat read() {
14         Mat res(readSize("m"), readSize("n"));
15
16         for (int i = 0; i < res.m; ++i)
17             for (int j = 0; j < res.n; ++j)
18                 std::cin >> res.data[i][j];
19         return res;
20     }
21     void setSize(int m, int n) {
22         this->m = m;
23         this->n = n;
24     }
25 }
```

<sup>1</sup>Dacă  $A \in \mathcal{M}_{m \times n}(\mathbb{R})$ , atunci  $\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |a_{ij}|$ ,  $\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^n |a_{ij}|$ ,  $\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n a_{ij}^2}$ .

<sup>2</sup> $A \in \mathcal{M}_n(\mathbb{R})$  este strict diagonal dominantă pe linii dacă  $|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}|$ , pentru orice  $i = 1, \dots, n$ .

<sup>3</sup> $A \in \mathcal{M}_n(\mathbb{R})$  este strict diagonal dominantă pe colonane dacă  $|a_{jj}| > \sum_{\substack{i=1 \\ i \neq j}}^n |a_{ij}|$ , pentru orice  $j = 1, \dots, n$ .



```

25     double& at(int i, int j) { return data[i][j]; }
26     double at(int i, int j) const { return data[i][j]; }
27
28     void print(const char* name) const {
29         std::cout << name << " = Mat " << m << "x" << n << "{\n";
30         for (int i = 0; i < m; ++i) {
31             for (int j = 0; j < n; ++j)
32                 std::cout << at(i, j) << " ";
33             std::cout << "\n";
34         }
35         std::cout << "}\n";
36     }
37 private:
38     enum class Type { Row, Col };
39     template<Type type>
40     bool isStrictlyDiagonallyDominantImpl() const {
41         assert(m == n, "Matrix must be square");
42         for (int i = 0; i < m; ++i) {
43             double val = std::abs(at(i, i));
44             double sum = -val;
45             for (int j = 0; j < m; ++j)
46                 sum += std::abs(type == Type::Col ? at(j, i) : at(i, j));
47             if (sum >= val) return false;
48         }
49         return true;
50     }
51
52     template<Type type>
53     double normImpl(int sz1, int sz2) const {
54         double max = -1;
55         for (int j = 0; j < sz1; ++j) {
56             double x = 0;
57             for (int i = 0; i < sz2; ++i)
58                 x += std::abs(type == Type::Col ? at(j, i) : at(i, j));
59             if (x > max) max = x;
60         }
61         return max;
62     }
63 public:
64     double norm1() const { return normImpl<Type::Row>(n, m); }
65     double normInf() const { return normImpl<Type::Col>(m, n); }
66     double normF() const {
67         double res = 0;
68         for (int i = 0; i < m; ++i)
69             for (int j = 0; j < n; ++j)
70                 res += at(i, j) * at(i, j);
71
72         return std::sqrt(res);
73     }
74     bool isStrictlyRowDiagonallyDominant() const {

```

```

75         return isStrictlyDiagonallyDominantImpl<Type::Row>();
76     }
77     bool isStrictlyColDiagonallyDominant() const {
78         return isStrictlyDiagonallyDominantImpl<Type::Col>();
79     }
80 };
81
82 Mat& add(const Mat& a, const Mat& b, Mat& res) {
83     assert(a.m == b.m && a.n == b.n, "Sizes don't match, can't add");
84     res.setSize(a.m, a.n);
85     for (int i = 0; i < a.m; ++i)
86         for (int j = 0; j < a.n; ++j)
87             res.at(i, j) = a.at(i, j) + b.at(i, j);
88     return res;
89 }
90 Mat& mul(double a, const Mat& b, Mat& res) {
91     res.setSize(b.m, b.n);
92
93     for (int i = 0; i < res.m; ++i)
94         for (int j = 0; j < res.n; ++j)
95             res.at(i, j) = a * b.at(i, j);
96     return res;
97 }
98 Mat& neg(const Mat& a, Mat& res) { return mul(-1, a, res); }
99 Mat& sub(const Mat& a, const Mat& b, Mat& res) {
100     return add(a, neg(b, res), res);
101 }
102 Mat& mul(const Mat& a, const Mat& b, Mat& res) {
103     assert(a.n == b.m, "Sizes don't match, can't multiply");
104     res.setSize(a.m, b.n);
105
106     for (int i = 0; i < res.m; ++i)
107         for (int j = 0; j < res.n; ++j) {
108             res.at(i, j) = 0;
109             for (int k = 0; k < a.n; ++k)
110                 res.at(i, j) += a.at(i, k) * b.at(k, j);
111         }
112     return res;
113 }
114 Mat& trans(const Mat& a, Mat& res) {
115     assert(a.data != res.data, "Can't calculate the transpose inplace");
116     res.setSize(a.n, a.m);
117
118     for (int i = 0; i < res.m; ++i)
119         for (int j = 0; j < res.n; ++j)
120             res.at(i, j) = a.at(j, i);
121     return res;
122 }

```

## Laborator 2

18. Scrieți funcții pentru implementarea operațiilor specifice pe vectori din  $\mathbb{R}^n$ : suma, diferența și produsul scalar al doi vectori, produsul dintre un vector și un scalar real, negativarea unui vector, norma euclidiană a unui vector, citirea de la tastatură a celor  $n$  componente ale unui vector, afișarea componentelor vectorului sub forma unui  $n$ -uplu de elemente. Se vor folosi tablouri unidimensionale alocate dinamic.

```
1  #include "utils.h"
2
3  #include <iostream>
4  #include <utility>
5  #include <cmath>
6
7  struct Vec {
8      double *_begin, *_end;
9
10     constexpr double* begin() { return _begin; }
11     constexpr const double* begin() const { return _begin; }
12
13
14     constexpr double* end() { return _end; }
15     constexpr const double* end() const { return _end; }
16
17     constexpr Vec() : _begin(nullptr), _end(nullptr) {}
18     explicit Vec(size_t n) : _begin(new double[n]), _end(_begin+n) {}
19     Vec(std::initializer_list<double> list) : Vec(getSize(list)) {
20         auto it = _begin;
21         for (const auto& v : list) *(it++) = v;
22     }
23     Vec(const Vec&) = delete;
24     Vec(Vec&& rhs) noexcept
25         : _begin(std::exchange(rhs._begin, nullptr)),
26         _end(std::exchange(rhs._end, nullptr)) {}
27
28     Vec& operator=(const Vec&) = delete;
29     Vec& operator=(Vec&& rhs) noexcept {
30         this->~Vec();
31         _begin = std::exchange(rhs._begin, nullptr);
32         _end = std::exchange(rhs._end, nullptr);
33         return *this;
34     }
35
36     ~Vec() { delete[] _begin; }
37
38     constexpr size_t size() const { return _end - _begin; }
39
40     constexpr double& operator[](size_t i) { return _begin[i]; }
41     constexpr double operator[](size_t i) const { return _begin[i]; }
42
43     void setSize(size_t n) {
```

```

44         if (size() == n) return;
45         *this = Vec(n);
46     }
47
48     friend std::ostream& operator<<(std::ostream& s, const Vec& v) {
49         s << "(";
50         double* it = v._begin;
51         for (double* end = v._end - 1; it < end; ++it)
52             s << *it << ", ";
53
54         if (it < v._end) s << *it;
55
56         return s << ")";
57     }
58     static Vec read() {
59         Vec res(readSize("n"));
60         for (auto& v: res) std::cin >> v;
61         return res;
62     }
63     double norm() const;
64 };
65 void assertSizes(const Vec& a, const Vec& b) {
66     assert(a.size() == b.size(), "Sizes don't match");
67 }
68
69 Vec& add(const Vec& a, const Vec& b, Vec& res) {
70     assertSizes(a, b);
71     res.setSize(a.size());
72     auto aIt = a.begin();
73     auto bIt = b.begin();
74     for (auto& v : res) v = *(aIt++) + *(bIt++);
75     return res;
76 }
77
78 Vec& mul(double a, const Vec& b, Vec& res) {
79     res.setSize(b.size());
80     auto it = b.begin();
81     for (auto& v : res) v = a * (*(it++));
82     return res;
83 }
84
85 Vec& neg(const Vec& b, Vec& res) { return mul(-1, b, res); }
86
87 Vec& sub(const Vec& a, const Vec& b, Vec& res) {
88     return add(a, neg(b, res), res);
89 }
90 double dot(const Vec& a, const Vec& b) {
91     double res = 0;
92     assertSizes(a, b);
93     auto bIt = b.begin();

```

```
94     for (auto& v : a) res += v * (*(bIt++));
95     return res;
96 }
97 double norm(const Vec& a) {
98     return std::sqrt(dot(a, a));
99 }
100 double Vec::norm() const {
101     return ::norm(*this);
102 }
```

# Tablouri

## Laborator 3

7. Folosind structurile de date `VECTOR` și `MATRICE` definite la curs și funcțiile necesare, rezolvați următorul sistem algebric liniar cu  $n$  ecuații și  $n$  necunoscute folosind metoda lui Gauß de eliminare.

$$\left\{ \begin{array}{l} 2x_1 - x_2 = 1 \\ -x_1 + 2x_2 - x_3 = 1 \\ -x_2 + 2x_3 - x_4 = 1 \\ \dots\dots\dots \\ -x_{n-2} + 2x_{n-1} - x_n = 1 \\ -x_{n-1} + 2x_n = 1, \quad n \in \mathbb{N}, 2 \leq n \leq 50 \end{array} \right. .$$

```
1  #include "utils.h"
2
3  #include "vector.h"
4  #include "matrix.h"
5
6  #include <iostream>
7  #include <cmath>
8
9  constexpr double eps = 1e-7;
10
11 Vec& mul(const Mat& m, const Vec& v, Vec& res) {
12     assert(v.begin() != res.begin(), "Can't multiply inplace");
13     assert(v.size() == size_t(m.n), "Sizes don't match");
14     res.setSize(m.m);
15
16
17     for (size_t i = 0; i < res.size(); ++i) {
18         res[i] = 0;
19         for (size_t k = 0; k < v.size(); ++k)
20             res[i] += m.at(i, k) * v[k];
21     }
22     return res;
23 }
24
25 // A * X = b
26 struct System {
27     Mat A;
```

```

28 Vec b;
29
30 System(int n, int m) : A(n, m), b(n) {}
31
32 System(std::initializer_list<std::initializer_list<double>> A,
33         std::initializer_list<double> b) : A(A), b(b) {
34     assert(std::size_t(this->A.m) == this->b.size(),
35            "sizes don't match");
36 }
37
38 friend std::ostream& operator<<(std::ostream& s, const System& sys) {
39     s << "System " << sys.A.m << "x" << sys.A.n << ": \n";
40     auto& A = sys.A;
41     for (int i = 0; i < A.m; ++i) {
42         s << "{";
43         for (int j = 0; j < A.n; ++j) {
44             //showpos shows a '+' in front of positive numbers
45             if (std::abs(A.at(i, j)) > eps)
46                 s << std::showpos << A.at(i, j)
47                 << std::noshowpos << "*x" << (j+1) << " ";
48         }
49         s<< "= " << sys.b[i] << "\n";
50     }
51
52     return s;
53 }
54
55 // L_i += f * L_j
56 void addLines(int i, double f, int j) {
57     for (int k = 0; k < A.n; ++k) {
58         A.at(i, k) += f * A.at(j, k);
59     }
60     b[i] += f * b[j];
61 }
62
63 // L_i *= f
64 void multiplyLine(int i, double f) {
65     for (int k = 0; k < A.n; ++k) {
66         A.at(i, k) *= f;
67     }
68     b[i] *= f;
69 }
70
71 Vec solveTriangulated() {
72     for (int i = A.m-1; i > 0; --i) {
73         addLines(i-1, -A.at(i-1, i), i);
74         multiplyLine(i, 1 / A.at(i, i));
75     }
76     return b;
77 }

```

```

78
79 bool checkSolution(const Vec& x) const {
80     Vec r;
81     mul(A, x, r);
82     sub(b, r, r);
83     for (auto& v : r) {
84         if (std::abs(v) > eps) return false;
85     }
86     return true;
87 }
88
89 static Vec solveCustom(int n) {
90     System s(n, n);
91     for (auto& v : s.A) v = 0;
92     s.A.at(0,0) = 2;
93     s.A.at(0,1) = -1;
94     s.b[0] = 1;
95     for (int i = 1; i < n - 1; ++i) {
96         s.b[i] = 1;
97
98         s.A.at(i,i-1) = -1;
99         s.A.at(i,i) = 2;
100        s.A.at(i,i+1) = -1;
101    }
102    s.b[n-1] = 1;
103    s.A.at(n-1, n-2) = -1;
104    s.A.at(n-1, n-1) = 2;
105    s.customTriangulate();
106    return s.solveTriangulated();
107 }
108 void customTriangulate() {
109     multiplyLine(0, 1 / A.at(0, 0));
110     for (int i = 1; i < A.m; ++i) {
111         addLines(i, 1, i-1);
112         multiplyLine(i, 1 / A.at(i, i));
113     }
114 }
115 };
116
117 int main() {
118     try {
119         int n = readSize("n", 2, 51);
120         std::cout << "x = " << System::solveCustom(n) << "\n";
121     } catch (std::exception& e) {
122         std::cerr << "Error" << e.what() << "\n";
123         return 1;
124     }
125     return 0;
126 }

```



# Liste liniare simplu înlănțuite

## Stive și cozi

### Laborator 4

5. Se citește un text de la tastatura (poate conține orice caracter, inclusiv spații) și se încarcă în două stive: o stivă va conține doar litere mici, iar cealaltă doar litere mari. Se citește de la tastatură o vocală a alfabetului englez (literă mare sau mică). Ștergeți stiva corespunzătoare până la întâlnirea vocalei citite.

```
1  #include "utils.h"
2  #include "stack.h"
3
4  #include <iostream>
5  #include <cstdlib>
6
7  bool isVowel(char c) {
8      c = tolower(c);
9      return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
10 }
11
12 int main() {
13     std::string str;
14     std::cout << "str: ";
15     std::getline(std::cin, str);
16
17     char v;
18     do {
19         v = read<char>("vowel");
20     } while (!isVowel(v));
21
22     Stack<char> lower;
23     Stack<char> upper;
24     auto printStacks = [&] (const char* s) {
25         std::cout << s;
26         std::cout << "lower:" << lower << "\n";
27         std::cout << "upper:" << upper << "\n";
28     };
29
30     printStacks("Before:\n");
31     for (auto& c : str) {
32         if (islower(c)) lower.push(c);
33         else if (isupper(c)) upper.push(c);
```

```

34     }
35
36     printStacks("After Adding:\n");
37     Stack<char>& stack = isupper(v)? upper: lower;
38     while (!stack.empty()) {
39         if (char c = stack.pop(); c == v) {
40             break;
41         }
42     }
43     printStacks("Result:\n");
44     return 0;
45 }

```

## Laborator 5

11. Creați o listă liniară simplu înlănțuită în nodurile căreia sunt memorate numere naturale. Separați numerele naturale memorate în listă, în două liste, una corespunzătoare numerelor pare și cealaltă, numerelor impare. Afișați cele două liste. Ștergeți din lista numerelor pare, o valoare pară  $x$ , citită de la tastatură, ori de câte ori apare în listă.

```

1  #include "utils.h"
2  #include "list.h"
3
4  #include <iostream>
5
6  int main() {
7      auto nums = List<int>::read("numbers");
8
9      auto [evens, odds] = partition_split(nums, [](int n) {return n % 2;});
10     std::cout << "odd: " << odds << "\n";
11     std::cout << "even: " << evens << "\n";
12
13     int x;
14     do {
15         std::cout << "x (must be even): ";
16         std::cin >> x;
17     } while (x % 2);
18
19     std::cout << "removing...\n";
20     evens.remove(x);
21
22     std::cout << "even: " << evens << "\n";
23     return 0;
24 }

```

17. Modelați printr-o LLSI un stoc de produse caracterizate prin: denumire, unitate de măsură, cantitate și preț unitar. Implementați principalele operații pe stoc: crearea stocului, introducerea unui produs nou, eliminarea unui produs în cazul în care acesta a fost vândut în întregime, modificare informații despre un produs (de exemplu, modificarea cantității unui produs, în cazul vânzării), calculul valorii stocului la un moment dat, listare stoc.

```
1 #include <iostream>
```