НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. Ігоря СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Протокол
до комп'ютерного практикуму №4

ОБЧИСЛЕННЯ ВЛАСНИХ ЗНАЧЕНЬ

Виконав студент
групи ФІ-81

Шкаліков О.В.

Перевірила:
Стьопочкіна І.В.

Київ — 2021

Практична частина

Розглянемо застосування ітераційного метода Якобі на прикладі наступної матриці:

$$A = \begin{pmatrix} 7 & 0.88 & 0.93 & 1.23 \\ 0.88 & 4.16 & 1.3 & 0.15 \\ 0.93 & 1.3 & 6.44 & 2 \\ 1.21 & 0.15 & 2 & 9 \end{pmatrix}$$

Наведемо таблицю зі значенням матиці повороту та сферичної норми(S) матриці A на кожній ітерації алгоритму. Зазначимо, що $T^{-1} = T^T$.

k	Матриця T_k	S_d	S_{nd}	S
1	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.877227 & 0.480076 \\ 0 & 0 & -0.480076 & 0.877227 \end{pmatrix}$	196.779	9.6318	206.411
2	$\begin{pmatrix} 0.926325 & 0 & 0 & 0.376726 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -0.376726 & 0 & 0 & 0.926325 \end{pmatrix}$	201.327	5.08418	206.411
3	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.861724 & 0.507378 & 0 \\ 0 & -0.507378 & 0.861724 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	203.61	2.8013	206.411
4	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.993337 & 0 & 0.115243 \\ 0 & 0 & 1 & 0 \\ 0 & -0.115243 & 0 & 0.993337 \end{pmatrix}$	205.034	1.37667	206.411
5	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.992703 & 0.120586 \\ 0 & 0 & -0.120586 & 0.992703 \end{pmatrix}$	205.744	0.667086	206.411
6	$\begin{pmatrix} 0.512215 & 0 & 0.858857 & 0 \\ 0 & 1 & 0 & 0 \\ -0.858857 & 0 & 0.512215 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	206.146	0.264725	206.411
7	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.996784 & 0.0801361 & 0 \\ 0 & -0.0801361 & 0.996784 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	206.282	0.128844	206.411
8	$\begin{pmatrix} 0.103824 & 0.994596 & 0 & 0 \\ -0.994596 & 0.103824 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	206.392	0.0187992	206.411
9	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.999817 & 0.0191274 \\ 0 & 0 & -0.0191274 & 0.999817 \end{pmatrix}$	206.405	0.0058601	206.411

10	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.999959 & 0 & 0.00901594 \\ 0 & 0 & 1 & 0 \\ 0 & -0.00901594 & 0 & 0.999959 \end{pmatrix}$	206.41	0.00142838	206.411
11	$\begin{pmatrix} 0.999997 & 0 & 0 & 0.00264493 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -0.00264493 & 0 & 0 & 0.999997 \end{pmatrix}$	206.41	0.00064294	206.411
12	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.999845 & 0.0175874 & 0 \\ 0 & -0.0175874 & 0.999845 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	206.411	5.10303e-06	206.411
13	$\begin{pmatrix} 1 & 0 & 0.000479337 & 0 \\ 0 & 1 & 0 & 0 \\ -0.000479337 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	206.411	1.39792e-07	206.411
14	$\begin{pmatrix} 1 & 9.08961e-05 & 0 & 0 \\ -9.08961e-05 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	206.411	5.45773e-08	206.411
15	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 3.92664e-05 \\ 0 & 0 & -3.92664e-05 & 1 \end{pmatrix}$	206.411	2.28514e-11	206.411

Таким чином отримали наступні значення власних чисел:

$$\lambda_1 = 3.38753 \qquad \lambda_2 = 5.65842$$
$$\lambda_3 = 6.67398 \qquad \lambda_4 = 10.8801$$

Додатки

Далі наведено програмний код імплементованих алгоритмів. Вихідний код, який було створено для даного практикума (у тому числі LaTeX), можна знайти за наступним посиланням.

Класс для інкупсуляції роботи з пам'ятю

```cpp
#pragma once

#include <cstddef>
#include <algorithm>

using std::size_t;


namespace LinAlg
{
    template <typename T>
    class Buffer
    {
    public:
        explicit Buffer(size_t size)
        {
            _size = size;
            _data = new T[size]();
        }

        Buffer(const Buffer &buff)
        {
            _data = new T[buff._size];
            _size = buff._size;
            std::copy(buff._data, buff._data + buff._size, _data);
        }

        Buffer(Buffer &&lhs) noexcept
        {
            _data = std::exchange(lhs._data, nullptr);
            _size = std::exchange(lhs._size, 0);
        }

        ~Buffer()
        {
            delete[] _data;
        }

        Buffer& operator=(const Buffer &buff)
        {
            if (&buff == this)
```

```
                return *this;

            T *_newData = new T[buff._size];
            std::copy(buff._data, buff._data + buff._size,
               ↪ _newData);

            _size = buff._size;
            delete[] _data;
            _data = _newData;

            return *this;
        }

        Buffer& operator=(Buffer &&buff) noexcept
        {
            std::swap(_data, buff._data);
            std::swap(_size, buff._size);
            return *this;
        }

        T& operator[](std::size_t pos)
        {
            return _data[pos];
        }

        const T& operator[](std::size_t pos) const
        {
            return _data[pos];
        }

        [[nodiscard]] inline size_t size() const noexcept {
           ↪ return _size; }

    private:
        T *_data;
        size_t _size;
    };
}
```

Клас векторів

```
#pragma once

#include <cmath>
#include "Buffer.h"

namespace LinAlg
{
    template<typename T>
    class Vector
    {
```

```
public:
    explicit Vector(size_t nval) : _buff(nval) {}

    template<typename U>
    Vector(const Vector<U> &vector) : _buff(vector.nval())
    {
        for (size_t i = 0; i < vector.nval(); i++)
        {
            _buff[i] = vector(i);
        }
    };

    inline size_t nval() const noexcept
    {
        return _buff.size();
    }

    T& operator()(size_t i);

    const T &operator()(size_t i) const;

    auto norm() const noexcept;

    template<typename U>
    friend Vector<U> operator-(const Vector<U> &vector);

private:
    Buffer <T> _buff;
};

template<typename T>
T &Vector<T>::operator()(size_t i)
{
    if (i >= _buff.size())
        throw std::invalid_argument("Position is out of range");
    return _buff[i];
}

template<typename T>
const T& Vector<T>::operator()(size_t i) const
{
    if (i >= _buff.size())
        throw std::invalid_argument("Position is out of range");
    return _buff[i];
}

template<typename T>
auto Vector<T>::norm() const noexcept
```

```cpp
{
    T sum = 0;
    for (size_t i = 0; i < _buff.size(); i++)
    {
        sum += _buff[i] * _buff[i];
    }
    return std::sqrt(sum);
}

template<typename T>
Vector<T> operator-(const Vector<T> &vector)
{
    Vector<T> result(vector.nval());
    for (size_t i = 0; i < vector.nval(); i++)
    {
        result(i) = -vector(i);
    }
    return result;
}

template<typename T, typename U>
auto operator+(const Vector<T> &rhs, const Vector<U> &lhs)
{
    if (rhs.nval() != lhs.nval())
        throw std::invalid_argument("Different size");

    Vector<typename std::common_type<T, U>::type> result(rhs.
       ↪ nval());
    for (size_t i = 0; i < rhs.nval(); i++)
    {
        result(i) = rhs(i) + lhs(i);
    }
    return result;
}

template<typename T, typename U>
auto operator-(const Vector<T> &rhs, const Vector<U> &lhs)
{
    if (rhs.nval() != lhs.nval())
        throw std::invalid_argument("Different size");

    Vector<typename std::common_type<T, U>::type> result(rhs.
       ↪ nval());
    for (size_t i = 0; i < rhs.nval(); i++)
    {
        result(i) = rhs(i) - lhs(i);
    }
    return result;
}
```

```
    template<typename T>
    T getMaxAbsValue(const Vector<T> &vector)
    {
        T max = std::abs(vector(0));

        for (size_t i = 1; i < vector.nval(); i++)
        {
            if (max < std::abs(vector(i)))
                max = std::abs(vector(i));
        }

        return max;
    }
}
```

Клас матриць

```
#pragma once

#include "Buffer.h"
#include "Vector.h"

namespace LinAlg
{
    template <typename T>
    class IMatrix
    {
    public:
        [[nodiscard]] size_t nrow() const noexcept { return _nrow; }
        [[nodiscard]] size_t ncol() const noexcept { return _ncol; }

        virtual T &operator()(size_t i, size_t j) = 0;
        virtual const T& operator()(size_t i, size_t j) const = 0;

    protected:
        size_t _nrow;
        size_t _ncol;
    };

    template <typename T>
    class Matrix : public IMatrix<T>
    {
    public:
        Matrix(size_t nrow, size_t ncol) : _buff(nrow * ncol)
        {
            this->_nrow = nrow;
            this->_ncol = ncol;
        }
```

```
template<typename U>
Matrix(const Matrix<U>& matrix) : _buff(matrix._buff.size
    ↪ ())
{
    for (size_t i = 0; i < matrix.nrow(); i++)
    {
        for (size_t j = 0; j < matrix.ncol(); j++)
        {
            _buff[i] = matrix(i, j);
        }
    }
};

T &operator()(size_t i, size_t j) override
{
    if (i >= this->_nrow || j >= this->_ncol)
        throw std::invalid_argument("Position␣is␣out␣of␣
            ↪ range");
    return _buff[i * this->_ncol + j];
}

const T& operator()(size_t i, size_t j) const override
{
    if (i >= this->_nrow || j >= this->_ncol)
        throw std::invalid_argument("Position␣is␣out␣of␣
            ↪ range");
    return _buff[i * this->_ncol + j];
}

Matrix<T> transpose()
{
    Matrix<T> result(this->_nrow, this->_ncol);

    for (size_t i = 0; i < this->_nrow; i++)
    {
        for (size_t j = 0; j < this->_ncol; j++)
        {
            result(i, j) = operator()(j, i);
        }
    }

    return result;
}

[[nodiscard]] bool isSymmetric() const noexcept
{
    if (this->_ncol != this->_nrow)
        return false;
```

```
        for (size_t i = 0; i < this->_nrow; i++)
        {
            for (size_t j = 0; j < i; j++)
            {
                if (operator()(i, j) != operator()(j, i))
                    return false;
            }
        }

        return true;
    }

private:
    Buffer<T> _buff;
};

template <typename T, typename U>
auto operator*(const IMatrix<T> &rhs, const IMatrix<U> &lhs)
{
    if (rhs.ncol() != lhs.nrow())
        throw std::invalid_argument("Number␣of␣column␣!=␣
          ↪ number␣of␣row");

    Matrix<typename std::common_type<T, U>::type> result(rhs.
        ↪ nrow(), lhs.ncol());

    for (size_t i = 0; i < rhs.nrow(); i++)
    {
        for (size_t j = 0; j < lhs.ncol(); j++)
        {
            for (size_t k = 0; k < rhs.ncol(); k++)
            {
                result(i, j) += rhs(i, k) * lhs(k, j);
            }
        }
    }

    return result;
}

template <typename T, typename U>
auto operator*(const IMatrix<T> &matrix, const Vector<U> &
    ↪ vector)
{
    if (matrix.ncol() != vector.nval())
        throw std::invalid_argument("Number␣of␣column␣!=␣size
          ↪ ␣of␣vector");

    Vector<typename std::common_type<T, U>::type> result(
        ↪ matrix.nrow());
```

```
        for (size_t i = 0; i < matrix.nrow(); i++)
        {
            for (size_t j = 0; j < matrix.ncol(); j++)
            {
                result(i) += vector(j) * matrix(i, j);
            }
        }

        return result;
    }
}
```

Утілити для матриць

```
#pragma once

#include <unordered_map>

#include "Vector.h"
#include "Matrix.h"

namespace LinAlg
{
    template<typename T>
    class MatrixView : public IMatrix<T>
    {
    public:
        explicit MatrixView(IMatrix<T> &matrix)
        {
            this->_matrix = &matrix;
            this->_nrow = matrix.nrow();
            this->_ncol = matrix.ncol();
        }

        const T& operator()(size_t i, size_t j) const override
        {
            return this->_matrix->operator()(this->getRow(i), j);
        }

        T& operator()(size_t i, size_t j) override
        {
            return this->_matrix->operator()(this->getRow(i), j);
        }

        void swapRows(size_t i, size_t j)
        {
            if (i == j)
                return;

            size_t iNew, jNew;
```

```
        auto it = _rowMap.find(i);
        if (it != _rowMap.end())
            jNew = it->second;
        else
            jNew = i;

        it = _rowMap.find(j);
        if (it != _rowMap.end())
            iNew = it->second;
        else
            iNew = j;

        _rowMap.insert({i, iNew});
        _rowMap.insert({j, jNew});
    }

protected:
    size_t getRow(size_t i) const noexcept
    {
        auto it = _rowMap.find(i);
        if (it != _rowMap.end())
            i = it->second;

        return i;
    }

    std::unordered_map<size_t, size_t> _rowMap;
    IMatrix<T>* _matrix;
};

template <typename T>
class TransposeView : public MatrixView<T>
{
public:
    explicit TransposeView(IMatrix<T> &matrix) : MatrixView<T
        ↪ >(matrix)
    {
        this->_ncol = matrix.nrow();
        this->_nrow = matrix.ncol();
    }

    T &operator()(size_t i, size_t j) override
    {
        if (j >= this->_nrow || i >= this->_ncol)
            throw std::invalid_argument("Position is out of 
               ↪ range");
        return this->_matrix->operator()(this->getRow(j), i);
    }
```

```
    const T& operator()(size_t i, size_t j) const override
    {
        if (j >= this->_nrow || i >= this->_ncol)
            throw std::invalid_argument("Position␣is␣out␣of␣
              ↪ range");
        return this->_matrix->operator()(this->getRow(j), i);
    }
};

template<typename T>
class MatrixMatrixView : public MatrixView<T>
{
public:
    MatrixMatrixView(IMatrix<T> &matrix1, IMatrix<T> &matrix2
       ↪ ) : MatrixView<T>(matrix1)
    {
        if (matrix1.nrow() != matrix2.nrow())
            throw std::invalid_argument("Invalid␣shapes");

        this->_matrix2 = &matrix2;
        this->_ncol = matrix1.ncol() + matrix2.ncol();
    }

    const T& operator()(size_t i, size_t j) const override
    {
        if (j >= this->_ncol)
            throw std::invalid_argument("Position␣is␣out␣of␣
              ↪ range");

        if (j >= this->_matrix->ncol())
        {
            return this->_matrix2->operator()(this->getRow(i)
              ↪ , j - this->_matrix->ncol());
        }

        return this->_matrix->operator()(i, j);
    }

    T& operator()(size_t i, size_t j) override
    {
        if (j >= this->_ncol)
            throw std::invalid_argument("Position␣is␣out␣of␣
              ↪ range");

        if (j >= this->_matrix->ncol())
        {
            return this->_matrix2->operator()(this->getRow(i)
              ↪ , j - this->_matrix->ncol());
        }
```

```
        return this->_matrix->operator()(i, j);
    }

private:
    IMatrix<T>* _matrix2;
};

template<typename T>
class MatrixVectorView : public MatrixView<T>
{
public:
    MatrixVectorView(IMatrix<T> &matrix, Vector<T> &vector):
      ↪ MatrixView<T>(matrix)
    {
        if (matrix.nrow() != vector.nval())
            throw std::invalid_argument("Invalid␣shapes");

        _vector = &vector;
        this->_ncol = matrix.ncol() + 1;
    }

    const T& operator()(size_t i, size_t j) const override
    {
        if (j >= this->_ncol)
            throw std::invalid_argument("Position␣is␣out␣of␣
              ↪ range");

        i = this->getRow(i);
        if (j >= this->_matrix->ncol())
        {
            return _vector->operator()(i);
        }

        return this->_matrix->operator()(i, j);
    }

    T& operator()(size_t i, size_t j) override
    {
        if (j >= this->_ncol)
            throw std::invalid_argument("Position␣is␣out␣of␣
              ↪ range");

        i = this->getRow(i);
        if (j >= this->_matrix->ncol())
        {
            return _vector->operator()(i);
        }

        return this->_matrix->operator()(i, j);
    }
```

```
    private:
        Vector<T>* _vector;
    };

    template<typename T>
    size_t getMaxRowPivotIndex(const IMatrix<T> &mat, size_t i =
      ↪ 0)
    {
        auto max = i;
        for (size_t l = i + 1; l < mat.nrow(); l++)
        {
            if (std::abs(mat(l, i)) > std::abs(mat(max, i)))
                max = l;
        }
        return max;
    }
}
```

Клас для роботи з поліномами

```
#pragma once

#include "vector"
#include "initializer_list"

namespace LinAlg
{
    template<typename T>
    class Polynomial
    {
    private:
        std::vector<T> m_coeffs;

        void shrinkToFit()
        {
            while (m_coeffs.back() == 0 && m_coeffs.size() > 1)
                m_coeffs.pop_back();
            m_coeffs.shrink_to_fit();
        }

        template<typename U>
        U pow(U value, int n) const
        {
            if (n == 0)
                return 1;
            if (n == 1)
                return value;
            return value * pow(value, n - 1);
        }
```

```cpp
public:
    explicit Polynomial(std::vector<T> coeffs) : m_coeffs(
        ↪ coeffs)
    {
        shrinkToFit();
    }

    Polynomial(std::initializer_list<T> initializerList) :
        ↪ m_coeffs(initializerList)
    {
        shrinkToFit();
    }

    [[nodiscard]] inline int degree() const noexcept
    {
        return m_coeffs.size() - 1;
    }

    template<typename U>
    auto operator()(U value) const
    {
        using ret_type = std::common_type_t<U, T>;
        ret_type result{};

        for (int i = 0; i < m_coeffs.size(); i++)
        {
            result += pow(value, i) * m_coeffs[i];
        }

        return result;
    }

    Polynomial derivate() const
    {
        std::vector<T> derivCoeffs;
        derivCoeffs.reserve(m_coeffs.size() - 1);

        for (int i = 1; i < m_coeffs.size(); i++)
        {
            derivCoeffs.push_back(i * m_coeffs[i]);
        }

        return Polynomial(derivCoeffs);
    }

    auto divmod(const Polynomial &poly) const
    {
        Polynomial mod = *this;
        std::vector<T> divCoeffs;
```

```
        const auto &divider = poly.m_coeffs[poly.degree()];
        while (mod.degree() >= poly.degree())
        {
            auto coeff = mod.m_coeffs[mod.degree()] / divider
              ↪ ;
            divCoeffs.push_back(coeff);

            for (int i = 0; i < poly.degree(); i++)
            {
                auto &modCoeff = mod.m_coeffs[mod.degree() -
                  ↪ 1 - i];
                modCoeff = modCoeff - coeff * poly.m_coeffs[
                  ↪ poly.degree() - 1 - i];
            }

            mod.m_coeffs.pop_back();
        }

        mod.shrinkToFit();
        return std::make_tuple(Polynomial(divCoeffs), mod);
    }

    Polynomial operator%(const Polynomial &poly) const
    {
        auto[div, mod] = divmod(poly);
        return mod;
    }

    Polynomial operator/(const Polynomial &poly) const
    {
        auto[div, _] = divmod(poly);
        return div;
    }

    Polynomial operator-(const Polynomial &poly) const
    {
        std::vector<T> coeffs;
        if (poly.degree() > degree())
        {
            for (int i = 0; i < m_coeffs.size(); i++)
                coeffs.push_back(m_coeffs[i] - poly.m_coeffs[
                  ↪ i]);
            for (int i = degree() + 1; i < poly.m_coeffs.size
              ↪ (); i++)
                coeffs.push_back(-poly.m_coeffs[i]);
        } else
        {
            for (int i = 0; i < poly.m_coeffs.size(); i++)
                coeffs.push_back(m_coeffs[i] - poly.m_coeffs[
                  ↪ i]);
```

```
            for (int i = poly.degree() + 1; i < m_coeffs.size
                ↪ (); i++)
                coeffs.push_back(m_coeffs[i]);
        }

        return Polynomial(coeffs);
    }

    template<typename U>
    friend Polynomial<U> operator-(const Polynomial<U> &);
};

template<typename T>
Polynomial<T> operator-(const Polynomial<T> &poly)
{
    auto result = poly;
    for (auto &coeff: result.m_coeffs)
    {
        coeff = -coeff;
    }
    return result;
}
}
```

Методи для обрахунку коренів поліномів

```
#pragma once

#include "vector"
#include "algorithm"
#include "Polynomial.h"

namespace LinAlg
{
    namespace detail
    {
        template<typename T>
        auto sturmSequence(const Polynomial<T> &poly)
        {
            std::vector<Polynomial<T>> sequence;

            sequence.push_back(poly);
            if (poly.degree() != 0)
                sequence.push_back(poly.derivate());

            while (sequence.back().degree() > 0)
            {
                auto &p2 = sequence[sequence.size() - 2];
                auto &p1 = sequence[sequence.size() - 1];
                sequence.push_back(-(p2 % p1));
            }
```

```
    return sequence;
}

template<typename T, typename U>
int calculateSignChange(const std::vector<Polynomial<T>>
  ↪ &sturmSequence, U x)
{
    std::vector<T> f;
    std::transform(sturmSequence.begin(), sturmSequence.
      ↪ end(), std::back_inserter(f),
                  [x](const auto& poly){ return  poly(x);
                    ↪  });

    int result = 0;
    for (auto it = f.cbegin(); it != f.cend() - 1; it++)
    {
        if(*it >= 0 && *(it+1) < 0)
            result++;
        if(*it < 0 && *(it+1) >= 0)
            result++;
    }

    return result;
}

template<typename T, typename U1, typename U2>
int calculateRoots(const std::vector<Polynomial<T>> &
  ↪ sturmSequence, U1 a, U2 b)
{
    return std::abs(calculateSignChange(sturmSequence, a)
      ↪  - calculateSignChange(sturmSequence, b));
}

template<typename T>
auto splitRoots(const Polynomial<T> &poly)
{
    const double INFTY = 2 << 20;
    const double scale = 0.5;
    const double step = 2;

    auto sturmSeq = sturmSequence(poly);
    int positiveRootsNumber = calculateRoots(sturmSeq, 0,
      ↪  INFTY);
    int negativeRootsNumber = calculateRoots(sturmSeq, -
      ↪ INFTY, 0);
    std::vector<std::tuple<T, T>> ranges;

    auto a = 0;
    auto b = a + step;
```

```
        while (positiveRootsNumber > 0)
        {
            int rootsCount = calculateRoots(sturmSeq, a, b);
            if (rootsCount == 0)
            {
                a = b;
                b = a + step;
            }
            else if (rootsCount > 1)
            {
                b = a + step * scale;
            }
            else
            {
                ranges.emplace_back(a, b);
                a = b;
                b = a + step;
                positiveRootsNumber--;
            }
        }

        b = 0;
        a = b - step;
        while (negativeRootsNumber > 0)
        {
            int rootsCount = calculateRoots(sturmSeq, a, b);
            if (rootsCount == 0)
            {
                b = a;
                a = b - step;
            }
            else if (rootsCount > 1)
            {
                a = b - step * scale;
            }
            else
            {
                ranges.emplace_back(a, b);
                b = a;
                a = b - step;
                negativeRootsNumber--;
            }
        }

        return ranges;
    }
}

template<typename T>
auto newton(const Polynomial<T> &poly, double precision =
```

```
        ↪ 0.00001)
    {
        std::vector<T> roots;
        auto df = poly.derivate();

        for (const auto& range: detail::splitRoots(poly))
        {
            auto[a, b] = range;
            auto x = (a + b) / 2;
            auto f = poly(x);

            while (std::abs(f) >= precision )
            {
                x = x - f / df(x);
                f = poly(x);
            }

            roots.push_back(x);
        }

        return roots;
    }
}
```

Методи знаходження власних значень

```
#pragma once

#include <vector>
#include <cmath>

#include "Vector.h"
#include "Matrix.h"
#include "MatrixUtils.h"
#include "Polynomial.h"
#include "PolynomialSolvers.h"
#include "Solvers.h"

#ifdef PRINT
#include "Utils.h"
#endif

namespace LinAlg
{
    namespace detail
    {
        template<typename T>
        auto maxNonDiagonal(const Matrix<T> &matrix)
        {
            T max = 0;
            size_t maxI;
```

```
        size_t maxJ;

        for (size_t i = 0; i < matrix.nrow(); i++)
        {
            for (size_t j = 0; j < matrix.ncol(); j++)
            {
                auto absValue = std::abs(matrix(i, j));
                if (i != j && absValue > max)
                {
                    max = absValue;
                    maxI = i;
                    maxJ = j;
                }
            }
        }

        return std::make_tuple(max, maxI, maxJ);
    }
}

template<typename T>
auto jacobi(const Matrix<T> &matrix, double precision =
   ↪ 0.00001, int maxIteration = 1000)
{
    if (matrix.ncol() != matrix.nrow())
        throw std::invalid_argument("Invalid␣sizes");

    auto A = matrix;
    auto[maxNd, i, j] = detail::maxNonDiagonal(A);

    int iteration = 0;
    do
    {
        auto tau = (A(j, j) - A(i, i)) / (2 * maxNd);
        auto tan = -tau + std::sqrt(1 + tau * tau);
        auto c = 1 / std::sqrt(1 + tan * tan);
        auto s = tan * c;

        for (size_t k = 0; k < A.nrow(); k++)
        {
            if (k != i && k != j)
            {
                auto aki = A(k, i);
                A(i, k) = A(k, i) = c * aki - s * A(k, j);
                A(j, k) = A(k, j) = s * aki + c * A(k, j);
            }
        }
        auto aii = A(i, i);
        A(i, i) = c * c * aii - 2 * c * s * A(i, j) + s * s *
           ↪ A(j, j);
```

```
            A(j, j) = c * c * A(j, j) + 2 * c * s * A(i, j) + s *
               ↪  s * aii;
            A(i, j) = A(j, i) = 0;

#ifdef PRINT
            std::cout << "Iteration:␣" << iteration + 1 << std::
               ↪ endl;

            std::cout << "Matrix␣T" << std::endl;
            auto F = eye<T>(A.nrow());
            F(i, i) = F(j, j) = c;
            F(j, i) = -s;
            F(i, j) = s;
            print(F);

            T Sd = 0;
            T Snd = 0;
            for (size_t i = 0; i < A.nrow(); i++)
            {
                for (size_t j = 0; j < A.ncol(); j++)
                {
                    if (i == j)
                        Sd += A(i, i) * A(i, i);
                    else
                        Snd += A(i, j) * A(i, j);
                }
            }
            std::cout << "Spherical␣norm␣diag:␣" << Sd << std::
               ↪ endl;
            std::cout << "Spherical␣norm␣non␣diag:␣" << Snd <<
               ↪ std::endl;
            std::cout << "Spherical␣norm:␣" << Sd + Snd << std::
               ↪ endl;
#endif

            std::tie(maxNd, i, j) = detail::maxNonDiagonal(A);
            iteration++;
        } while (maxNd > precision && iteration < maxIteration);

        std::vector<T> eigenValues;
        for (size_t i = 0; i < A.nrow(); i++)
        {
            eigenValues.push_back(A(i, i));
        }

        return eigenValues;
    }

    template<typename T>
    auto danilevsky(const Matrix<T> &matrix, double precision =
```

```
        0.00001)
    {
        if (matrix.ncol() != matrix.nrow())
            throw std::invalid_argument("Invalid sizes");

        auto P = matrix;
        for (int i = P.nrow() - 2; i >= 0; i--)
        {
            auto M = eye<T>(P.nrow());
            auto MInv = eye<T>(P.nrow());

            for (int j = 0; j < P.ncol(); j++)
            {
                MInv(i, j) = P(i + 1, j);
                if (i != j)
                {
                    M(i, j) = -P(i + 1, j) / P(i + 1, i);
                } else
                {
                    M(i, i) = 1 / P(i + 1, i);
                }
            }

#ifdef PRINT
            std::cout << "Danilevsky method. Step: " << P.nrow()
                - 1 - i << std::endl;
            std::cout << "Matrix M" << std::endl;
            print(M);
            std::cout << "Matrix M inverse" << std::endl;
            print(MInv);
#endif

            P = MInv * P * M;
        }

#ifdef PRINT
        std::cout << "Frobenius form" << std::endl;
        print(P);
#endif

        std::vector<T> characteristicCoeffs;
        int I = P.nrow() % 2 == 0 ? -1 : 1;
        for (size_t i = P.ncol(); i > 0; i--)
            characteristicCoeffs.push_back(I * P(0, i-1));
        characteristicCoeffs.push_back(-I);

        Polynomial characteristic(characteristicCoeffs);
        return newton(characteristic, precision);
    }
```

```
    template<typename T>
    auto krylov(const Matrix<T> &matrix, const Vector<T> &initialY, double precision = 0.00001)
    {
        if (matrix.ncol() != matrix.nrow())
            throw std::invalid_argument("Invalid sizes");

        Matrix<T> A(matrix.nrow(), matrix.ncol());
        Vector<T> b = initialY;

        for (size_t i = 0; i < A.nrow(); i++)
        {
            for (size_t j = 0; j < b.nval(); j++)
                A(j, A.nrow() - (i + 1)) = b(j);
            b = matrix * b;
        }
        if (A.nrow() % 2 == 0)
            b = -b;

#ifdef PRINT
        std::cout << "Krylov" << std::endl;
        std::cout << "Matrix A" << std::endl;
        print(A);
        std::cout << "Vector b" << std::endl;
        print(b);
#endif

        auto p = gaussPivotSolve(A, b);

        std::vector<T> characteristicCoeffs;
        for (size_t i = 0; i < p.nval(); i++)
        {
            characteristicCoeffs.push_back(p(p.nval() - i - 1));
        }
        characteristicCoeffs.push_back(1);

        Polynomial characteristic(characteristicCoeffs);
        return newton(characteristic, precision);
    }
}
```