Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»
Дисциплина «Численные методы»

Лабораторная работа №2

Тема: численные методы решения нелинейных уравнений

Студент: Глушатов И.С.
Группа: М8О-307Б-19
Преподаватель: Ревизников Д. Л.
Дата:
Оценка:

Москва, 2022

Лабораторная работа № 2.1

Задание: реализовать методы простой итерации и Ньютона решения нелинейных уравнений в виде программ, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения найти положительный корень нелинейного уравнения (начальное приближение определить графически). Проанализировать зависимость погрешности вычислений от количества итераций.

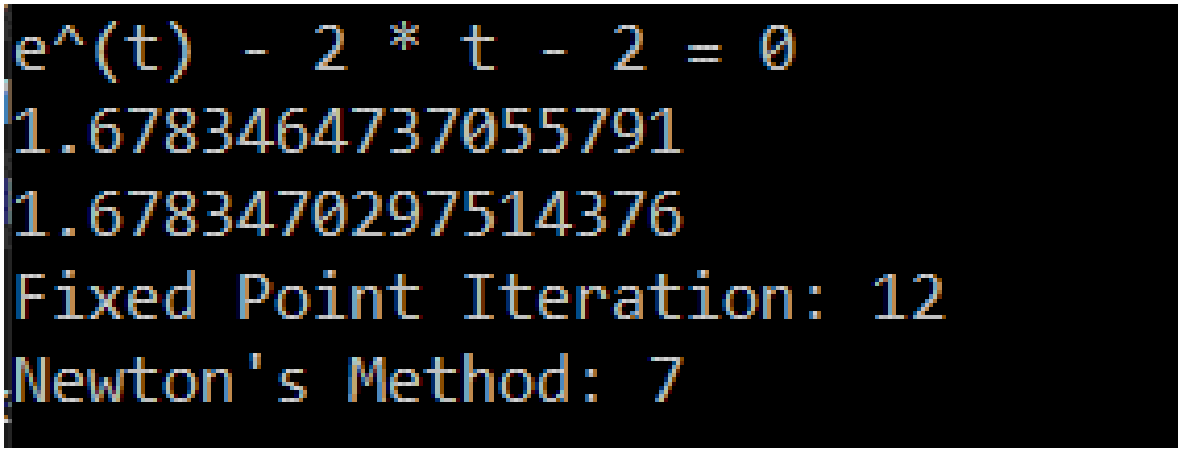
6. $e^x - 2x - 2 = 0$.

Листинг

```
#include <iostream>
#include <vector>
#include <cmath>
#include <functional>
#include <iomanip>

using namespace std;

void print() {
    std::cout << std::endl;
}

template<class T>
std::ostream& operator<<(std::ostream& out, const std::vector<T>& v) {
    for (auto& a : v) {
        out << a << " ";
    }
    return out;
}

template<class T>
void print(T obj) {
    std::cout << obj << std::endl;
}

template<class T, class ...Args>
void print(T obj, Args... args) {
    std::cout << obj;
    print(args...);
}

void cnt(int flag = -1) {
    static int a = 0;
    static int b = 0;

    switch (flag) {
    case 0:
        a++;
        break;
    case 1:
```

```
        b++;
        break;
    default:
        print("Fixed Point Iteration: ", a);
        print("Newton's Method: ", b);
        break;
    }
}

template<class T, class F>
T FixedPointIteration(std::function<F> phi, T a, T b, T q, T eps) {

    T x_prev = (a + b) / 2;
    T x_next = phi(x_prev);
    T err = q / (1 - q) * std::abs(x_next - x_prev);

    while (err > eps) {
        cnt(0);
        x_prev = x_next;
        x_next = phi(x_prev);
        err = q / (1 - q) * std::abs(x_next - x_prev);
    }

    return x_next;
}

template<class T, class F>
T FixedPointIteration(std::function<F> phi, T a, T b, T eps) {

    T q = -1;
    T dx = std::max(eps, T(0.000001));

    for (T i = 1; i <= (b - a) / dx; i++) {
        T dy = phi(a + i * dx) - phi(a + (i - 1) * dx);
        T tg = dy / dx;
        q = std::max(q, abs(tg));
    }

    return FixedPointIteration(phi, a, b, q, eps);
}

template<class T, class F>
T NewtonsMethod(std::function<F> phi, std::function<F> dphi, T x0, T eps) {

    T x_prev = x0;
    T err = phi(x_prev) / dphi(x_prev);
    T x_next = x_prev - err;


    while (std::abs(err) > eps) {
        cnt(1);
        x_prev = x_next;
        err = phi(x_prev) / dphi(x_prev);
        x_next = x_prev - err;
    }

    return x_next;
}

int main()
{
    print("e^(t) - 2 * t - 2 = 0");

    auto func = [](double t) -> double {
        return std::log(2 * t + 2);
```

```
    };

    print(std::setprecision(17),
FixedPointIteration(std::function<double(double)>(func), 1., 2., 0.000001));

    auto f = [](double t) -> double {
        return std::exp(t) - 2 * t - 2;
    };

    auto df = [](double t) -> double {
        return std::exp(t) - t;
    };

    print(std::setprecision(17), NewtonsMethod(std::function<double(double)>(f),
std::function<double(double)>(df), 0., 0.000001));
    cnt();


    //print(f(FixedPointIteration(std::function<double(double)>(func), 1., 2.,
0.0001)));
    //print(f(NewtonsMethod(std::function<double(double)>(f),
std::function<double(double)>(df), 0., 0.0001)));
}
```



Лабораторная работа № 1.2

Задание: реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то из них, в котором значения неизвестных являются положительными); начальное приближение определить графически. Проанализировать зависимость погрешности вычислений от количества итераций.

4	1	$\begin{cases} x_1 - \cos x_2 = 1, \\ x_2 - \lg(x_1 + 1) = a. \end{cases}$
5	2	
6	3	

Листинг

```
#include <iostream>
#include <vector>
#include <cmath>
#include <functional>
#include <iomanip>
#include <cassert>

using namespace std;

void print() {
    std::cout << std::endl;
}

template<class T>
std::ostream& operator<<(std::ostream& out, const std::vector<T>& v) {
    for (size_t i = 0; i < v.size(); i++) {
        out << v[i];

        if (i != v.size() - 1) {
            out << " ";
        }
    }
    return out;
}

template<class T>
void print(T obj) {
    std::cout << obj << std::endl;
}

template<class T, class ...Args>
void print(T obj, Args... args) {
    std::cout << obj;
    print(args...);
}

template<class T>
```

```
class Matrix {

    vector<vector<T>> matrix;
    std::vector<Matrix<T>> plu;

public:

    Matrix(size_t _n) {
        matrix = vector<vector<T>>(_n, vector<T>(_n, T()));
    }

    Matrix(size_t _n, vector<T> v) : Matrix(_n) {
        for (size_t i = 0; i < _n; i++) {
            for (size_t j = 0; j < _n; j++) {
                matrix[i][j] = v[i * _n + j];
            }
        }
    }

    Matrix(size_t _n, std::vector<std::vector<T>>& matr) : Matrix(_n) {
        for (size_t i = 0; i < _n; i++) {
            for (size_t j = 0; j < _n; j++) {
                matrix[i][j] = matr[i][j];
            }
        }
    }

    void SwapColumns(size_t i, size_t j) {
        for (size_t k = 0; k < size(); k++) {
            std::swap(matrix[k][i], matrix[k][j]);
        }
    }

    void SwapLines(size_t i, size_t j) {
        for (size_t k = 0; k < size(); k++) {
            std::swap(matrix[i][k], matrix[j][k]);
        }
    }

    Matrix<T> E(size_t _n) {
        Matrix<T> result(_n);
        for (size_t i = 0; i < _n; i++) {
            result[i][i] = 1;
        }
        return result;
    }

    std::vector<Matrix<T>> LUFactorizing(int* count = nullptr) {
        Matrix<T> P = E(size());
        Matrix<T> L = E(size());
        Matrix<T> U(size(), matrix);

        /*std::cout << P << std::endl;
        std::cout << L << std::endl;
        std::cout << U << std::endl;*/

        for (size_t i = 0; i < size(); i++) {

            // 1. Находим строку с максимальным по модулю элементом.
            {
                size_t k = i;
                T max = std::abs(U[i][i]);

                for (size_t j = i + 1; j < size(); j++) {
                    if (std::abs(U[j][i]) > max) {
```

```
                    max = std::abs(U[j][i]);
                    k = j;
                }
            }

            if (U[k][i] == 0) {
                continue;
            }

            // 2. Меняем строки в U и обновляем L.
            if (k != i) {
                P.SwapColumns(i, k);
                L.SwapLines(i, k);
                L.SwapColumns(i, k);
                U.SwapLines(i, k);
                if (count != nullptr) (*count) += 1;
            }
        }

        // 3. Алгоритм Гаусса
        for (size_t j = i + 1; j < size(); j++) {
            double koef = U[j][i] / U[i][i];

            U[j][i] = 0;
            L[j][i] = koef;

            for (size_t t = i + 1; t < size(); t++) {
                U[j][t] -= koef * U[i][t];
            }
        }
    }

    /*std::cout << P << std::endl;
    std::cout << L << std::endl;
    std::cout << U << std::endl;*/

    return std::vector<Matrix<T>>({ P, L, U });
}

std::vector<T> Solve(const std::vector<T>& b) {
    // A * x = b => P * L * U * x = b => L * U * x = P^(-1) * b = P^(T) * b

    if (b.size() != size()) throw "размерность не совпадает";

    // 1. Делаем LU - разложение
    if (plu.size() == 0) {
        plu = LUFactorizing();
    }

    // 2. Вычисляем P^(T) * b = b * P = y
    auto y = b * plu[0];
    // 3. Вычисляем L * z = y;

    std::vector<T> z(size(), T());

    for (size_t i = 0; i < size(); i++) {
        z[i] = y[i];

        for (size_t j = 0; j < i; j++) {
            z[i] -= plu[1][i][j] * z[j];
        }

        z[i] /= plu[1][i][i];
    }
```

```
    // 4. Вычисляем U * x = z
    std::vector<T> x(size(), T());

    for (long i = size() - 1; i >= 0; i--) {
        x[i] = z[i];

        for (long j = i + 1; j < size(); j++) {
            x[i] -= plu[2][i][j] * x[j];
        }

        x[i] /= plu[2][i][i];
    }

    return x;
}

friend Matrix<T> operator*(const Matrix<T>& m1, const Matrix<T>& m2) {
    std::vector<std::vector<T>> vec(m1.size(), std::vector<T>(m1.size(), T()));
    for (size_t i = 0; i < m1.size(); i++) {
        for (size_t j = 0; j < m1.size(); j++) {
            for (size_t k = 0; k < m1.size(); k++) {
                vec[i][j] +=
                    m1[i][k] *
                    m2[k][j];
            }
        }
    }

    return Matrix<T>(m1.size(), vec);
}

friend std::vector<T> operator*(const Matrix<T>& m1, const std::vector<T>& m2) {
    if (m1.size() != m2.size()) {
        throw "bad thing";
    }

    std::vector<T> result(m1.size(), T());
    for (size_t i = 0; i < m1.size(); i++) {
        for (size_t j = 0; j < m1.size(); j++) {
            result[i] += m1[i][j] * m2[j];
        }
    }

    return result;
}

friend std::vector<T> operator*(const std::vector<T>& m2, const Matrix<T>& m1) {
    if (m1.size() != m2.size()) {
        throw "bad thing";
    }

    std::vector<T> result(m1.size(), T());
    for (size_t i = 0; i < m1.size(); i++) {
        for (size_t j = 0; j < m1.size(); j++) {
            result[i] += m1[j][i] * m2[j];
        }
    }

    return result;
}

size_t inline size() {
    return matrix.size();
}
```

```
    size_t size() const {
        return matrix.size();
    }

    vector<T>& operator[] (size_t i) {
        return matrix[i];
    }

    vector<T> operator[] (size_t i) const {
        return matrix[i];
    }

    vector<T> operator* (vector<T> v) {
        vector<T> result(size(), T());

        for (size_t i = 0; i < size(); i++) {
            for (size_t j = 0; j < size(); j++) {
                result[i] += matrix[i][j] * v[j];
            }
        }

        return result;
    }

    Matrix<T> operator+ (const Matrix<T>& m) {
        Matrix<T> result(size());

        for (size_t i = 0; i < size(); i++) {
            for (size_t j = 0; j < size(); j++) {
                result[i][j] = matrix[i][j] + m[i][j];
            }
        }

        return result;
    }

    friend ostream& operator<<(ostream& out, Matrix<T> m) {
        for (size_t i = 0; i < m.size(); i++) {
            for (size_t j = 0; j < m.size(); j++) {
                out << m[i][j] << " ";
            }
            out << endl;
        }
        return out;
    }
};

template<class T>
vector<T> operator+ (vector<T> v1, vector<T> v2) {
    vector<T> result(v1.size(), T());

    for (size_t i = 0; i < v1.size(); i++) {
        result[i] = v1[i] + v2[i];
    }

    return result;
}

template<class T>
vector<T> operator- (vector<T>& v1, vector<T>& v2) {
    vector<T> result(v1.size(), T());

    for (size_t i = 0; i < v1.size(); i++) {
        result[i] = v1[i] - v2[i];
    }
```

```
    return result;
}

template<class T>
vector<T> operator- (vector<T> v1) {
    vector<T> result(v1);

    for (size_t i = 0; i < v1.size(); i++) {
        result[i] = -result[i];
    }

    return result;
}

void cnt(int flag = -1) {
    static int a = 0;
    static int b = 0;

    switch (flag) {
    case 0:
        a++;
        break;
    case 1:
        b++;
        break;
    default:
        print("Fixed Point Iteration: ", a);
        print("Newton's Method: ", b);
        break;
    }
}

template<class T>
T Norm(vector<T> v) {
    T max = std::abs(v[0]);

    for (T& e : v) {
        max = std::max(max, std::abs(e));
    }

    return max;
}

template<class T, class F>
std::vector<T> FixedPointIteration(const std::function<F> phi, const size_t size,
std::vector<T> start, const T q, const T eps) {

    std::vector<T> X_prev(start);
    std::vector<T> X(start);

    auto E = [&]() -> T {
        return q / (1 - q) * Norm(X - X_prev);
    };

    do {
        X_prev = X;
        X = phi(X);
        cnt(0);
    } while (std::abs(E()) > eps);

    return X;
}

template<class T, class F, class M>
```

```
std::vector<T> NewtonsMethod(const std::function<F> f, const std::function<M>
jacobian, const size_t size, std::vector<T> start, const int w, const T eps) {

    std::vector<T> X(start);
    Matrix<T> J = jacobian(X);
    std::vector<T> dX = J.Solve(-f(X));

    auto E = [&]() -> T {
        return Norm(dX);
    };

    int times = 1;

    do {
        if (times == 0)
            J = jacobian(X);

        dX = J.Solve(-f(X));
        X = X + dX;
        cnt(1);
        times = (times + 1) % w;

    } while (std::abs(E()) > eps);

    return X;
}

int main()
{
    print("x1 - cos(x2) = 1");
    print("x2 - lg(1 + x1) = 3\n");

    print("f(X) = (x1 - cos(x2) - 1; x2 - lg(1 + x1) - 3)\n");

    print("Fixed Point Iteration:");
    print("x1 = 1 + cos(x2):");
    print("x2 = 3 + lg(1 + x1)\n");

    auto f = [](std::vector<double> X) -> std::vector<double> {
        assert((X.size() == 2) && "size must be 2");

        return std::vector<double>({ X[0] - std::cos(X[1]) - 1, X[1] - std::log10(1 +
X[0]) - 3 });
    };

    auto phi = [](std::vector<double> X) -> std::vector<double> {
        assert((X.size() == 2) && "size must be 2");

        return std::vector<double>({1 + std::cos(X[1]), 3 + std::log10(1 + X[0])});
    };

    auto jacobian = [](std::vector<double> X) -> Matrix<double> {
        assert((X.size() == 2) && "size must be 2");

        Matrix<double> result(2,
            {                                  1, std::sin(X[1]),
            -1 / (std::log(10.0) * (X[0] + 1)),            1});

        return result;
    };

    auto res =
FixedPointIteration(std::function<std::vector<double>(std::vector<double>)>(phi), 2, {
0.0, 3.0 }, 0.5, 0.000001);
    print("X = (", res, ")\n", "f(X) = (", f(res), ")");
```

```
    res = NewtonsMethod(std::function<std::vector<double>(std::vector<double>)>(f),
std::function<Matrix<double>(std::vector<double>)>(jacobian), 2, { 0.0, 3.0 }, 1,
0.00000000001);
    print("X = (", res, ")\n", "f(X) = (", f(res), ")");

    cnt();
}
```

```
x1 - cos(x2) = 1
x2 - lg(1 + x1) = 3

f(X) = (x1 - cos(x2) - 1; x2 - lg(1 + x1) - 3)

Fixed Point Iteration:
x1 = 1 + cos(x2):
x2 = 3 + lg(1 + x1)

X = (0.00943979 3.00408)
f(X) = (-1.23161e-07 0)
X = (0.00943991 3.00408)
f(X) = (0 0)
Fixed Point Iteration: 8
Newton's Method: 4
```