Лабораторная работа №1

Цель: Построение и программная реализация алгоритма полиномиальной интерполяции табличных функций.

Исходные данные

1. Таблица функции и её производных

x	y	y'
0.00	1.000000	-1.000000
0.15	0.838771	-1.14944
0.30	0.655336	-1.29552
0.45	0.450447	-1.43497
0.60	0.225336	-1.56464
0.75	-0.018310	-1.68164
0.90	-0.278390	-1.78333
1.05	-0.552430	-1.86472

2. Степень аппроксимирующего полинома - n.

3. Значение аргумента для которого выполняется интерполяция.

Код программы

main.cpp

```
#include "GL/glew.h"
#include "GLFW/glfw3.h"
#include "imgui.h"
#include "imgui_impl_glfw.h"
#include "imgui_impl_opengl3.h"
#include "implot.h"
#include "hermite.hpp"
#include "imfilebrowser.h"
#include <iostream>
```

```
#include <string>

static void error_callback(int error, const char* description)
{
    fputs(description, stderr);
}
static void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GLFW_TRUE);
}

int main(int argc, char *argv[])
{

    /* GLFW */

    if (!glfwInit())
    {
        std::cout << "[FAIL] can't init glfw.\n";
        return -1;
    }
    else
    {
        std::cout << "[SUCCESS] glfw was inited.\n";
    }
#ifdef __APPLE__
    const char *glsl_version = "#version 150";
    glfwWindowHint (GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint (GLFW_CONTEXT_VERSION_MINOR, 2);
    glfwWindowHint (GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
    glfwWindowHint (GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    float highDPIscaleFactor = 1.0;
#endif

    GLFWwindow* window;

    window = glfwCreateWindow(800, 600, "lab1", NULL, NULL);


    if (!window)
    {
        std::cout << "[FAIL] can't create window.\n";
        glfwTerminate();
        return -1;
    }

    glfwSetErrorCallback(error_callback);
    glfwSetKeyCallback(window, key_callback);
    glfwMakeContextCurrent(window);

    /* GLFW */

    /* ================== */
    /* GLEW */
    if(GLEW_OK != glewInit())
    {
        std::cout << "[FAIL] can't init glew.\n";
        return -1;
    }
```

```
else
{
    std::cout << "[SUCCESS] glew was initialized.\n";
}
/* GLEW */

/* IMGUI */
IMGUI_CHECKVERSION();
ImGui::CreateContext();
ImGuiIO &io = ImGui::GetIO();
(void)io;

ImGui_ImplGlfw_InitForOpenGL(window, true);
ImGui_ImplOpenGL3_Init(glsl_version);

io.Fonts->AddFontDefault();
ImVec4 clear_color = ImVec4(0.3f, 0.55f, 0.60f, 1.00f);


ImGui::FileBrowser fileDialog;

/* IMGUI */

/* IMPLOT */
ImPlot::CreateContext();
/* IMPLOT */

int polynomSize = 1;
double x;

result tmp;

std::string selectedFile;

std::vector<result> res;


interpolation obj(1);

while (!glfwWindowShouldClose(window))
{
    ImGui_ImplOpenGL3_NewFrame();
    ImGui_ImplGlfw_NewFrame();


    ImGui::NewFrame();

    ImGui::Begin("Lab");


    ImGui::Text("This is some useful text.");

    ImGui::InputInt("Polynom Size", &polynomSize);

    ImGui::InputDouble("X", &x);

    if (ImGui::Button("Newton"))
    {
        obj.setPolynomSize(polynomSize);
        obj.loadFile(selectedFile);
```

```
    obj.tableSlice(x);
    tmp.y = obj.Newtonf(x);
    tmp.x = x;
    tmp.type = "Newton";
    tmp.polynomSize = polynomSize;
    res.push_back(tmp);
}

if (ImGui::Button("Hermite"))
{
    obj.setPolynomSize(polynomSize);
    obj.loadFile(selectedFile);
    obj.tableSlice(x);
    tmp.y = obj.Hermitef(x);
    tmp.x = x;
    tmp.type = "Hermite";
    tmp.polynomSize = polynomSize;
    res.push_back(tmp);
}


if (ImGui::Button("Root"))
{
    obj.setPolynomSize(polynomSize);
    obj.loadFile(selectedFile);
    obj.tableSlice(x);
    obj.invertTable();

    tmp.y = obj.Newtonf(0.0);
    tmp.x = 0.0f;
    tmp.type = "Root";

    tmp.polynomSize = polynomSize;
    res.push_back(tmp);
}

if (ImGui::Button("Clear table"))
{
    res.clear();
}




if (ImGui::BeginMainMenuBar())
{
    if (ImGui::BeginMenu("File"))
    {
        if (ImGui::MenuItem("Open..", "Ctrl+O"))
        {
            fileDialog.Open();

        }
        ImGui::EndMenu();
    }
    ImGui::EndMainMenuBar();
}

ImGui::BeginChild("Scrolling");
for (int n = 0; n < res.size(); n++)
```

```
            ImGui::Text("[%d] Type: %7s Polynom size: %d Result: y(%7lf) = %7lf", n + 1, (res[n].type).c_str(),
res[n].polynomSize, res[n].x, res[n].y);
        ImGui::EndChild();

        ImGui::End();

        fileDialog.Display();

        if(fileDialog.HasSelected())
        {
            selectedFile = fileDialog.GetSelected();
            fileDialog.ClearSelected();
        }
        ImGui::Render();


        glClearColor(clear_color.x, clear_color.y, clear_color.z, clear_color.w);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);


        ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());

        glfwSwapBuffers(window);


        glfwPollEvents();
    }

    glDisableVertexAttribArray(0);

    ImGui_ImplOpenGL3_Shutdown();
    ImGui_ImplGlfw_Shutdown();
    ImGui::DestroyContext();

    glfwDestroyWindow(window);
    glfwTerminate();
    return 0;
}
```

algorithm.cpp

```
#include "hermite.hpp"

#include <iostream>
#include <fstream>
#include <algorithm>
#include <iterator>
#include <stdbool.h>
#include <math.h>

template<typename T>
void print_vector(std::vector<T> const &v)
{
```

```
    for (int i = 0; i < v.size(); i++)
    {
        std::cout << v[i] << " ["  << i << "]"<<  "\n";
    }
}
template<typename T>
std::vector<T> sub(std::vector<T> const &v, int begin, int end)
{
    auto first = v.begin() + begin;
    auto last = v.begin() + end + 1;

    std::vector<T> tmp(first, last);

    return tmp;

}

static double findx;

bool biggerThan(record elem)
{
    return(elem.x >= findx);
}

interpolation::interpolation(unsigned int polynomSize)
{
    this->polynomSize = polynomSize;
}

void interpolation::loadFile(std::string name)
{
    std::ifstream f;
    f.open(name, std::ios::in);

    double x, y, dy;
    record tmp;

    table.records.clear();

    while (f >> x >> y >> dy)
    {
        tmp.x = x; tmp.y = y; tmp.dy = dy;

        table.records.push_back(tmp);
    }

    f.close();
    std::sort(
            table.records.begin(),
            table.records.end(),
            [](record a, record b){
                return a.x < b.x;
            }
    );
}

void interpolation::tableSlice(double x)
{
    findx = x;
    size_t len = table.records.size();
```

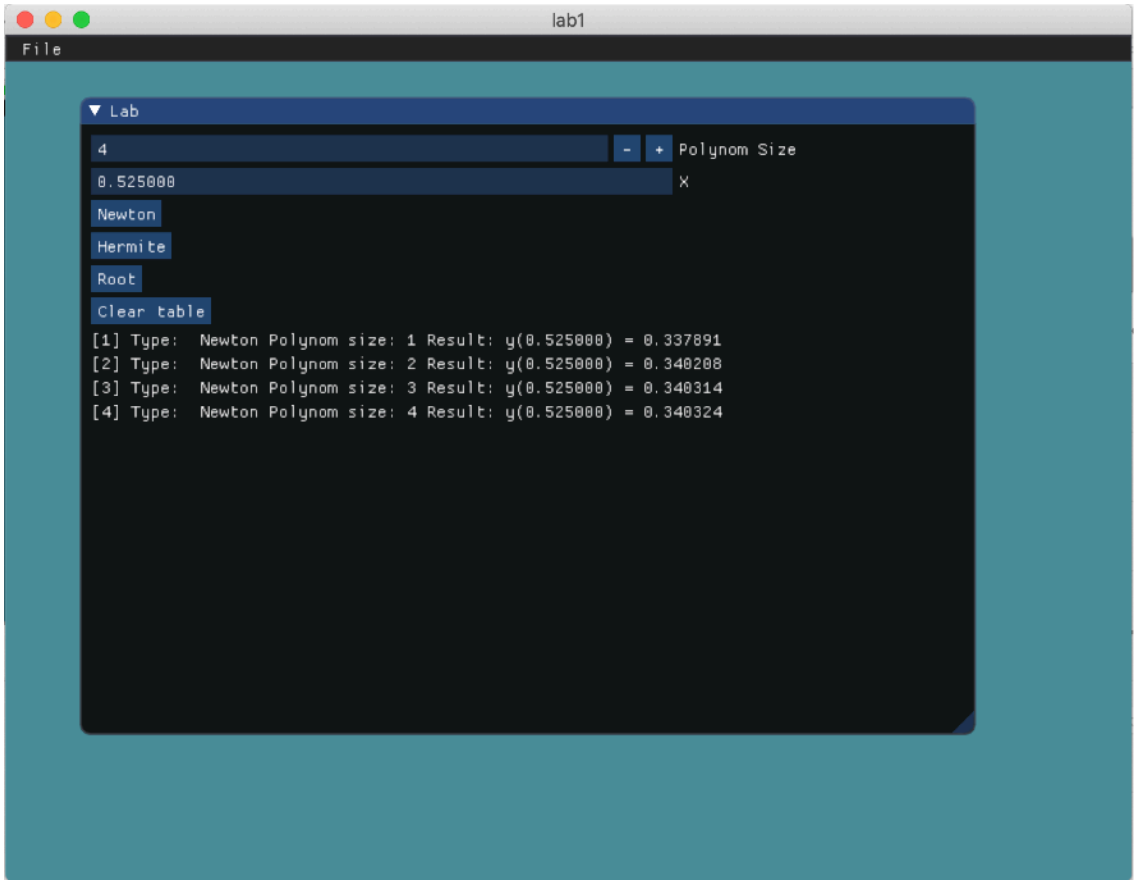
```
    unsigned int num  =
        this->polynomSize + 1;

    std::vector<record>::iterator it =
        std::find_if(this->table.records.begin(),
              table.records.end(), biggerThan);

    std::cout << it->x << "\n";
    if (it != this->table.records.end())
    {
        int half = ceil(num / 2.f);


        int ind = it - table.records.begin();

        while(ind < len && half > 0)
        {
            half--;
            ind++;
        }


        ind -= num;


        if (ind < 0)
            ind = 0;

        table.records = std::vector<record>
            (table.records.begin() + ind, table.records.begin() + ind + num);

        this->setSeparateDiffs();

        this->tableDuplicate();

        this->setHermiteDiffs();

        return;
    }

    table.records = sub(table.records, len - 1 - polynomSize, len);

    this->setSeparateDiffs();

    this->tableDuplicate();

    this->setHermiteDiffs();

    return;
}

void interpolation::setSeparateDiffs()
{
    std::vector<double> next;

    std::vector<double> x;

    std::vector<double> y;
```

```
    newtonDiffs.clear();

    for (std::vector<record>::iterator it = this->table.records.begin(); !(it == table.records.end()); it++)
    {
        x.push_back((*it).x); y.push_back((*it).y);

    }

    newtonDiffs.push_back(y[0]);

    for (unsigned int i = 1; i < this->table.records.size(); ++i)
    {
        next.clear();
        for (unsigned int t = 0; t < y.size() - 1; ++t)
        {
            next.push_back( ( y[ t + 1 ] - y[ t ] ) / ( x[ ( t + i ) ] - x[ t ] ));
        }

        y = next; newtonDiffs.push_back(y[0]);
    }

}

void interpolation::setHermiteDiffs()
{

    std::vector<double> next;

    std::vector<double> x;

    std::vector<double> y;

    std::vector<double> dy;


    hermiteDiffs.clear();

    for (std::vector<record>::iterator it = this->hermite.records.begin(); !(it == this->hermite.records.end()); it+
+)
    {
        x.push_back((*it).x);
        y.push_back((*it).y);
        dy.push_back((*it).dy);
    }

    hermiteDiffs.push_back(y[0]);

    for (unsigned int i = 1; i < hermite.records.size(); ++i)
    {
        next.clear();
        for (unsigned int t = 0; t < y.size() - 1; ++t)
        {
            if (i == 1)
            {
                if (x[t] == x[t + 1])
                    next.push_back(dy[t]);
                else
                    next.push_back((y[t + 1] - y[t]) / (x[t + i] - x[t]));

            }
```

```
            else
            {
                next.push_back((y[t + 1] - y[t]) / (x[t + i] - x[t]));
            }

        }

        y = next; hermiteDiffs.push_back(y[0]);
    }

}

void interpolation::invertTable()
{
    double tmp;
    for(auto &j: table.records)
    {
        tmp = j.x;
        j.x = j.y;
        j.y = tmp;
    }
    setSeparateDiffs();
}

void interpolation::tableDuplicate()
{
    this->hermite.records.clear();
    for(auto &v: table.records)
    {
        this->hermite.records.push_back(v);
        this->hermite.records.push_back(v);
    }
}

double interpolation::Newtonf(double x)
{

    double yy = table.records[0].y;
    double vl = 0.f;

    for (int i = 1; i < polynomSize + 1; i++)
    {
        vl = newtonDiffs[i];



        for (int j = 0; j < i; j++)
        {
            vl *= (x - table.records[j].x);
        }
        yy += vl;
    }

    return yy;
}

double interpolation::Hermitef(double x)
{

    double _y = hermite.records[0].y;
```

```
    for (double i = 1; i < polynomSize + 1; i++)
    {
        double val = hermiteDiffs[i];

        for (double j = 0; j < i; j++)
        {
            val *= (x - hermite.records[j].x);
        }
        _y += val;

    }
    return _y;
}

void interpolation::tablePrint()
{

    std::cout << "\n==========\n\n";

    for (auto &j: this->table.records)
    {
        std::cout << "|" << j.x << "|" << j.y << "|" << j.dy << "|" << "\n";
    }
    std::cout << "\n==========\n\n";
}

void interpolation::setPolynomSize(unsigned int polynomSize)
{
    this->polynomSize = polynomSize;
}
```

Для графического представления результата использовались сторонние библиотеки ImGui,



Результаты работы

Значения y(x) при степенях полиномов Ньютона и Эрмита n=1, 2, 3 и 4 при фиксированном x, например, x=0.525 (середина интервала 0.45-0.60). Результаты свести в таблицу для сравнения полиномов.

Полином Ньютона при x = 0.525

Степень полинома	Значение функции
0	0.225336
1	0.337891
2	0.340208
3	0.340314
4	0.340324

Полином Эрмита при x = 0.525

Степень полинома	Значение функции
0	0.225336
1	0.342824
2	0.340358
3	0.340312
4	0.340324

Корень функции вычисленный при помощи обратной интерполяции Ньютона.

Степень полинома	Корень функции
0	0.75
1	0.750150
2	0.739174
3	0.739050
4	0.739081

Контрольные вопросы:

1. Будет ли работать программа при степени полинома n=0?

– Да, в таком случае результатом интерполяции будет построен по одному узлу.

2. Как практически оценить погрешность интерполяции? Почему сложно применить для этих целей теоретическую оценку?

– Погрешность многочлена Ньютона можно оценить по формуле

$|y(x) - P_n(x)| \le \frac{M_{n+1}}{(n+1)!} |\varpi_n(x)|$, где

$M_{n+1} = \max |y^{n+1}(\xi)|$

$\varpi_n(x) = \prod (x - x_i)$

Трудность использования теоретических оценок заключается в том, что производные интерполируемой функции скорее всего неизвестны, тогда для определения погрешности удобнее использовать оценку первого отброшенного члена.

3. Если в двух точках заданы значения функции и ее первых производных, то полином какой минимальной степени может быть построен на этих точках?

Минимальная – 0, и максимальная – 3.

4. В каком месте алгоритма построения полинома существенна информация об упорядоченности аргумента функции (возрастает, убывает)?

Порядок нумерации узлов безразличен.

5. Что такое выравнивающие переменные и как их применить для повышения точности интерполяции?

Выравнивающие переменные - это такие переменные $\eta=\eta(y)$ $\xi=\xi(x)$, что график $\eta(\xi)$ близок к прямой, хотя бы на отдельных участках. В случае быстроизменяющихся функций интерполяцию проводят на этих переменных, а зачем приводят обратно к (y, x). Это помогает избегать составления таблиц больших объемов.