Julia Plewa
MOwNiT
Laboratorium 2

1. **Zaimplementować metodę Interpolate1D (najprościej w sposóby analogiczny do metody Interpolate2D). (Cpp*) Zmienić implementacje template'a w taki sposób, aby przyjmował vector punktów (np. PointsList2D) i zwracał również taki wektor, ale odpowiednio "docięty" (ang. Clamped vector/list).**

   Kod:

```
void Interpolate1D(int pointsToInterpolate) override
{
    std::ofstream outfile;
    outfile.open("cubic.txt");
    std::vector<int> index(pointsToInterpolate);
    std::vector<float> t;
    std::vector<float> tx;
    int i = 0;
    int points_size = pointsList.size() - 1;
    std::generate(index.begin(), index.end(), [&i, &pointsToInterpolate,
                  &points_size, &t, &tx]()
    {
        float percent = ((float) i) / (float(pointsToInterpolate - 1));
        tx.push_back((points_size)* percent);
        t.push_back(tx[i] - floor(tx[i]));
        return int(tx[i++]);
    });
    for (int i = 0; i < pointsToInterpolate; ++i)
    {
        PolynomialCoeffs coeffs;
        std::array<float, 2> A = GetIndexClamped(pointsList, index[i] - 1);
        std::array<float, 2> B = GetIndexClamped(pointsList, index[i]);
        std::array<float, 2> C = GetIndexClamped(pointsList, index[i] + 1);
        std::array<float, 2> D = GetIndexClamped(pointsList, index[i] + 2);

        coeffs.A = A[0];
        coeffs.B = B[0];
        coeffs.C = C[0];
        coeffs.D = D[0];
```

```
        float x = CubicHermite(coeffs, t[i]);
        std::cout << "Value at " << tx[i] << " = " << x << std::endl;
        outfile << tx[i] << " " << x << std::endl;
    }
    outfile.close();
};
```

Przykładowe wyniki:

Wejście (drugi wymiar pomijany):	Wyjście:	

const PointsList2D ex1Points =
{
{ 0.0f, 1.1f },
{ 1.6f, 8.3f },
{ 2.3f, 6.5f },
{ 3.5f, 4.7f },
{ 4.3f, 3.1f },
{ 5.9f, 7.5f },
{ 6.8f, 0.0f },
};
	Value at 0 = 0
	Value at 0.5 = 0.75625
	Value at 1 = 1.6
	Value at 1.5 = 1.975
	Value at 2 = 2.3
	Value at 2.5 = 2.89375
	Value at 3 = 3.5
	Value at 3.5 = 3.875
	Value at 4 = 4.3
	Value at 4.5 = 5.09375
		Value at 5 = 5.9
		Value at 5.5 = 6.45
		Value at 6 = 6.8

2. **Zaimplementować metodę interpolacji Lagrange'a wpasowując się do powyższego schematu (może być jedynie interpolacja 1D). Porównać obie metody (wystarczy pod kątem teoretycznym, nie implementacyjnym).**

Kod:

```
float LagrangeInterpolate (float x)
{
    float sum = 0.0f;
    for (int i = 0; i < pointsList.size(); ++i)
    {
        float lx = 1.0;
        for (int j = 0; j < pointsList.size(); ++j)
        {
            if (j != i)
            {
                lx *= (x - pointsList[j][0]) / (pointsList[i][0] - pointsList[j][0]);
            }
        }
        sum += pointsList[i][1] * lx;
    }
    return sum;
}
```

```
void Interpolate1D(int pointsToInterpolate) override
{
    std::ofstream outfile;
    outfile.open("lagrange.txt");
    for (int i = 0; i < pointsToInterpolate; ++i)
    {
        float percent = ((float)i) / (float(pointsToInterpolate - 1));
        float x = pointsList.back()[0] * percent;
        float y = LagrangeInterpolate(x);
        std::cout << "Value at " << x << " = " << y << std::endl;
        outfile << x << " " << y << std::endl;
    }
    outfile.close();
};
```

<u>Przykładowe wyniki:</u>

<table>
<tr><th>Wejście:</th><th colspan="2">Wyjście:</th></tr>
<tr><td><pre>const PointsList2D ex2Points =
{
    { 0.0f, 0.0f },
    { 1.0f, 1.6f },
    { 2.0f, 2.3f },
    { 3.0f, 3.5f },
    { 4.0f, 4.3f },
    { 5.0f, 5.9f },
    { 6.0f, 6.8f },
};</pre></td><td><pre>Value at 0 = 0
Value at 0.5 = 1.39883
Value at 1 = 1.6
Value at 1.5 = 1.80664
Value at 2 = 2.3
Value at 2.5 = 2.93633
Value at 3 = 3.5
Value at 3.5 = 3.91289
Value at 4 = 4.3
Value at 4.5 = 4.91133</pre></td><td><pre>Value at 5 = 5.9
Value at 5.5 = 6.95664
Value at 6 = 6.8</pre></td></tr>
</table>

Porównanie:

Cubic Hermite Spline Interpolation między każdymi kolejnymi dwoma punktami ma postać

$$f(x) = ax^3 + bx^2 + cx + d.$$

Zarówno $f(x)$ jak i jej pochodna

$$f'(x) = 3ax^2 + 2bx + c$$

są funkcjami ciągłymi.

Każde cztery kolejne równoodległe punkty kontrolne y_{-1}, y_0, y_1, y_2 wyznaczają fragment funkcji przebiegający między punktami y_0 i y_1. Współczynniki wielomianu między tymi punktami mają wartości:

$$\begin{gathered} a = -\frac{y_{-1}}{2} + \frac{3y_0}{2} - \frac{3y_1}{2} + \frac{y_2}{2}, \\ b = y_{-1} - \frac{5y_0}{2} + 2y_1 - \frac{y_2}{2}, \\ c = -\frac{y_{-1}}{2} + \frac{y_1}{2}, \\ d = y_0. \end{gathered}$$

Złożoność obliczeniowa *Cubic Hermite Spline Interpolation* nie rośnie w przypadku zwiększenia liczby punktów wejściowych.

Dla zbioru k punktów (x_i, y_i) gdzie $f(x_i) = y_i$ interpolacja Lagrange'a tworzy funkcję

$$L(x) = \sum_{j=0}^{k} y_j \ell_j(x),$$

gdzie $\ell_j(x)$ to następująca funkcja bazowa następującej postaci:

$$\ell_j(x) = \prod_{0 \le m \le k,\, m \ne j} \frac{x - x_m}{x_j - x_m}.$$

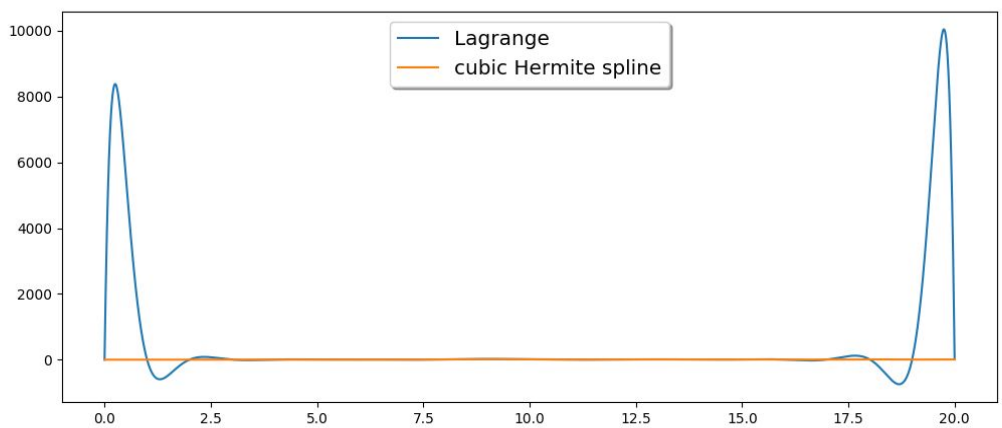
W wyniku interpolacji Lagrange'a dla n punktów wejściowych powstaje wielomian stopnia $n-1$. Punkty wejściowe nie muszą być równomiernie rozłożone (ale wartości x muszą być unikalne). Złożoność obliczeniowa rośnie wraz z liczbą punktów na wejściu.

Interpolacja Lagrange'a nie jest odporna na tzw. *efekt Rungego* czyli pogorszenie jakości interpolacji wielomianowej mimo zwiększenia liczby węzłów (szczególnie widoczne na końcach przedziałów).

Porównanie wyników przykładowych wywołań:

```
const PointsList2D ex2ComparisonLagrange =
{
    { 0,  0.0f }, { 1.0f,  1.6f }, { 2.0f,  2.3f }, { 3.0f,  3.5f }, { 4.0f,  4.3f },
    { 5.0f,  5.9f }, { 6.0f,  6.8f }, { 7.0f,  1.2f }, { 8.0f,  8.1f },
    { 9.0f,  20.5f }, { 10.0f, 10.0f }, { 11.0f,  1.8f }, { 12.0f,  5.2f },
    { 13.0f,  10.1f }, { 14.0f,  3.5f }, { 15.0f,  8.0f },  { 16.0f,  1.8f },
    { 17.0f,  5.2f }, { 18.0f,  10.1f }, { 19.0f,  3.5f }, { 20.0f,  8.0f }
};
```

Dla wartości $pointsToInterpolate = 10000$ oraz przedstawionych powyżej par (x, y) wykorzystanych przy interpolacji Lagrange'a (oraz analogicznie samych drugich elementów powyższej tablicy w przypadku *Cubic Spline Interpolation)* otrzymano następujące wyniki:



Na wykresie wyraźnie zauważalne duże odchylenia na krańcach przedziału w przypadku interpolacji Lagrange'a - jest to właśnie wspomniany wcześniej efekt Rungego. Warto podkreślić, że jest on szczególnie dotkliwy w przypadku punktów znajdujących się w równych odstępach od siebie (jak w tej sytuacji), a pomocne może być wykorzystanie być wielomianów Czebyszewa.

3. **Napisz funkcję liczącą błąd średniokwadratowy. Na wejściu musi dostawać dwie tablice/dwa wektory równej długości, a na wyjściu ma zwracać sumę kwadratów różnic pomiędzy kolejnymi elementami tych wektorów.**

Kod:

```
float mse(std::vector<float> x, std::vector<float> y)
{
    if (x.size() != y.size())
    {
        throw exceptions::IncorrectVectorsException();
    }
    float mse = 0;
    for (int i = 0; i < x.size(); i++)
    {
        mse += pow(x[i] - y[i], 2);
    }
    return mse / x.size();
}
```

Wzór:

$$MSE(x,y) = \frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2$$
$$RMSE(x,y) = \sqrt{MSE(x,y)}$$

4. **Napisz funkcję pobierającą dwa wektory floatów i zwracającą parametry a i b prostej o równaniu y = ax + b, będącej najlepszą aproksymacją tych punktów.**

```
float vectorElemSum(std::vector<float> v)
{
    float result = 0;
    for (int i = 0; i < v.size(); i++)
    {
        result += v[i];
    }
    return result;
}

float vectorElemMean(std::vector<float> v)
{
   return vectorElemSum(v) / v.size();
}
```

```cpp
std::vector<float> subtractValueFromVectorElems(std::vector<float> x, float y)
{
    std::vector<float> result;
    for (int i = 0; i < x.size(); i++)
    {
        result.push_back(x[i] - y);
    }
    return result;
}

std::vector<float> multiplyVectorElems(std::vector<float> x, std::vector<float> y)
{
    std::vector<float> result;
    for (int i = 0; i < x.size(); i++)
    {
        result.push_back(x[i] * y[i]);
    }
    return result;
}

std::pair<float, float> linearRegression(std::vector<float> x, std::vector<float> y)
{
    float xMean = vectorElemMean(x);
    float yMean = vectorElemMean(y);
    std::vector<float> xMinusMean = subtractValueFromVectorElems(x, xMean);
    std::vector<float> yMinusMean = subtractValueFromVectorElems(y, yMean);
    float a = vectorElemSum(multiplyVectorElems(xMinusMean, yMinusMean));
    a /= vectorElemSum(multiplyVectorElems(xMinusMean, xMinusMean));
    float b = yMean - a * xMean;
    std::cout << "y = " << a << " * x + "  << b << std::endl;
    return std::make_pair(a, b);
}
```

Wzory:

Regresja liniowa metodą najmniejszych kwadratów umożliwia dopasowanie do zbioru punktów prostej o równaniu $y = a \cdot x + b$, gdzie

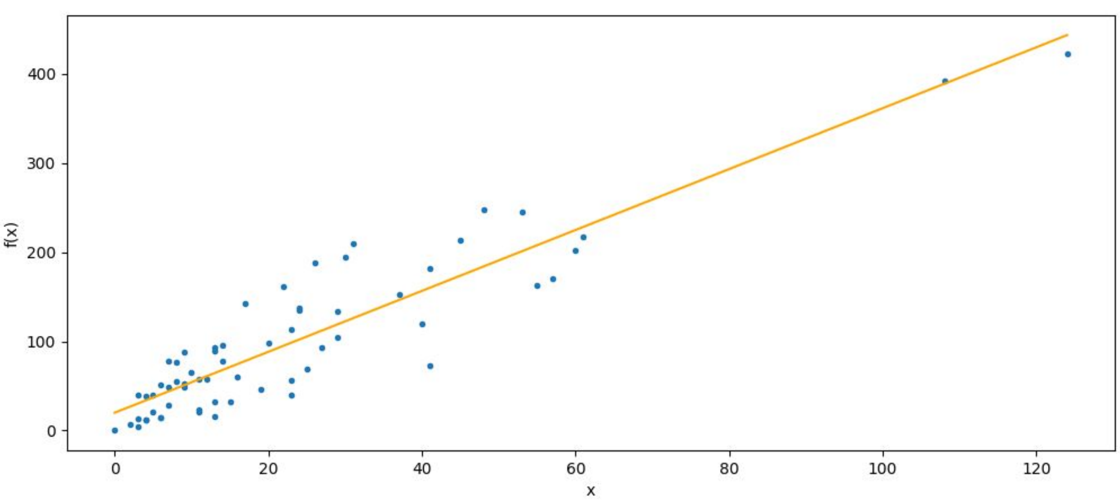
$$a = \frac{\sum_{i=1}^{N}(x_i - \overline{x}) \times (y_i - \overline{y})}{\sum_{i=1}^{N}(x_i - \overline{x})^2} \quad ,$$

$$b = \overline{y} - b \cdot \overline{x} \ .$$

Dla prostej o takiej postaci zminimalizowany jest błąd średniokwadratowy.

5. **Wynik najlepiej przedstawić na wykresie w postaci umieszczenia na nim punktów oraz dopasowanej do nich prostej (najłatwiej dane z C++ zrzucić do pliku lub po prostu skopiować z konsoli, a następnie wykres sporządzić za pomocą gnuplot lub matplotlib - python). Dla odważnych - można korzystać z bibliotek C++ (np. QtCharts).**

Wykres:



Wykorzystany został zbiór danych *Auto Insurance in Sweden* podany w konspekcie. Dopasowana została do niego prosta o wzorze

$$y = 3.41382 \cdot x + 19.9945 \, .$$

6. (*) Zadanie 6 Napisz klasę enkapsulującą model regresji liniowej.

Kod:

```
class LinearRegressor
{
    public:
        LinearRegressor() { }

        float fit(std::vector<float> x, std::vector<float> y)
        {
            coefficients = regression::linearRegression(x, y);
            float mse = regression::mse(y, predict(x));
            return mse;
        }

        std::vector<float> predict(std::vector<float> x)
        {
            std::vector<float> y;
            for (int i = 0; i < x.size(); i++)
            {
                y.push_back(coefficients.first * x[i] + coefficients.second);
            }
            return y;
        }
    private:
        std::pair<float, float> coefficients;
};
```

Wyniki:

Dla prostej o wzorze

$$y = 3.41382 \cdot x + 19.9945 .$$

dopasowanej do datasetu *Auto Insurance in Sweden* błąd średniokwadratowy MSE oraz jego pierwiastek kwadratowy $RMSE$ wynoszą kolejno:

$$MSE(y, y_{predicted}) = 1250,74$$
$$RMSE(y, y_{predicted}) = 35,3658$$

Wykres:

Do wizualizacji wyników wykorzystany został dataset z poprzedniego podpunktu.