Zadanie 1 W powyższym przykładzie zaimplementować metodę Interpolate1D (najprościej w sposóby analogiczny do metody Interpolate2D).

Kod:

```
void Interpolate1D(int pointsToInterpolate) override
{
	// TASK1
	std::vector<int> index(pointsToInterpolate);
	std::vector<float> t;
	std::vector<float> tx;

	int i = 0, 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;
	}
}
```

Output:

```
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
```

Zadanie 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:

```
void Interpolate1D(int pointsToInterpolate) override
{
	// TASK2
	std::vector<int> index(pointsToInterpolate);
	std::vector<float> x;

	int i = 0, points_size = pointsList.size() - 1;
	std::generate(index.begin(), index.end(), [&i, &pointsToInterpolate,
			&points_size, &x]()
	{
		float percent = ((float)i) / (float(pointsToInterpolate - 1));
		x.push_back((points_size)* percent);
		return int(x[i++]);
	});

	for(int k = 0; k < pointsToInterpolate; k++){
		// loop for each value to calculate
		float result = 0.0f;

		for (int i = 0; i < points_size; i++)
		{
			// summing
			float product = pointsList[i][2];
			for (int j = 0; j < points_size; j++)
			{   // multiplicating
				if (j!=i) product = product * (x[k] -
		pointsList[j][0]) / double(pointsList[i][0] - pointsList[j][0]);
			}

			result += product;
		}

		std::cout << "Value at " << x[k] << " = " << result << std::endl;
	}
}
```

Output:

```
Value at 0 = 1.6
Value at 0.5 = -0.02791
Value at 1 = 0.711671
Value at 1.5 = 2.05063
Value at 2 = 3.11045
Value at 2.5 = 3.67009
Value at 3 = 3.934
Value at 3.5 = 4.3
Value at 4 = 5.12725
Value at 4.5 = 6.50418
Value at 5 = 8.01645
Value at 5.5 = 8.51486
Value at 6 = 5.88331
```

1. Interpolacja Hermite'a:

- Obliczanie odbywa się dla zestawów czterech punktów $\{P_{-1}, P_0, P_1, P_2\}$.
- Funkcja interpolująca jest zawsze wielomianem trzeciego stopnia
  $w(x) \ = \ ax^3 + bx^2 + cx + d$, gdzie:
  $a \ = (-P_{-1} + 3P_0 - 3P_1 + P_2 \ )/2$
  $b \ = (P_{-1} - 5/2 \ * P_0 + 2P_1 - P_2/2$
  $c \ = (-P_{-1} + P_1 \ )/2$
  $d \ = P_0$
- Funkcja jest klasy C1
- Odporna na efekt Rungego

2. Interpolacja Lagrange'a:

- Funkcja interpolująca jest wielomianem $n - 1$ stopnia (n - ilość węzłów interpolujących)

$$w(x) \ = \sum_{i=0}^{n} y_i \prod_{j=0, j \neq i}^{n} \frac{x - x_j}{x_i - x_j}$$

- ww. iloczyn jest równy 1 dla $x = x_i$ i równy 0 dla $x = x_j$. Czyni to tę metodą prostą do interpolowania dowolnej funkcji.
- Występuje efekt Rungego - po przekroczeniu pewnej ilości węzłów przybliżenie funkcji przestaje się poprawiać, a zaczynają występować na granicach przedziału *znaczne* wzrosty i spadki od bazowej krzywej.

Zadanie 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)
{
	// TASK3
	if(x.size() != y.size()){
		return NAN;
	}

	float result = 0.0f;
	for(int i=0;i<x.size();i++){
		result += pow((x[i]-y[i]),2);
	}
	return result/x.size();
}
```

Output dla losowych wektorów z wartościami z zakresu (0,10):

x	y
9.38047	2.70302
7.26341	5.86444
5.69628	6.73482
2.35084	1.06571
6.93289	8.23786
3.33293	8.2519
7.34367	2.53914
9.73449	4.92538
2.12256	2.31025
2.79641	0.660115
mse(x,y) = 12.5985	

Błąd średniokwadratowy:

$$MSE = 1/n * \sum_{i=0}^{n} (x_i - y_i)$$

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

$$y = a * x + b$$

$$a = \sum_{i=0}^{n}((x_i - \overline{x}) * (y_i - \overline{y})) / \sum_{i=0}^{n}(x_i - \overline{x}^2)$$

$$b = \overline{y} - a * \overline{x}$$

Kod:

```
std::pair<float, float> linearRegression(std::vector<float> x,
std::vector<float> y)
{
	// TASK4
	std::pair<float, float> coeffs;  // y = ax + b
	if(x.size() != y.size()){
		coeffs.first = NAN;
		coeffs.second = NAN;
		return coeffs;
	}

	int size = x.size();
	float meanx=0, meany=0;
	for(int i=0;i<size;i++){
		meanx += x[i];
		meany += y[i];
	}
	meanx /= size;
	meany /= size;

	std::vector<float> errorx (size);
	std::vector<float> errory (size);
	float num = 0.0f;   // sum(errorx*errory)
	float denom = 0.0f; // sum(errorx^2)
	for(int i=0;i<size;i++){
		errorx[i] = x[i] - meanx;
		errory[i] = y[i] - meany;
		num += errorx[i]*errory[i];
		denom += pow(errorx[i],2);
	}

	// a = sum(errorx * errory) / sum(errorx^2)
	coeffs.first = num/denom;
	coeffs.second = meany - coeffs.first * meanx;

	return coeffs;
}
```

Zadanie 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).

Wartości:

x	y
10.0 20.0 40.0 45.0 60.0 65.0 75.0 80.0	40.0 40.0 60.0 80.0 90.0 110.0 100.0 130.0
a = 1.24249	b = 19.9022

Wykres (Python):