Laboratorium 4: Interpolacja

Metody Obliczeniowe w Nauce i Technice

Wiktor Tarsa

1. **Funkcja testowa**

Jako funkcję testową wybrałem funkcję: $f(x) = e^{cox(x)}$. Opisane w tym sprawozdaniu metody będą testowane na przedziale <-3.0, 3.0>. Zaimplementowałem funkcję, która zwraca wartość f(x) w podanym punkcie x:

```
double function(double x){
   return exp(cos(x));
}
```

Stworzyłem strukturę, która reprezentuje punkt:

```
struct point{
   double x;
   double y;
};
```

Zaimplementowałem funkcję, która zwraca wartości funkcji testowej na podanym przedziale, w podanej liczbie punktów. Funkcja ta może służyć zarówno do wyznaczenia dokładnego przebiegu funkcji, jak i do wyznaczenia wartości węzłów interpolacji w ich równoodległym rozmieszczeniu.

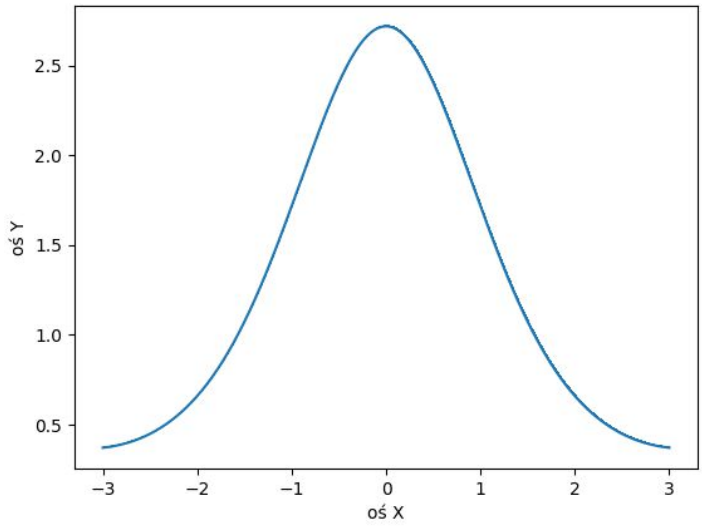
```
std::vector<point> getValueInPoints(double function(double x), int
pointsNumber, double p, double q){
   std::vector<point> values;
   double interval = (q-p)/(pointsNumber-1);
   for(int i = 0; i < pointsNumber; i++){
       point point;
       point.x = p+interval*i;
       point.y = function(point.x);
       values.push_back(point);
   }
   return values;
}
```

2. Wielomian interpolacyjny Lagrange’a

Metoda interpolacji wielomianem Lagrange’a polega na utworzeniu wielomianu złożonego z sumy iloczynów funkcji liniowych, które przechodzą przez określone na początku punkty (węzły interpolacji).

Dokładny przebieg funkcji

Korzystając z wcześniej zaimplementowanej funkcji oraz biblioteki matplotlib języka python wyznaczyłem wartość testowanej funkcji w 10001 punktach na przedziale <-3.0, 3.0>. Na podstawie otrzymanych wyników wygenerowałem wykres funkcji f(x):



Porównanie przebiegu funkcji interpolującej otrzymanej dla różnego rozmieszczenia węzłów(równoodległych i Czebyszewa).

W przypadku węzłów równoodległych zaimplementowałem dwie funkcje: pierwsza z nich wyznacza wartość funkcji interpolującej w podanym punkcie, druga - korzysta z pierwszej i wylicza wartości funkcji na całym przedziale.
Kod źródłowy funkcji wykonującej interpolację Lagrange’a:

```
double interpolateLagrange(std::vector<point> values, double xi){
   int n = values.size();
   double result = 0;
   for(int i = 0; i < n; i++){
       double yi = values[i].y;
       for(int j = 0; j < n; j++){
           if(i != j){
               yi = yi*(xi - values[j].x)/(values[i].x - values[j].x);
           }
       }
       result += yi;
   }
   return result;
}
```

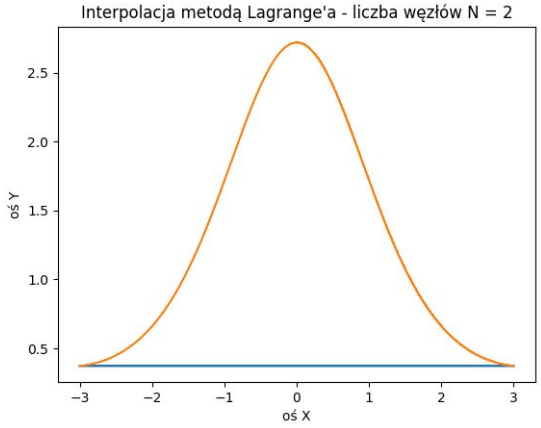
W przypadku węzłów Czebyszewa - zaimplementowałem funkcję wyznaczającą miejsca zerowe wielomianu Czebyszewa o podanym stopniu:

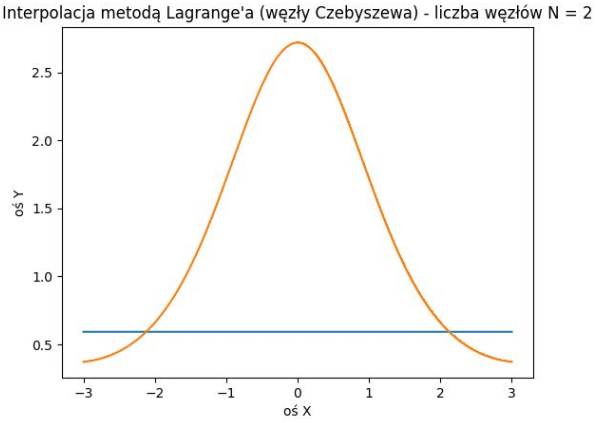
```
std::vector<point> getChebyshevPoints(double function(double), int n){
   std::vector<point> values;
   for(int k = n-1; k >= 0; k--){
       point p;
       p.x = 3*cos((M_PI * (2*k+1))/(2*(n)));
       p.y = function(p.x);
       values.push_back(p);
   }
   return values;
}
```

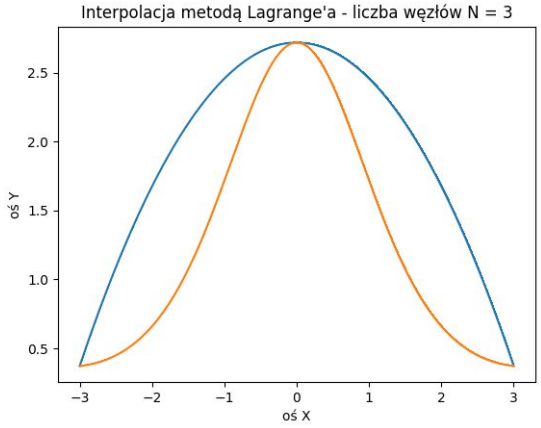
Funkcja również dokonuje transformacji na przedział <-3.0, 3.0>.
Do wykonania interpolacji wykorzystałem wcześniej zaimplementowana funkcję.

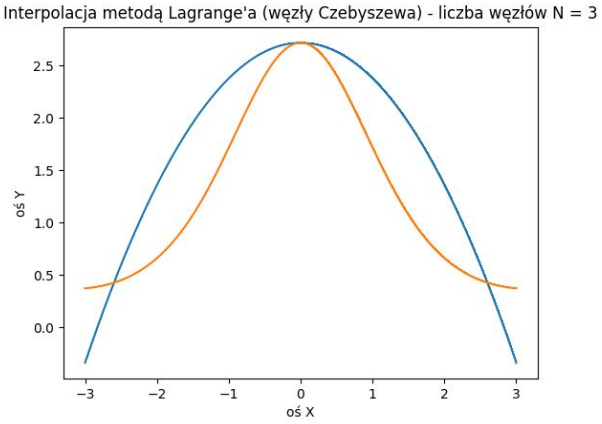
Wartości funkcji wyliczyłem w 10001 równoodległych punktach na przedziale <-3.0, 3.0>.
Wykonałem interpolację dla sześciu różnych ilości węzłów.

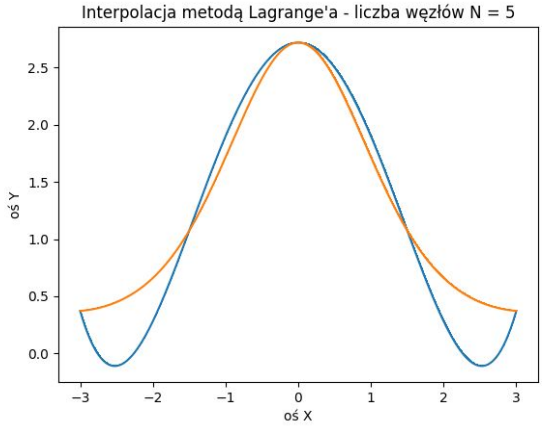
Porównanie wyników przedstawiłem na wykresach poniżej(kolorem żółtym jest zaznaczona funkcja interpolowana, a niebieskim - interpolująca).

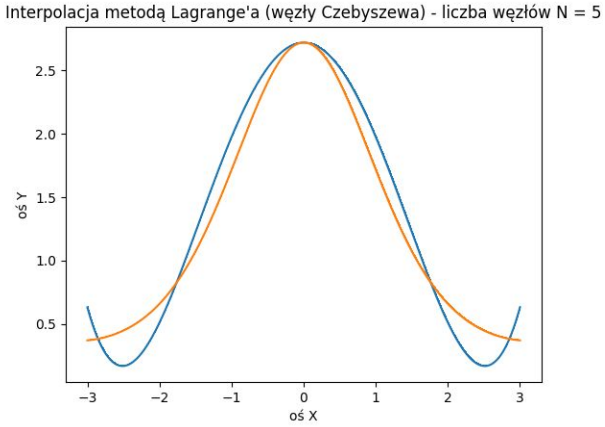


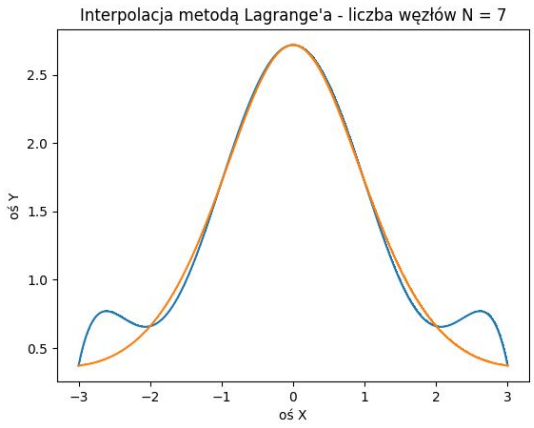


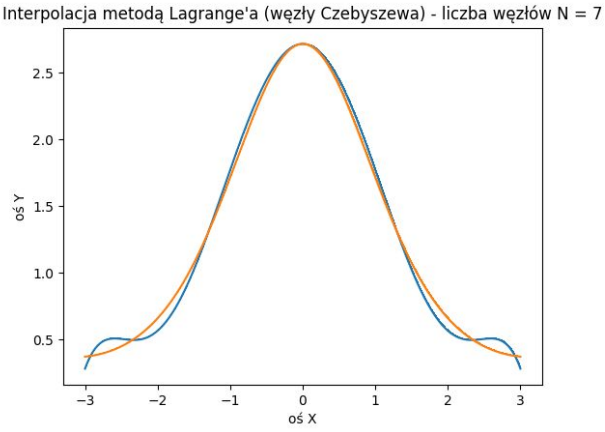

Interpolacja metodą Lagrange'a - liczba węzłów N = 3
oś Y
oś X

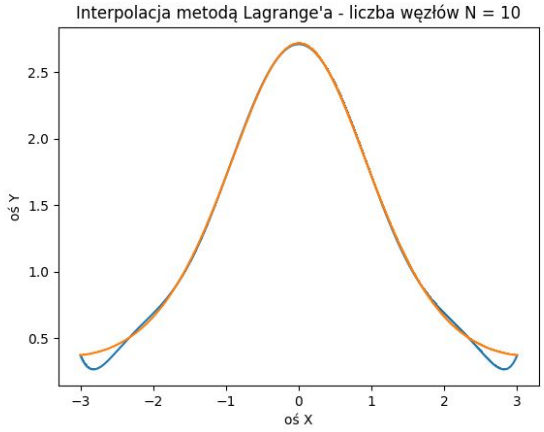

Interpolacja metodą Lagrange'a (węzły Czebyszewa) - liczba węzłów N = 3
oś Y
oś X

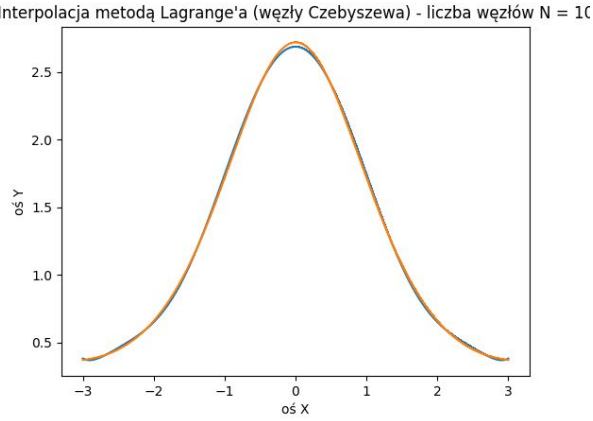

Interpolacja metodą Lagrange'a - liczba węzłów N = 5
oś Y
oś X

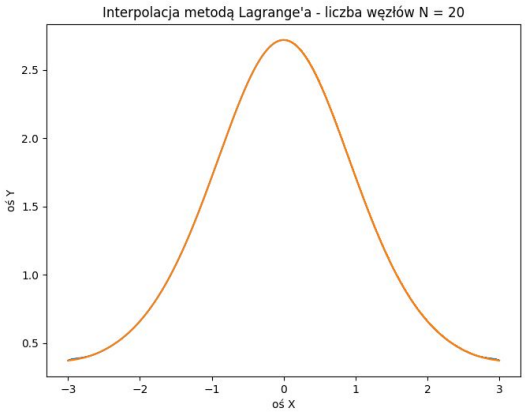

Interpolacja metodą Lagrange'a (węzły Czebyszewa) - liczba węzłów N = 5
oś Y
oś X

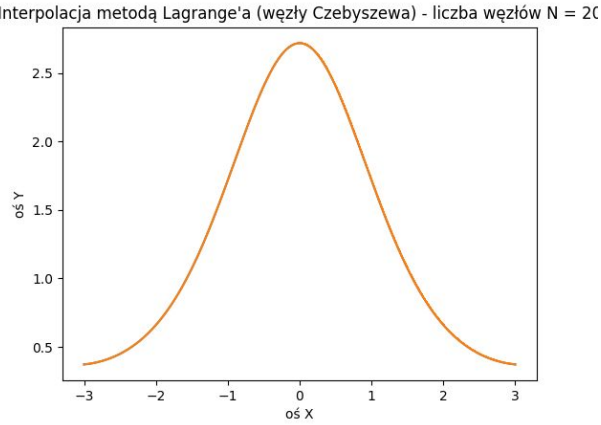

Interpolacja metodą Lagrange'a - liczba węzłów N = 7
oś Y
oś X


Interpolacja metodą Lagrange'a (węzły Czebyszewa) - liczba węzłów N = 7
oś Y
oś X









W porównywanych przypadkach interpolacja używając rozmieszczenia węzłów Czebyszewa jest dokładniejsza od równoodległego rozmieszczenia węzłów. Dla N = 20 ciężko zauważyć wielomian - w niemal każdym punkcie ma on porównywalną do funkcji wartość.

Analiza błędu w zależności od stopnia wielomianu interpolującego.

Błąd interpolacji w punkcie x to różnica wartości funkcji interpolowanej i interpolującej w tym punkcie. Zaimplementowałem funkcję, która sumuje wartość błędu na 10001 punktach przedziału <-3.0,3.0> :

```
double lagrangeError(int n, bool chebyshev){
    double error = 0.0;
    double interval = 6.0/10000;
    std::vector<point> results;
    if(!chebyshev) {
        results = interpolateLagrangeAllRange(n, 10001);
    }
    else{
        results = interpolateLagrangeUsingChebyshev(n, 10001);
    }
```

```
    for(int i = 0; i < 10001; i++){
        error += std::abs(function(i*interval-3.0) - results[i].y);
    }
    return error;
}
```

Zapętliłem funkcję dla wielomianów stopnia 2-99 dla obu rozmieszczeń węzłów. Częściowe wyniki przedstawia poniższa tabela:

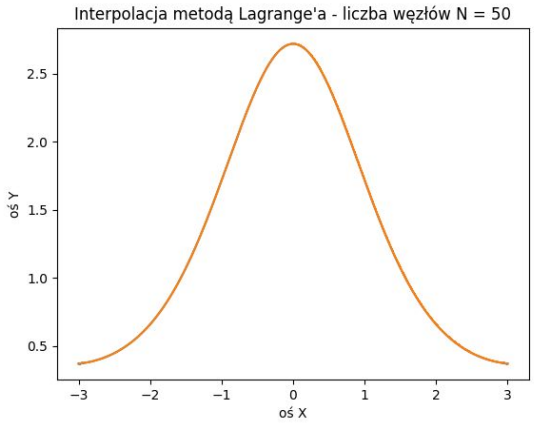
stopień wielomianu	węzły równoodległe	węzły Czebyszewa
7	943.00757	512.0375
10	234.72	135.47176
20	4.09755	0.13349
30	0.05115	0.0005
40	0.004	0
41	0.003	0
50	0.1001	0
75	1513298.126	0
90	32925050202	0
99	14270718995477	0

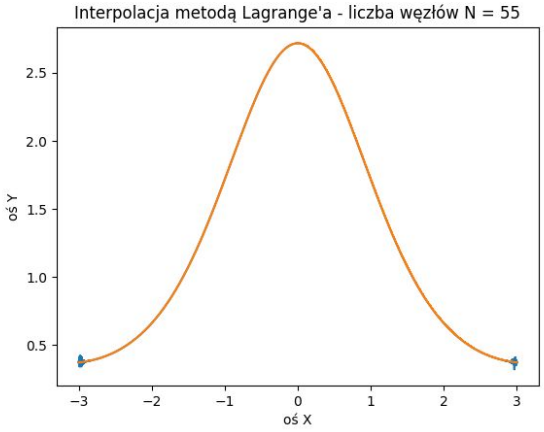
W przypadku rozmieszczenia równoodległego wartość błędu maleje, po czym zaczyna gwałtownie rosnąć. W przypadku węzłów Czebyszewa - wartość błędu maleje, wraz ze wzrostem stopnia wielomianu.

W metodzie korzystającej z równoodległego rozmieszczenia węzłów wartość błędu osiągnęła minimum dla wielomianu stopnia 41, dlatego uważam, że ten wielomian najlepiej przybliża zadaną funkcję.
W przypadku węzłów Czebyszewa - im wyższy stopień wielomianu, tym lepsza dokładność przybliżenia.

Stopień wielomianu, dla którego można zauważyć efekt Runge'go

Myślę, że właściwym przykładem jest wielomian stopnia 55. Wielomian stopnia 50 bardzo dobrze przybliża zadaną funkcję. Jeśli dodamy 5 węzłów - na krańcach przedziałów są widoczne różnice z przybliżaną funkcją. Takie zachowanie idealnie obrazuje efekt Runge'go - czyli pogorszenie jakości interpolacji, mimo zwiększenia jej węzłów.





3. Metoda Newtona

Polega na wyznaczaniu kolejnych ilorazów różnicowych oraz tworzeniu sum złożonych z iloczynu tych ilorazów i funkcji liniowych wg wzoru:

$$P_n(x) = f[x_0] + \sum_{k=1}^{n} f[x_0,\ x_1,\ \ldots,\ x_k](x - x_0)\cdots(x - x_{k-1})$$

Zaimplementowałem dwie funkcje pomocnicze:

Funkcja component - oblicza iloraz n funkcji liniowych

```
double component(int n, double x, std::vector<point> values){
   double result = 1;
   for(int i = 0; i < n; i++){
       result = result * (x - values[i].x);
   }
   return result;
}
```

Funkcja dividedDifferenceTable - wyznacza tablicę ilorazów różnicowych dla podanych danych początkowych.

```
std::vector<std::vector<double>> dividedDifferenceTable(std::vector<point>
values, int n){
   std::vector<std::vector<double>> result;
   std::vector<double> empty_vec;
   empty_vec.resize(n, -1.0);
   result.resize(n, empty_vec);

   for(int i = 0; i < n; i++){
       result[i][0] = values[i].y;
   }

   for(int i = 1; i < n; i++){
```

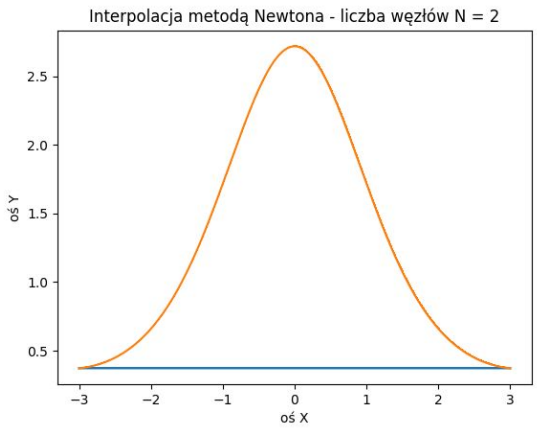
```
        for(int j = 0; j < n-i; j++){
            result[j][i] = 0.0;
            result[j][i] = (result[j][i-1] - result[j+1][i-1])/
                    (values[j].x - values[i+j].x);
        }
    }
    return result;
}
```

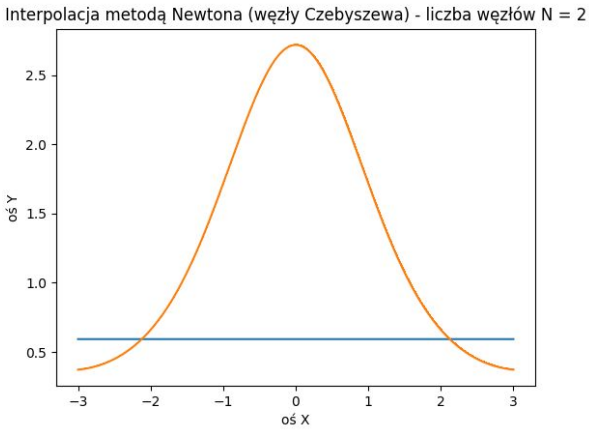
Na korzystając z dwóch powyższych metod napisałem funkcję, która wykonuje interpolację Newtona dla podanego punktu x.

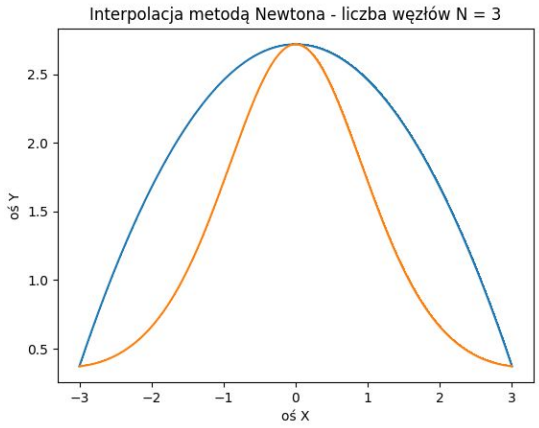
```
double interpolateNewton(std::vector<point> values, int n, double x){
    std::vector<std::vector<double>> y = dividedDifferenceTable(values, n);
    double sum = y[0][0];
    for(int i = 1; i < n; i++){
        sum = sum + (component(i, x, values) * y[0][i]);
    }
    return sum;
}
```

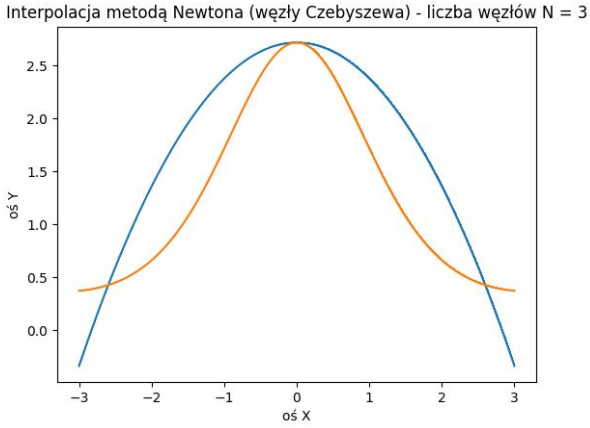
Porównanie przebiegu funkcji interpolującej otrzymanej dla różnego rozmieszczenia węzłów(równoodległych i Czebyszewa).

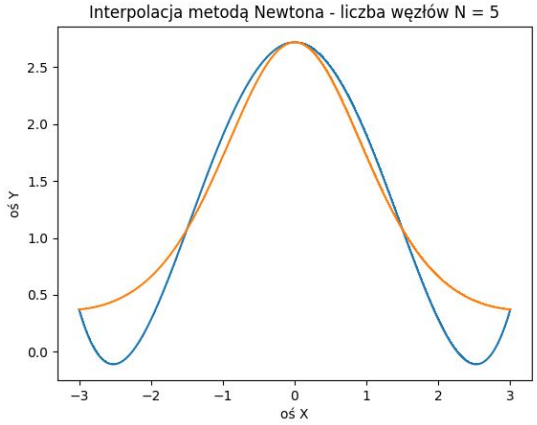
Analogicznie do metody Lagrange'a - utworzyłem dwie funkcje, które wyznaczają węzły interpolacji na dwa różne sposoby, po czym wywołują funkcję interpolateNewton w 10001 punktach przedziału <-3.0, 3.0>. Wyniki przedstawiam poniżej:

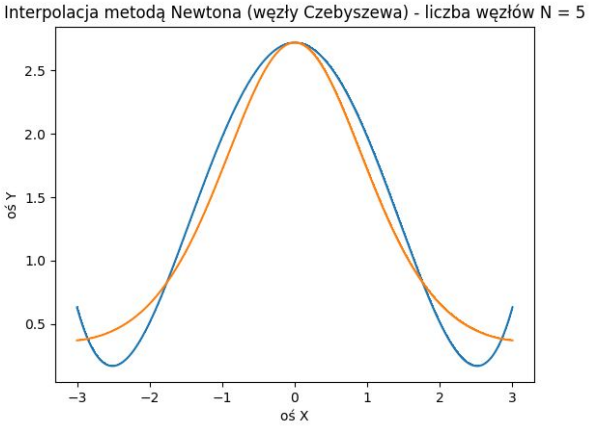


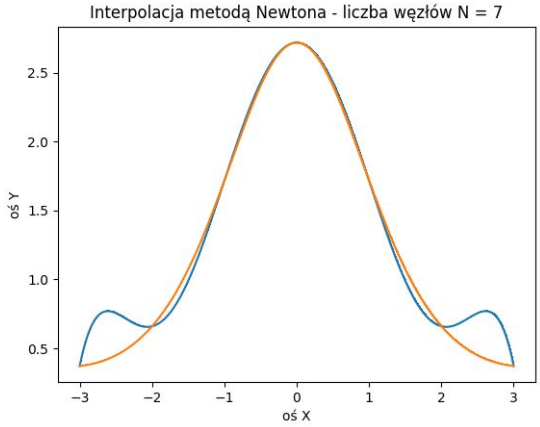


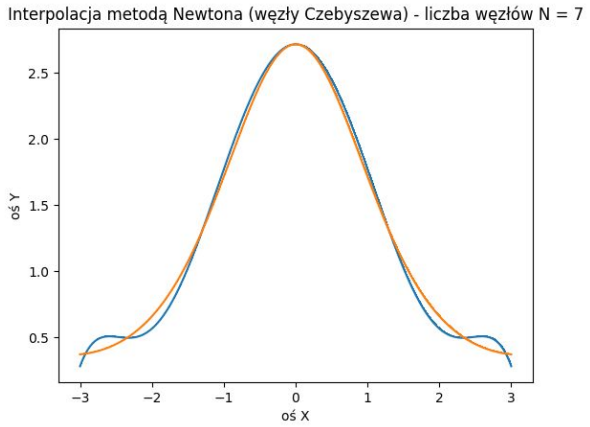

Interpolacja metodą Newtona - liczba węzłów N = 3
oś Y
oś X

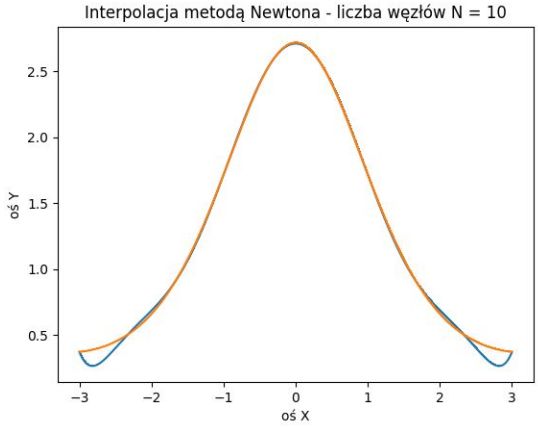

Interpolacja metodą Newtona (węzły Czebyszewa) - liczba węzłów N = 3
oś Y
oś X

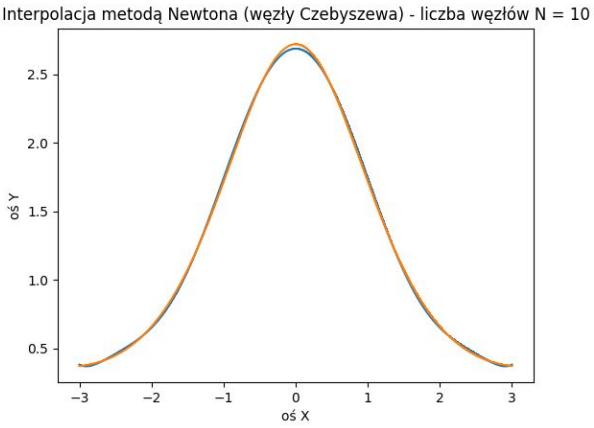

Interpolacja metodą Newtona - liczba węzłów N = 5
oś Y
oś X

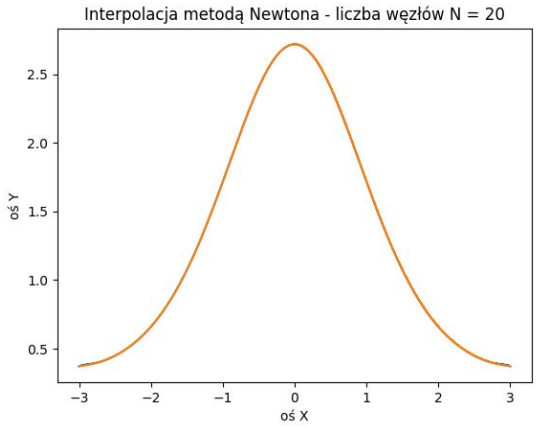

Interpolacja metodą Newtona (węzły Czebyszewa) - liczba węzłów N = 5
oś Y
oś X

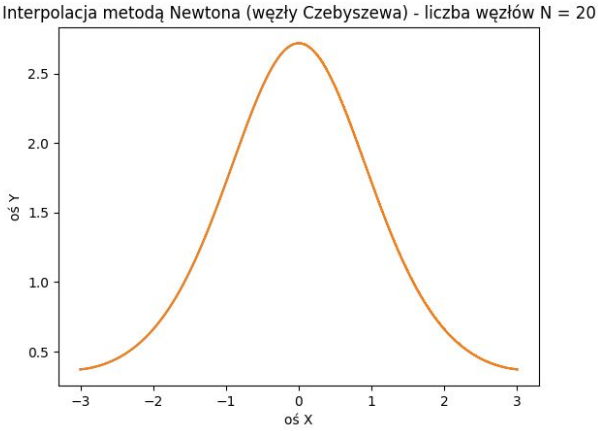

Interpolacja metodą Newtona - liczba węzłów N = 7
oś Y
oś X


Interpolacja metodą Newtona (węzły Czebyszewa) - liczba węzłów N = 7
oś Y
oś X









Podobnie jak w metodzie Lagrange'a - interpolacja korzystająca z rozmieszczenia węzłów Czebyszewa jest dokładniejsza od interpolacji na węzłach równoodległych.

Analiza błędu w zależności od stopnia wielomianu interpolującego.

Wyznaczyłem błąd w zależności od stopnia wielomianu w sposób analogiczny, do sposobu który opisałem w wyznaczaniu błędu dla metody Lagrange'a. Wyniki przedstawia poniższa tabela:

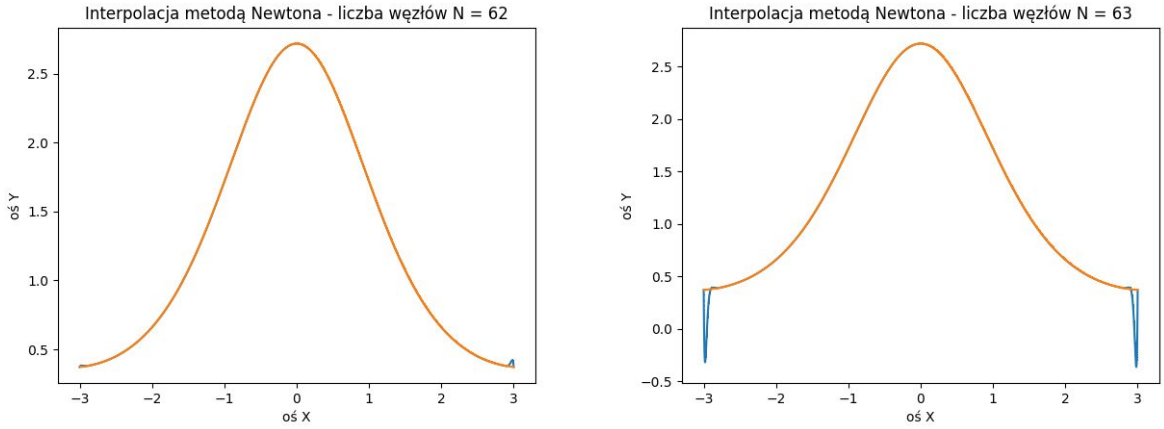
stopień wielomianu	węzły równoodległe	węzły Czebyszewa
7	943.00757	512.0375
10	234.72	135.47176
20	4.09755	0.13349
30	0.05115	0.00005
40	0.0004	0
43	0.00005	0
50	0.00403	0
60	1.37128	0.00658
70	549.69151	92.58153
80	1376210.391	67025004.4
90	101209339876	2367058999995

Tym razem wielomianem wyznaczonym na węzłach równoodległych, który najlepiej przybliża zadaną funkcję okazał się wielomian o stopniu 43. Podobnie jak we wcześniejszej metodzie wartość błędu maleje wraz ze wzrostem stopnia wielomianu do pewnego momentu, a potem zaczyna rosnąć. Co ciekawe, przy zwiększaniu stopnia wielomianu powyżej 50 można zaobserwować zmniejszanie się błędu w pojedynczych przypadkach(np wielomian stopnia 63 jest obarczony błędem 107,99; natomiast wielomian stopnia 64 - 30.86). Jest to odmienne zachowanie względem błędu wyznaczonego dla metody Lagrange'a - gdzie błąd od stopnia 41 konsekwentnie wzrastał. Warto dodać, że w badanych przypadkach spadek wartości błędu nie był większy, niż jeden rząd wielkości.

W przypadku błędu wielomianu korzystającego z węzłów Czebyszewa błąd maleje praktycznie do 0(stopień 40), po czym zaczyna gwałtownie wzrastać(stopień 60). Również w tym przypadku jest to odmienne zachowanie względem błędu metody Lagrange'a - gdzie wraz ze wzrostem stopnia wielomianu korzystającego z rozmieszczenia węzłów Czebyszewa malała wartość błędu.

Stopień wielomianu, dla którego można zauważyć efekt Runge'go

Dla tej metody wybrałem wielomiany stopnia 62 i 63. Wielomian stopnia 62 niemal idealnie pasuje do zadanej funkcji. Dodanie zaledwie jednego węzła spowodowało wystąpienie ogromnego błędu na krańcach przedziału:



Powyższy przypadek jest odpowiednim przykładem pogorszenia dopasowania wielomianu interpolującego pomimo zwiększenia ilości węzłów.

4. Interpolacja Hermite'a

Interpolacja metodą Hermite'a jest bardzo podobna do interpolacji metodą Newtona. W tej metodzie również tworzymy tablicę ilorazów różnicowych. Istotną różnicą jest używanie kolejnych pochodnych funkcji w celu zwiększenia precyzji przybliżenia. Do tablicy dodajemy każdy węzeł interpolacji k+1 razy, gdzie k to stopień najwyższej pochodnej, którą będziemy używać. Następnie budujemy tablicę ilorazów w sposób analogiczny do metody Newtona.

Poniższa implementacja stosuje metodę Hermite'a dla jednej pochodnej.
Zaimplementowałem funkcję, która zwraca wartość pochodnej zadanej funkcje w podanym przez argument punkcie:

```
double derivative(double x){
   return -sin(x) * exp(cos(x));
}
```

Funkcja która wyznacza tablicę ilorazów różnicowych:

```
std::vector<std::vector<double>> dividedDifferenceTableHermite
(std::vector<point> values, int n){
   std::vector<std::vector<double>> result;
   std::vector<double> empty_vec;
   empty_vec.resize(2*n, -1.0);
   result.resize(2*n, empty_vec);

   std::vector<point> values2;

   for(int i = 0; i < n; i++){
       values2.push_back(values[i]);
       values2.push_back(values[i]);
   }

   for(int i = 0; i < 2*n; i++){
       for(int j = 0; j < i+1; j++){
           if(j == 0) result[i][j] = function(values2[i].x);
           else if(j == 1 && i%2 == 1)
                   result[i][j] = derivative(values2[i].x);
           else{
               result[i][j] = result[i][j-1] - result[i-1][j-1];
               result[i][j] =
                   result[i][j] / (values2[i].x - values2[i-j].x);
           }
       }
   }

   return result;
}
```

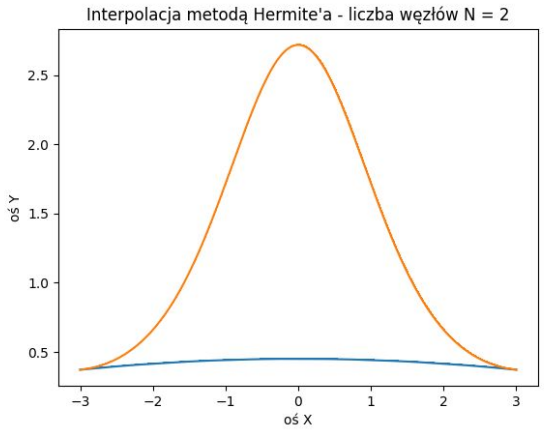
Funkcja wyznaczająca wartość funkcji interpolującej w punkcie x:

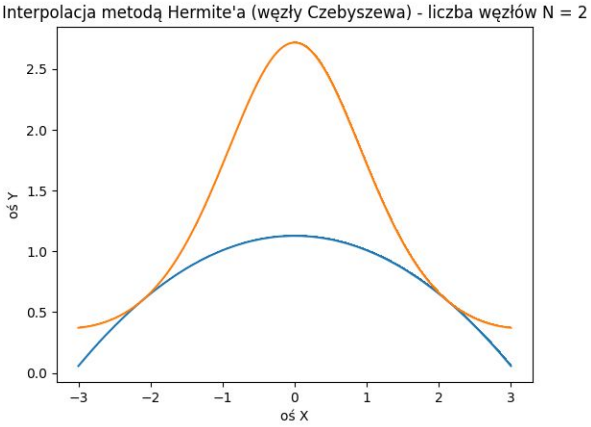
```
double interpolateHermite(std::vector<point> values, int n, double x){
   std::vector<point> values2;

   for(int i = 0; i < n; i++){
       values2.push_back(values[i]);
       values2.push_back(values[i]);
   }
```

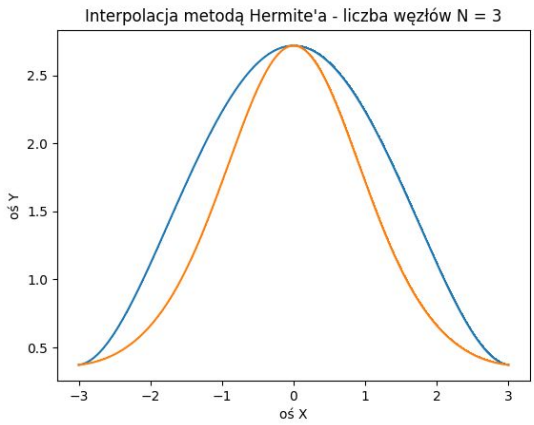
```
    std::vector<std::vector<double>> y =
dividedDifferenceTableHermite(values, n);
    double result = 0.0;
    double component = 1.0;
    for(int i = 0; i < 2*n; i++){
        result = result + y[i][i] * component;
        component = component * (x - values2[i].x);
    }
    return result;
}
```

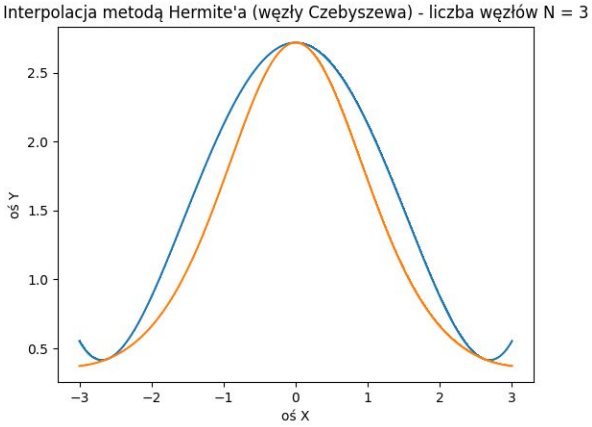
Porównanie przebiegu funkcji interpolującej otrzymanej dla różnego rozmieszczenia węzłów(równoodległych i Czebyszewa).

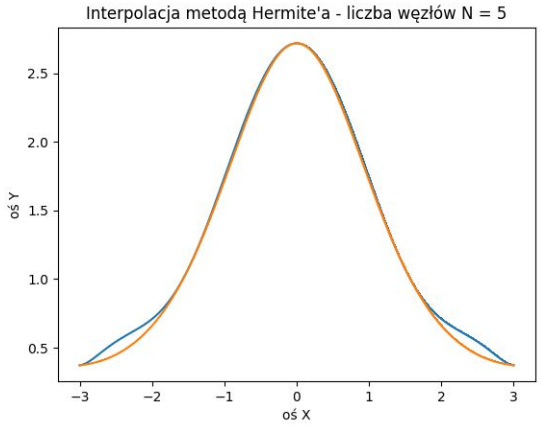
Analogicznie poprzednich metod - utworzyłem dwie funkcje, które wyznaczają węzły interpolacji na dwa różne sposoby, po czym wywołują funkcję interpolateHermite w 10001 punktach przedziału <-3.0, 3.0>. Wyniki przedstawiam poniżej:

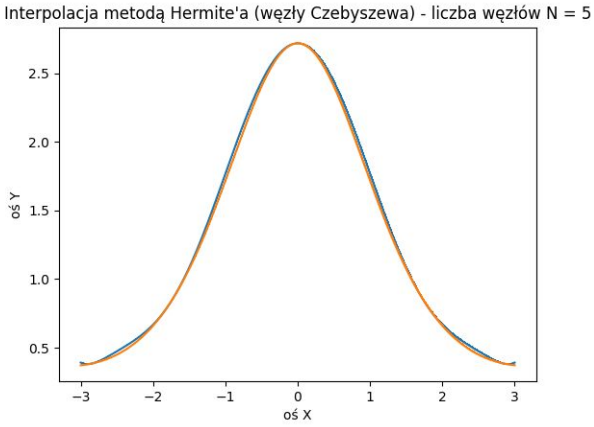


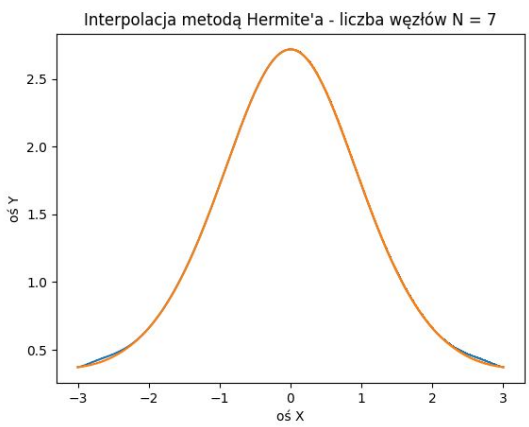


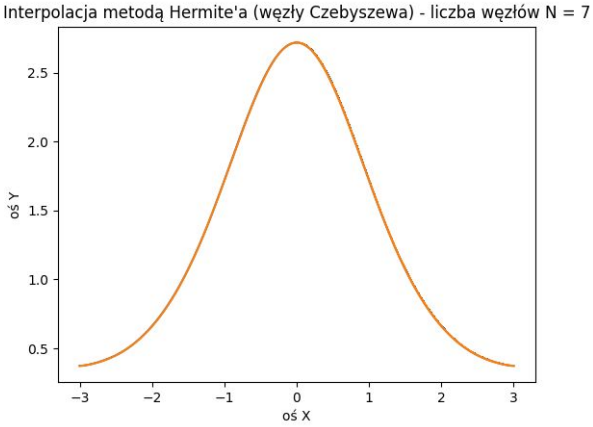


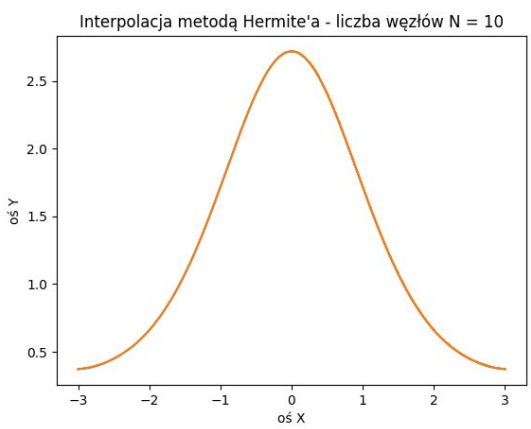


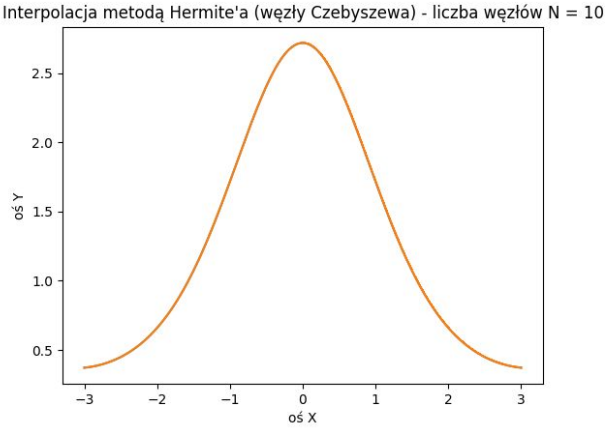

Interpolacja metodą Hermite'a - liczba węzłów N = 5
oś Y
oś X

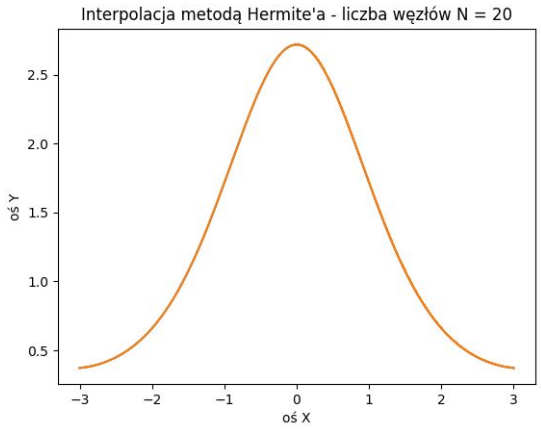

Interpolacja metodą Hermite'a (węzły Czebyszewa) - liczba węzłów N = 5
oś Y
oś X

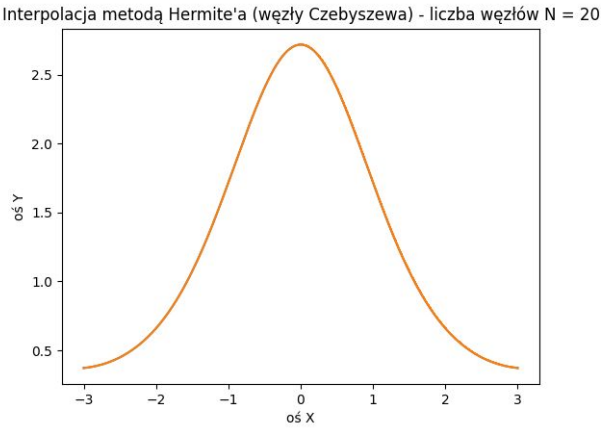

Interpolacja metodą Hermite'a - liczba węzłów N = 7
oś Y
oś X


Interpolacja metodą Hermite'a (węzły Czebyszewa) - liczba węzłów N = 7
oś Y
oś X


Interpolacja metodą Hermite'a - liczba węzłów N = 10
oś Y
oś X


Interpolacja metodą Hermite'a (węzły Czebyszewa) - liczba węzłów N = 10
oś Y
oś X





Podobnie jak w poprzednich metodach - dopasowanie korzystając z rozmieszczenia węzłów Czebyszewa jest dokładniejsze od dopasowania na węzłach równoodległych.
Na podstawie wykresów można stwierdzić, że metoda Hermite'a jest dużo bardziej dokładna od metod: Lagrange'a i Newtona. Interpolacja na 10 węzłach jest obarczona błędem niemożliwym do uchwycenia przez ludzkie oko.

Analiza błędu w zależności od stopnia wielomianu interpolującego.

Wyznaczyłem błąd metody Hermite'a w sposób analogiczny do poprzednich metod. Sumę wartości bezwzględnych pomiędzy funkcją interpolowaną, a interpolującą na 10001 punktach przedstawia poniższa tabela.

stopień wielomianu	węzły równoodległe	węzły Czebyszewa
3	3252.41581	2121.37546
5	325.06603	212.03748
7	41.84849	15.63678
10	2.53203	0.20945
20	0.00014	0
22	0.00002	0
30	0.01996	0.00074
33	0.47381	0.22117
34	35.10614	4.07704
38	3877.44056	51932.22959
40	48275.54233	5083211.632
45	7100375831	83739950596

W przypadku węzłów równoodległych - wartość błędu była minimalna dla N = 22. Wartość błędu błyskawicznie maleje - bez względu na sposób rozmieszczenia węzłów. Następnie, mniej więcej w okolicy N = 25 błąd zwiększa się drastycznie szybko. Dla N > 40 - po dodaniu dwóch węzłów błąd zwiększa się o jeden rząd wielkości. Wielomianem wyznaczonym przez interpolację na węzłach równoodległych, który najlepiej przybliża zadaną funkcję jest wielomian stopnia 22.

Stopień wielomianu, dla którego można zauważyć efekt Runge'go

Dla tej metody wybrałem wielomiany stopnia 33 i 35. Na podstawie obliczeń z poprzedniego podpunktu - wielomian stopnia 33 jest obarczony znikomym błędem. Dodanie dwóch węzłów interpolacyjnych spowodowało wystąpienie niedokładnego dopasowania na krańcach przedziałów.