Justyna Janiak 180566

Łukasz Nizik 180647

Rok akademicki 2013/14

Środa, 10:30

METODY NUMERYCZNE – LABORATORIUM

Zadanie 3 – Interpolacja Newtona na węzłach Czebyszewa

Opis metody

Celem interpolacji jest wyznaczenie przybliżonych wartości funkcji f(x) w punktach nie będących węzłami tej funkcji. Aby je wyznaczyć poszukujemy funkcji interpolującej, która w węzłach przyjmuje wartości równe wartościom funkcji f(x).

Węzły Czebyszewa wyznaczamy ze wzoru: $x_n = \cos\left(\frac{2m+1}{2n+1}\pi\right), m = 0,1,\dots,n$. Należą one do przedziału [−1; 1] i aby uzyskać węzły na przedziale [a; b] należy dokonać zamiany zmiennych:

$$y_n = \frac{1}{2}[(b-a)x_n + (a+b)], m = 0,1,\dots,n.$$

Interpolacja dla węzłów o różnych odległościach:
Wzór pozwalający wyznaczyć wielomian interpolacyjny $W_n(x)$ stopnia co najwyżej n funkcji $f(x)$, następującej postaci:

$$W_n(x) = f_0(x_0) + f_1(x_1)(x-x_0) + f_2(x_2)(x-x_0)(x-x_1) + \dots + f_n(x_n)(x-x_0)(x-x_1)\dots(x-x_{n-1})$$

gdzie: $f_0(x) = f(x)$, $f_j(x) = \frac{f_{j-1}(x) - f_{j-1}(x_{j-1})}{x - x_{j-1}}$, dla j= 1, 2, ..., n, gdzie $x_0, x_1, x_2, \dots, x_n$ są węzłami interpolacji.

Węzły równoodległe:

Wzór interpolacyjny Newtona:

$$W_n(t) = \sum_{k=0}^{n} \frac{\Delta^k y_0}{k!} a_k(t)$$

gdzie:

$$a_0 = 1$$

$$a_k(t) = \prod_{m=0}^{k} t - m\,, dla\ k > 0$$

Różnica progresywna funkcji $y = f(x)$:

$$\Delta^k y_0 = \sum_{i=0}^{k} (-1)^{k-i} \binom{k}{i} y_i$$

$$t = \frac{x - x_0}{h}$$

gdzie:

h – odległość od poprzedniego węzła

Kolejność wykonywanych kroków przez program

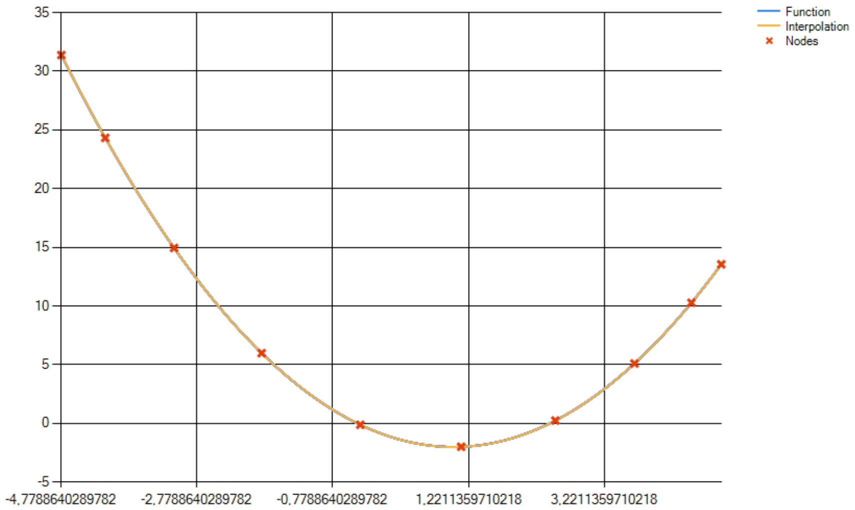
- Wybieramy funkcję
- Podajemy do programu węzły interpolacyjne (wg wyboru : równoodległe bądź Czebyszewa)
- Podajemy skok argumentu interpolacji i dziedzinę
- Program wylicza węzły
- Następnie przekazuje węzły do funkcji interpolującej

Interpolacja działa tak, że najpierw wyznaczamy ilorazy różnicowe korzystając ze schematu ilorazów następnie podstawiamy otrzymane wartości do wzoru na wielomian interpolacyjny, funkcja która to robi zwraca List<double> . Kolejne elementy listy to wartości współczynników przy x, element 0 to współczynnik przy x o najwyższej potędze.

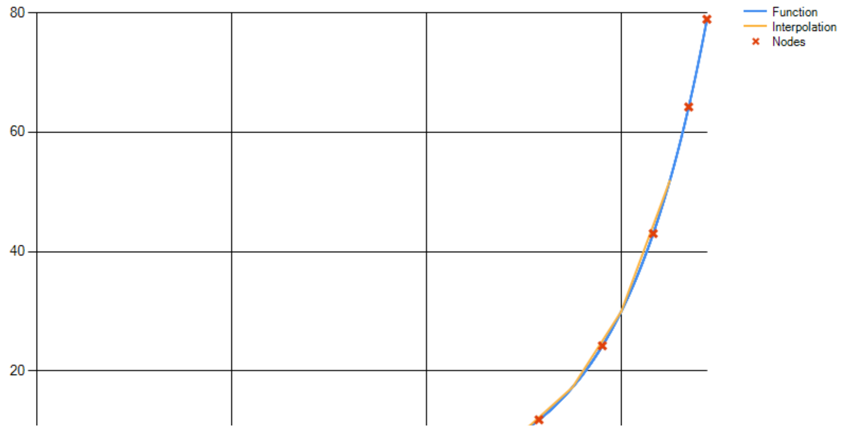
- Następnie używa schematu Hornera aby znaleźć wartość dla zadanego x (x to argument dla którego chcemy interpolować)
- Wartość zwrócona przez schemat Hornera to y

Wyniki

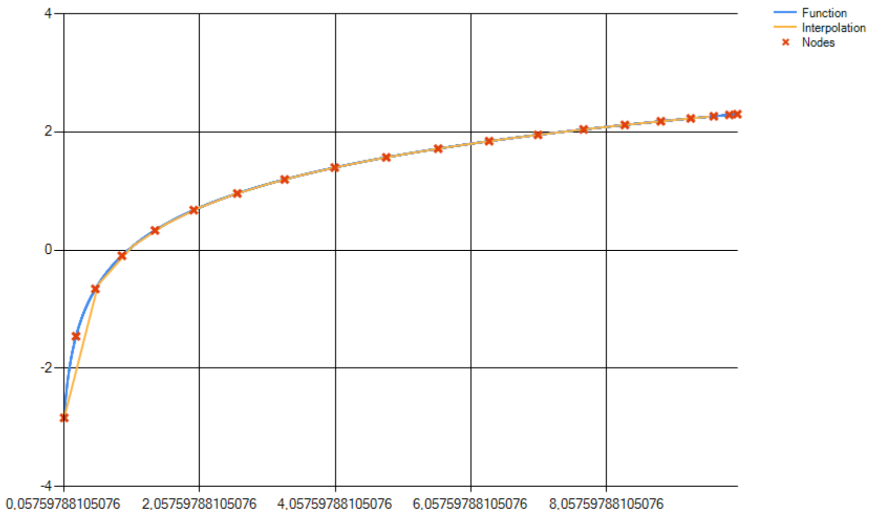
Funkcja	X^2-2x-1
Węzły	Czebyszew
Zakres	-5,5
Skok	0,1
Liczba węzłów	10



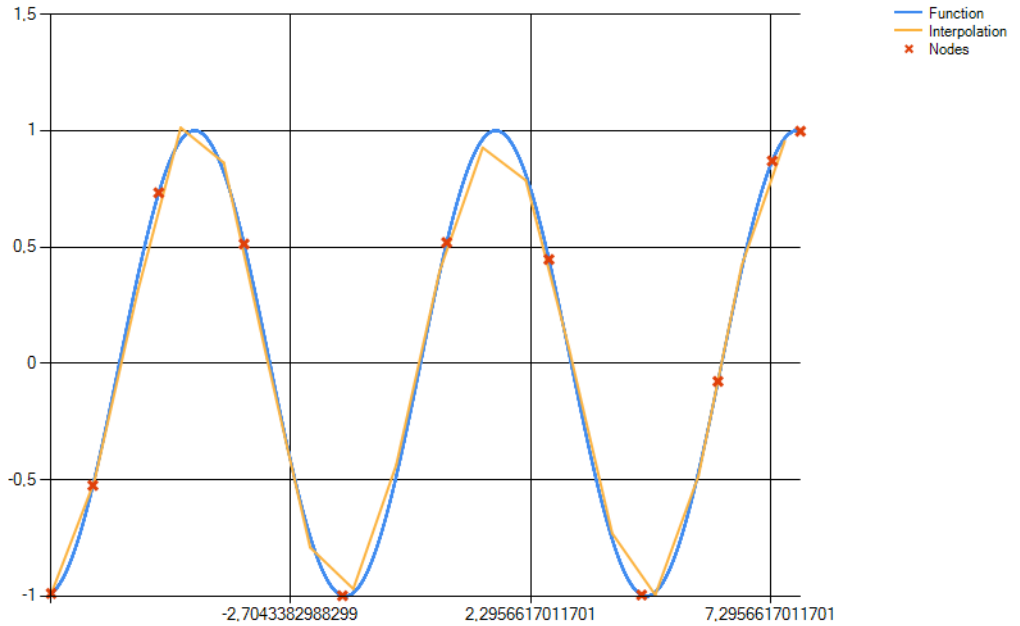
Funkcja	3^x
Węzły	Czebyszew
Zakres	-3,4
Skok	0,5
Liczba węzłów	13



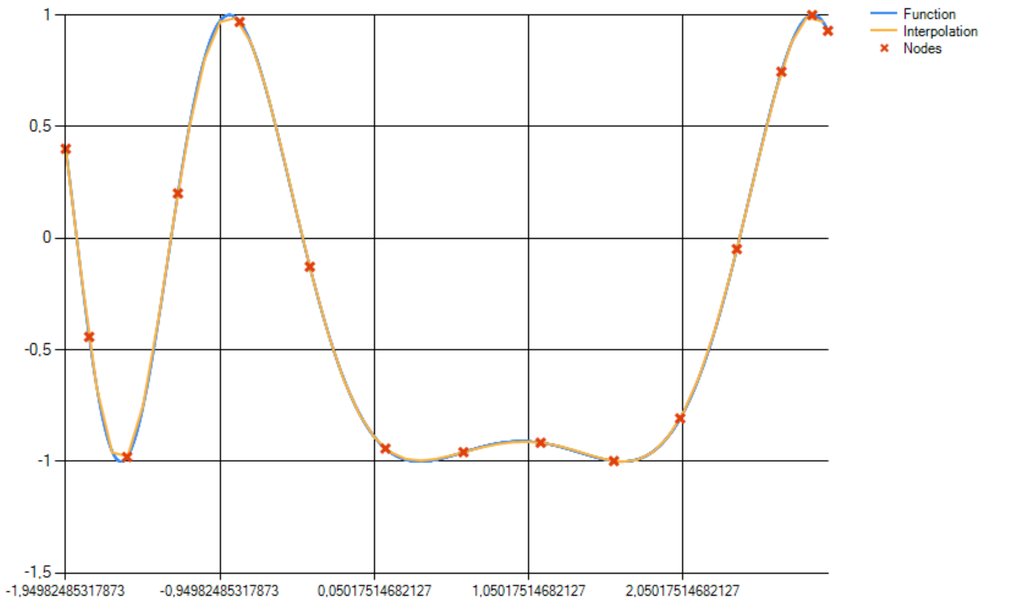
Funkcja	ln(x)
Węzły	Czebyszew
Zakres	0,10
Skok	0,5
Liczba węzłów	20



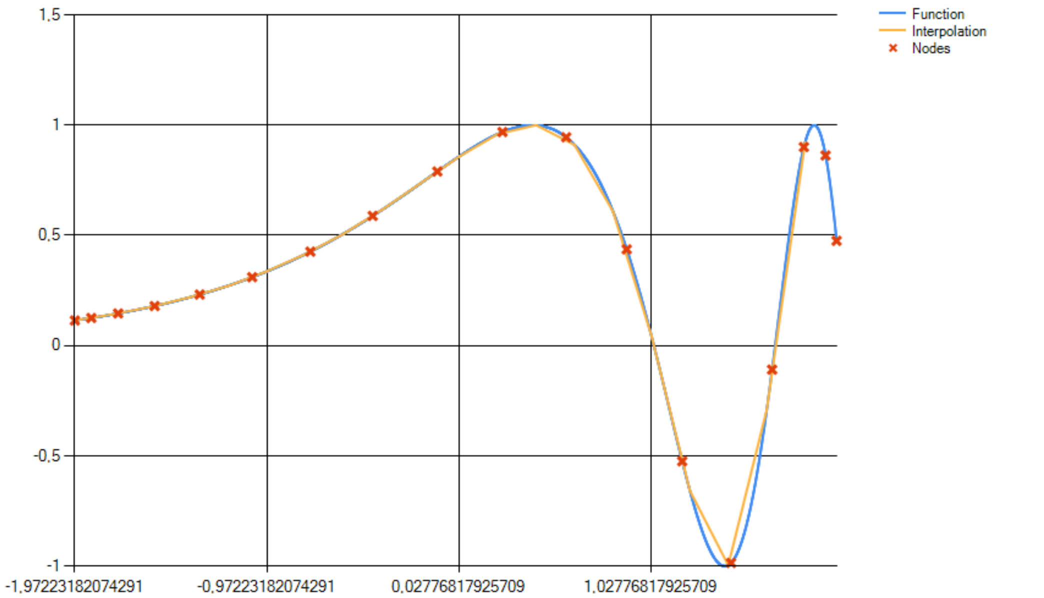
Funkcja	sin(x)
Węzły	Czebyszew
Zakres	-8,8
Skok	0,9
Liczba węzłów	11



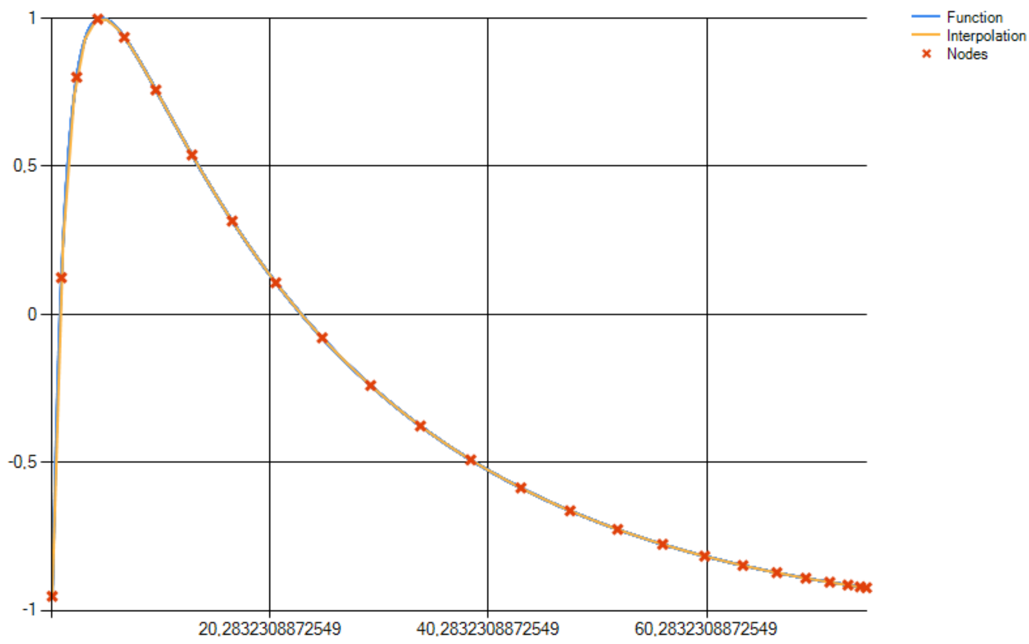
Funkcja	sin(x^2-2x-1)
Węzły	Czebyszew
Zakres	-2,3
Skok	0,1
Liczba węzłów	15



Funkcja	sin(3^x)
Węzły	Czebyszew
Zakres	-2,2
Skok	0,2
Liczba węzłów	18



Funkcja	sin(ln(x))
Węzły	Czebyszew
Zakres	0,75
Skok	01
Liczba węzłów	25



Wnioski

Węzły Czebyszewa są najlepszym wyborem do konstruowania wielomianów interpolujących. Wynika to z faktu, iż zagęszczają sie one na krańcach przedziału. Powoduje to małe oscylacje (miejsca zerowe są bardzo blisko siebie) i styczne wielomianu na krańcach są zbliżone do stycznych funkcji interpolowanych.

Do rysowania wykresów najlepiej jest wybrać większą ilość węzłów (ilość stosunkowo większą od długości przedziału, na którym dokonujemy interpolacji). Jednak zbyt duża ich ilość może mieć wpływ na otrzymanie nieprawidłowych wyników.
Wszelkie rozbieżności wynikać mogą ze skończonej dokładności reprezentacji liczb rzeczywistych w pamięci komputera.

Listing kodu

```
namespace Interpolation
{
    public struct Point
    {
        public double x;
        public double y;
        public Point(double x, double y)
        {
            this.x = x;
            this.y = y;
        }

    }

    class NewtonChebyshevInterpolation
    {
        public static double HornerScheme(List<double> tabA, double x)
        {
            double result = tabA[0];
            for (int i = 1; i < tabA.Count(); i += 1)
            {
                result = result * x + tabA[i];
            }
            return result;
        }

        double DifferentialQuotient(Point p1, Point p2)
```

```
        {
            return (p2.y - p1.y) / (p2.x - p1.x);
        }

        List<double> PolynomialsProduct(List<double> A, List<double> B)
        {
                List<double> result = new List<double>();
                for (uint i = 0; i < (A.Count() + B.Count() - 1); i += 1) result.Add(0);

                for (int i = 0; i < A.Count(); i += 1)
                {
                        for (int j = 0; j < B.Count(); j += 1)
                        {
                                result[i + j] += A[i] * B[j];
                        }
                }
                return result;
        }

        List<double> AddPolynomials (List<double> A, List<double> B)
        {
            List<double> result, tmpA, tmpB;

            if (A.Count() > B.Count())
            {
                tmpA = A;
                tmpB = B;
            }
            else
            {
                tmpA = B;
                tmpB = A;
            }

            result = new List<double>(tmpA);
            result.Reverse();
            for (int i = 0; i < tmpB.Count(); i++)
            {
                result[i] += tmpB[tmpB.Count()-i-1];
            }

            result.Reverse();
            return result;
        }

        public List<double> GetInterpolationPolynomial(List<Point> nodes)
        {
            List<Point> result = new List<Point>();
            int degree = nodes.Count();

                List<List<double>> differentialQuotients = new List<List<double>>();
            for (int i = 0; i < degree; i++ )
            {
                differentialQuotients.Add(new List<double>());
            }

            foreach (Point node in nodes)
            {
                differentialQuotients[0].Add(node.y);
            }

                for (int i = 1; i < degree; i += 1)
                {
                        for (int j = 0; j < degree - i; j += 1)
                        {
                                differentialQuotients[i].Add(DifferentialQuotient(new Point(nodes[j].x,
differentialQuotients[i - 1][j]), new Point(nodes[j + i].x, differentialQuotients[i - 1][j + 1])));
                        }
                }
```

```
            List<double> polynomial = new List<double>();
            for (int i = 0; i < degree; i += 1) polynomial.Add(0);

            for (int i = 0; i < degree; i += 1)
            {
                List<double> fi = new List<double>();
                fi.Add(differentialQuotients[i][0]);
                for (int j = 0; j < i; j += 1)
                {
                    List<double> tmp = new List<double>();
                    tmp.Add(1);
                    tmp.Add(-nodes[j].x);
                    fi = PolynomialsProduct(fi, tmp);
                }

                polynomial = AddPolynomials(polynomial, fi);
            }

            return polynomial;
        }

        public List<Point> Interpolate (List<Point> nodes, List<double> domain)
        {
            List<Point> result = new List<Point>();
            List<double> interpolationPoly = GetInterpolationPolynomial(nodes);
            foreach (double x in domain)
            {
                result.Add(new Point(x, HornerScheme(interpolationPoly, x)));
            }

            return result;
        }
    }

    class NewtonInterpolation
    {


        double GetVarT(double x0, double distance, double x)
        {
            double result = (x - x0) / (distance);
            return result;
        }

        public double Interpolate(List<Point> nodes, double x)
        {
            DateTime time = DateTime.Now;

            double result = 0;
            for (int k = 0; k < nodes.Count(); k++)
            {
                double h = nodes[1].x - nodes[0].x;
                double t = (x - nodes[0].x) / (h);

                double tmpDiff = ProgressiveDifference(k, nodes) * CountFactor(k, t) /
factorial((UInt64)k);
                result += tmpDiff;
            }

            return result;
        }

        double CountFactor(double k, double t)
        {
            double result = 1;
            if (k >= 0)
            {
                for (int m = 0; m < k; m++)
```

```
                {
                    result *= t - m;
                }
            }

            return result;
        }

        double ProgressiveDifference(int k, List<Point> nodes)
        {
            double result = 0;

            for (int i = 0; i <= k; i++)
            {
                result += (double)Math.Pow(-1, k - i) * NewtonSymbol((UInt64)k, (UInt64)i) *
nodes[i].y;
            }

            return result;
        }

        int NewtonSymbol(UInt64 n, UInt64 k)
        {
            UInt64 result = factorial(n) / (factorial(k) * factorial(n - k));
            return (int)result;
        }

        public static UInt64 factorial(UInt64 x)
        {
            UInt64 result = 1;
            if (x <= 0)
                return result;

            for (UInt64 i = 1; i <= x; i++)
            {
                result *= i;
            }

            return result;
        }
    }
}
```