Metody Obliczeniowe w Nauce i Technice

Laboratorium 4 – raport

Mateusz Kocot

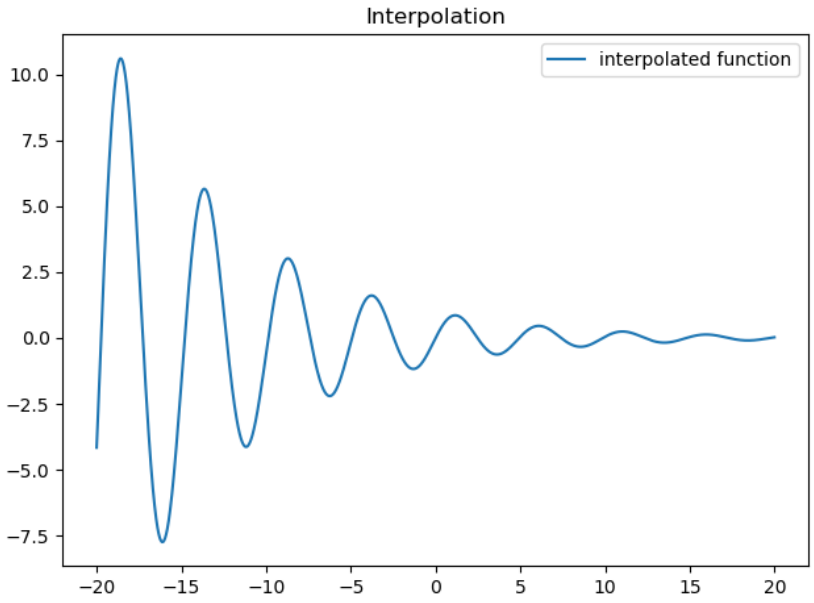
1. Funkcja testowa

Funkcją, którą wykorzystam do testowania interpolacji jest:

$$f(x) = \sin\frac{4x}{\pi}e^{-\frac{0.4x}{\pi}}.$$

```
inline double f(double x)
{
    return sin(k * x / M_PI) * exp(-m * x / M_PI);
}
```

Funkcję tę będę testował na przedziale $[-20, 20]$. Poniżej przedstawiono jej przebieg w tym przedziale. W celu sporządzenia wykresu obliczono wartość funkcji w 4001 punktach oddalonych od siebie o 0.01. Następnie punkty te wyświetlono na wykresie z wykorzystaniem pakietu Matplotlib w języku Python.



Wykres wybranej funkcji testowej w przedziale $[-20,20]$.

W celu zwiększenia czytelności zaimplementowano klasę $Point$ reprezentującą punkt.

```
class Point
{
private:
    const double x;
    const double y;
```

```
public:
    Point(double x, double y) : x(x), y(y) {}

    double getX() const;
    double getY() const;

    friend std::ostream& operator<<(std::ostream &out, const Point &point);
};
```

2. Interpolacja Lagrange'a

Zaimplementowano metodę interpolacji Lagrange'a.

```
std::vector<Point> lagrangeInterpolation(const std::vector<Point> &nodes,
const std::vector<double> &pointsX)
{
    int n = nodes.size();
    std::vector<Point> points;
    for (int i = 0; i < pointsX.size(); i++)
    {
        double x = pointsX[i];
        double y = 0;
        for (int k = 0; k < n; k++)
        {
            double xk = nodes[k].getX();
            double d = 1, m = 1;
            for (int j = 0; j < n; j++)
            {
                if (j == k)
                    continue;
                d *= (x - nodes[j].getX());
                m *= (xk - nodes[j].getX());
            }
            double Lk = d / m;
            y += (Lk * nodes[k].getY());
        }
        points.push_back(Point(x, y));
    }
    return points;
}
```

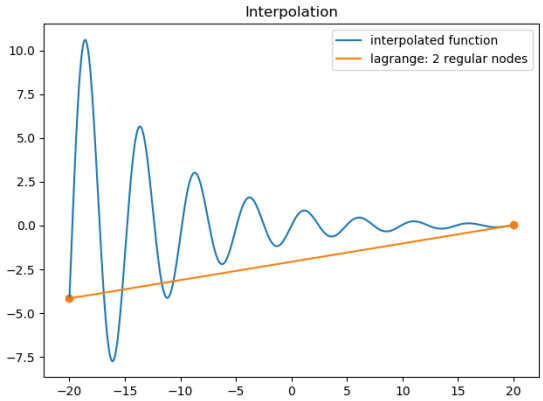
Funkcja ta przyjmuje tablicę węzłów $nodes$ (współrzędne x i y) oraz tablicę argumentów $pointsX$, dla których należy obliczyć wartości. Funkcja zwraca tablicę obliczonych punktów.

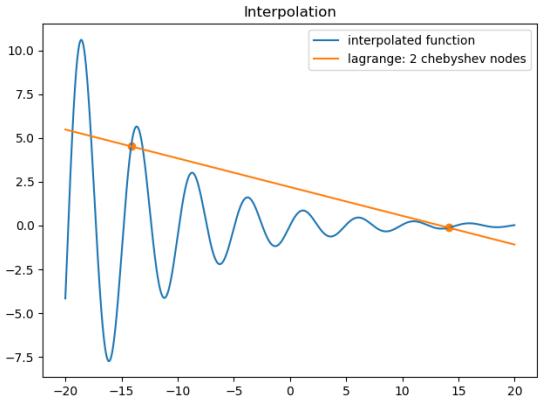
Ważne tutaj jest rozmieszczenie węzłów – równoodległe albo Czebyszewa. Rozmieszczenie Czebyszewa uzyskano za pomocą funkcji:

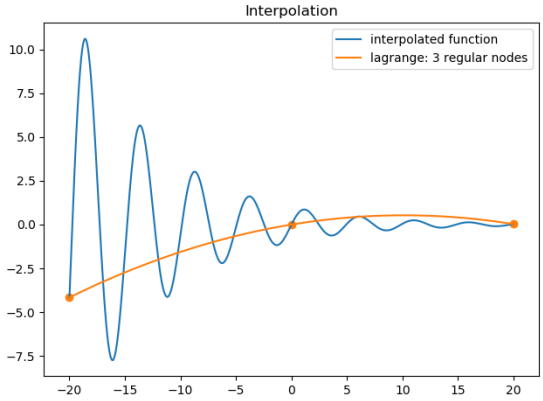
```
std::vector<Point> getChebyshevPoints(double f(double), double start,
```

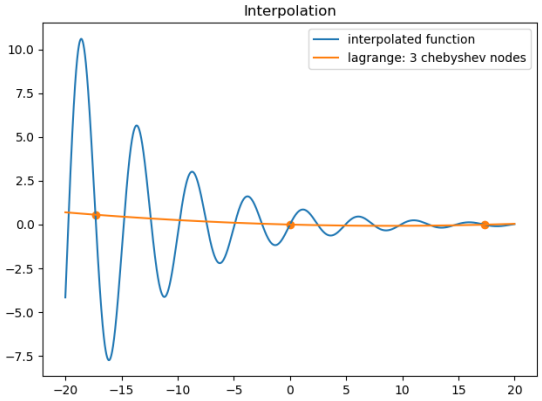
```
      double end, int count)
{
    std::vector<Point> points;
    for (int k = 0; k < count; k++)
    {
        double x = cos(((2 * k + 1) * M_PI) / (count * 2));
        x = ((end - start) / 2) * x + (start + end) / 2;    // Scaling to
                                                            // proper range
        points.push_back(Point(x, f(x)));
    }
    return points;
}
```

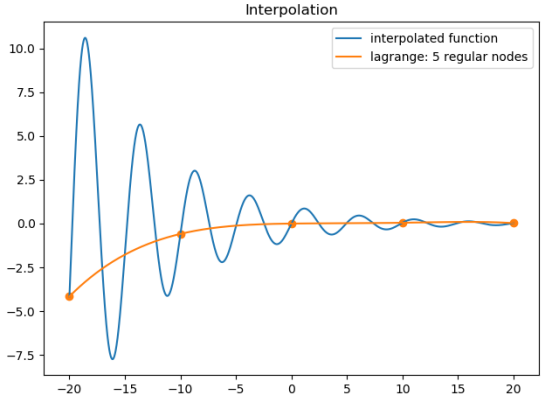
Poniżej zamieszczono przebiegi wielomianów obliczonych dla różnej liczby oraz rozmieszczenia węzłów.

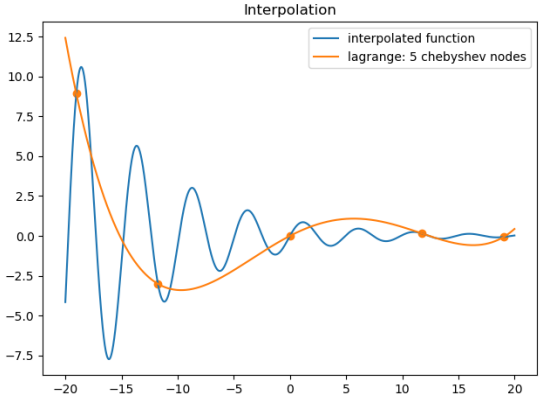


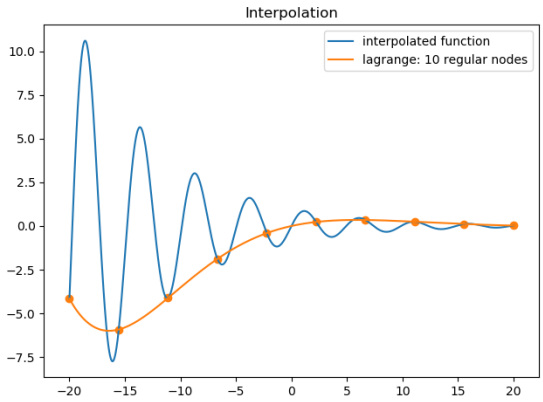


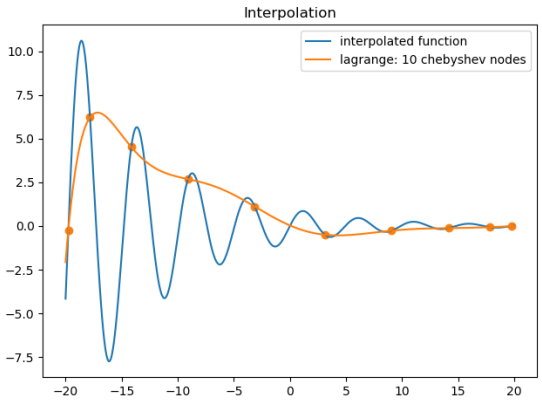


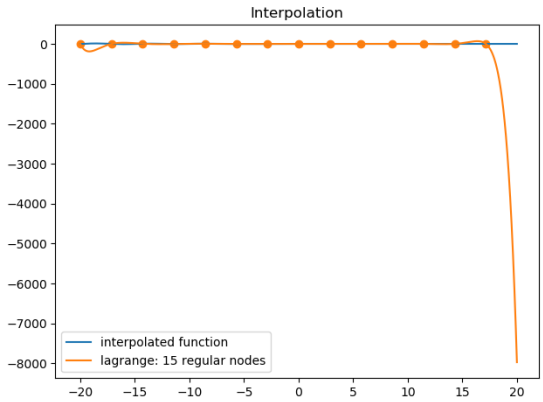


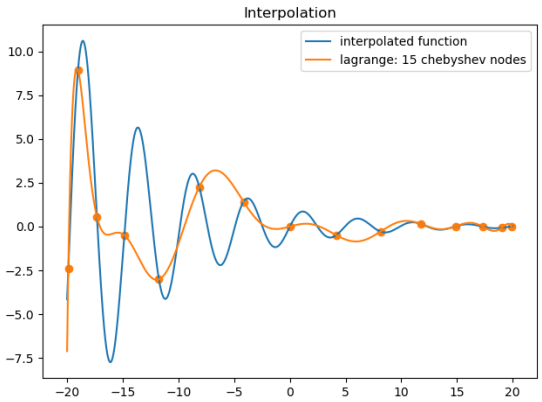


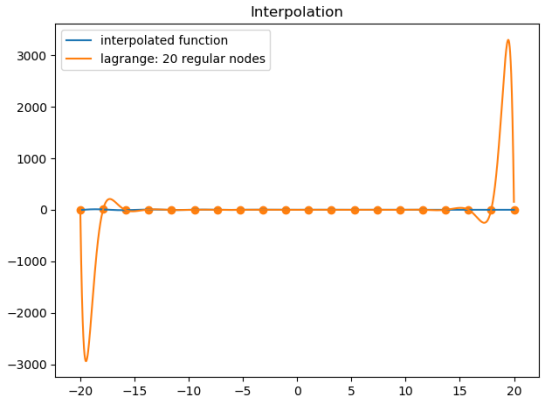


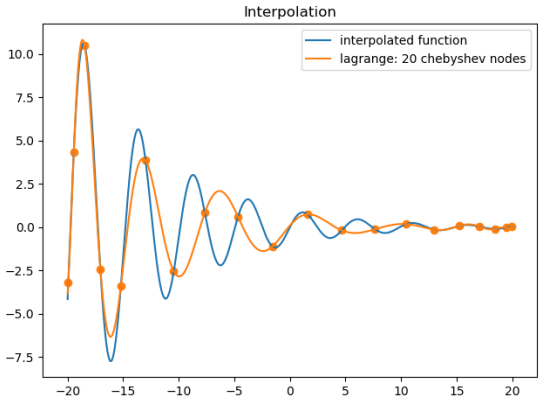


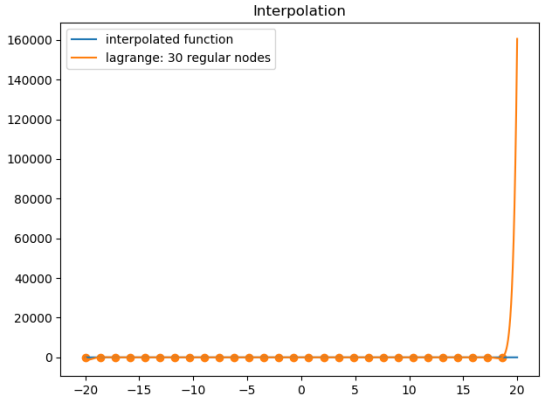


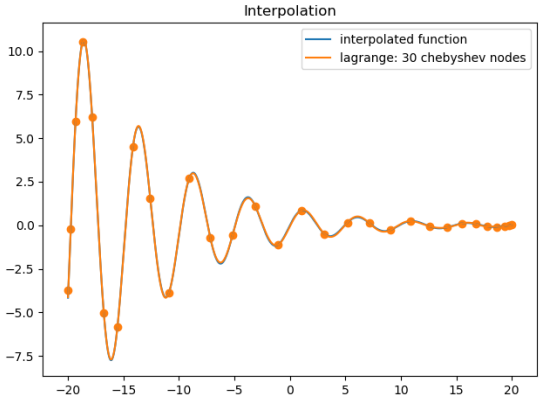












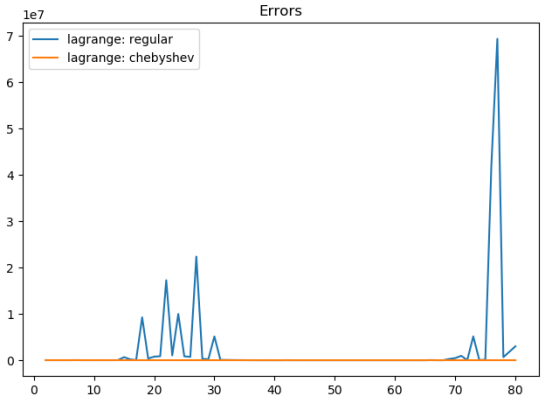
Z powyższych wielomianów funkcję testową najlepiej przybliża wielomian obliczony z wykorzystaniem 30 węzłów Czebyszewa. Nie jest to przypadek. Węzły Czebyszewa nie są losowymi liczbami. Są one wyznaczane właśnie tak, by zminimalizować błąd między wejściową funkcją a przybliżającym wielomianem.

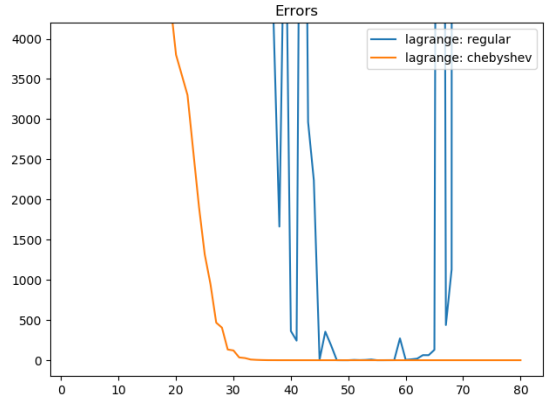
Dokonano analizy błędu w zależności od stopnia wielomianu interpolującego. W tym celu wykorzystano funkcję:

```
double findError(const std::vector<Point> &truePoints,
      const std::vector<Point> &interpolatedPoints)
{
    double error = 0;
    for (int i = 0; i < truePoints.size(); i++)
    {
        error += std::abs(interpolatedPoints[i].getY() - truePoints[i].getY())
;
    }
    return error;
}
```

Funkcja ta liczy sumę wartości bezwzględnych różnic pomiędzy wartościami funkcji testowej a wartościami obliczonego wielomianu.

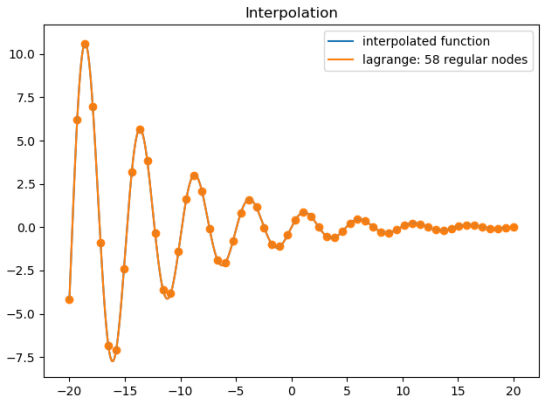
Błąd w zależności od stopnia wielomianu przestawiono na wykresach poniżej.

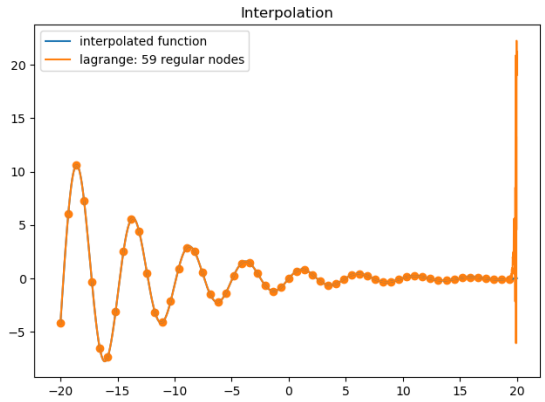




Dla węzłów równoodległych, wielomian dobrze odwzorowuje funkcję testową od 48 do 58 węzłów (z drobnym wzrostem błędu dla 54 węzłów). Natomiast dla węzłów Czebyszewa, od 36 węzłów błąd jest bliski zera.

Zachowanie wielomianu wyznaczanego przy użyciu węzłów Czebyszewa jest logiczne. Wraz ze wzrostem liczby węzłów maleje błąd. Zachowanie wielomianu wyznaczanego przy użyciu węzłów równoodległych jest jednak inne. W okolicy 50 węzłów błąd jest bliski 0, natomiast później ten błąd (dla niektórych liczb węzłów) znacznie wzrasta. Takim przykładem są liczby węzłów 58 i 59. Błąd dla 58 węzłów wynosi 1.09, natomiast dla 59 węzłów - 271.486. Zobrazowano to na wykresach poniżej.





Na prawym rysunku widać, jak wartość wielomianu nagle skacze na granicy obserwowanego przedziału. Jest to tzw. efekt Rungego.

3. Metoda Newtona

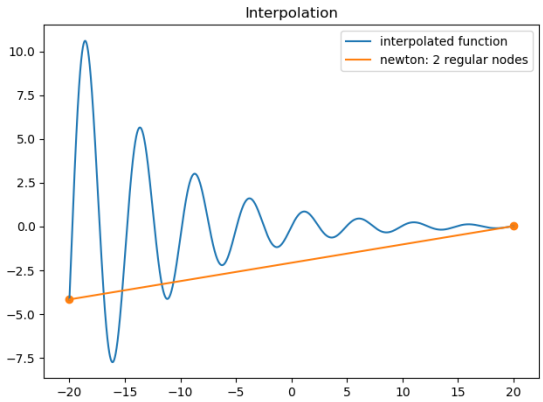
Zaimplementowano metodę interpolacji Newtona.

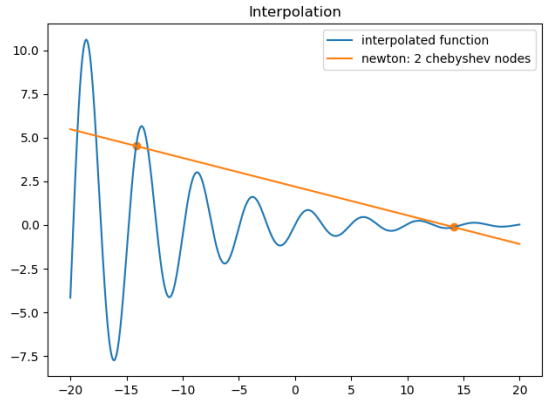
```
std::vector<Point> newtonInterpolation(const std::vector<Point> &nodes,
      const std::vector<double> &pointsX)
{
    int n = nodes.size();
    std::vector< std::vector<double> > F(n, std::vector<double>());
    for (Point node : nodes)
    {
        F[0].push_back(node.getY());
    }
    for (int i = 1; i < n; i++)
    {
        for (int p = 0; p < n - i; p++)
        {
            double val = (F[i - 1][p + 1] - F[i - 1][p]) / (nodes[p + i].getX(
) - nodes[p].getX());
            F[i].push_back(val);
        }
    }

    std::vector<Point> points;
    for (double x : pointsX)
    {
        double y = F[0][0];
        double factor = 1;
        for (int k = 1; k < n; k++)
        {
            factor *= (x - nodes[k - 1].getX());
            y += (factor * F[k][0]);
        }
```

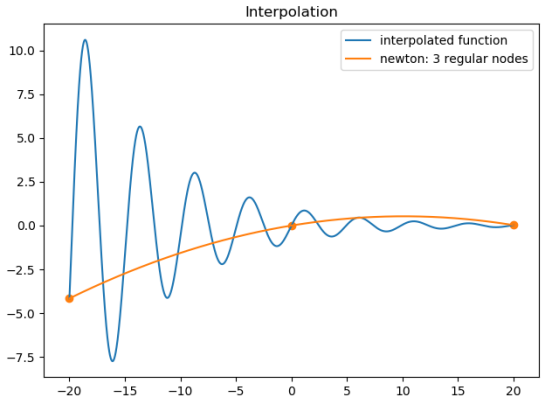
```
        points.push_back(Point(x, y));
    }
    return points;
}
```

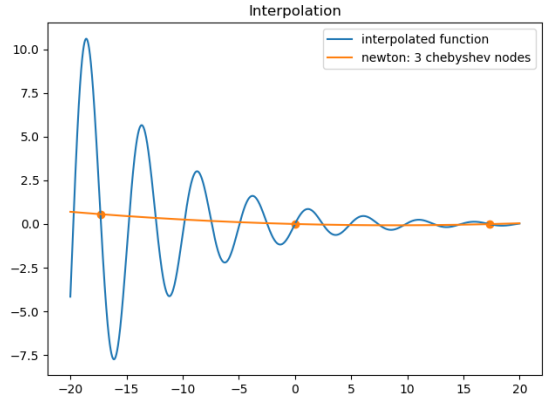
Argumenty i typ zwracanej wartości są takie same jak w przypadku interpolacji Lagrange'a.

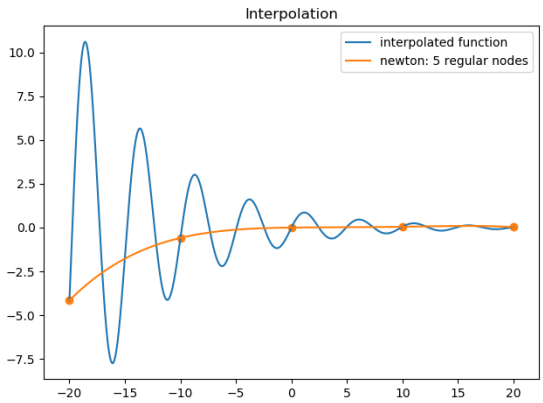
Poniżej zamieszczono przebiegi wielomianów obliczonych dla różnej liczby oraz rozmieszczenia węzłów.

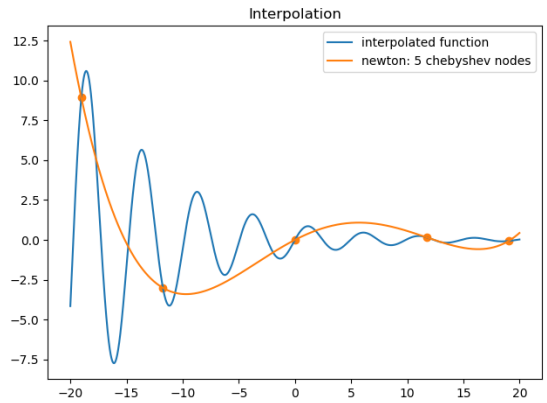


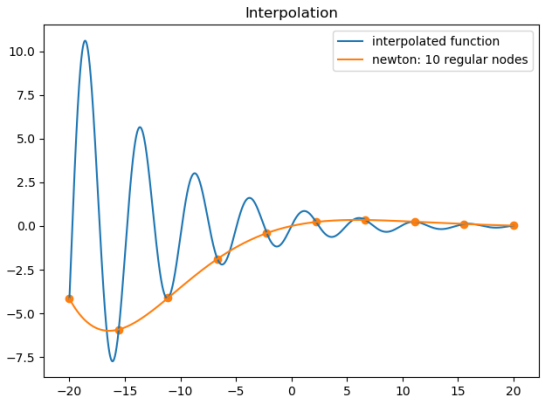


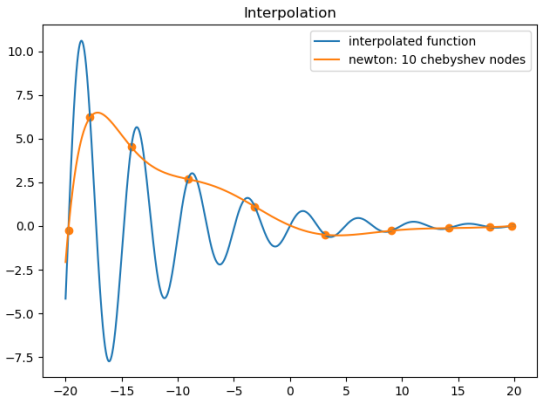


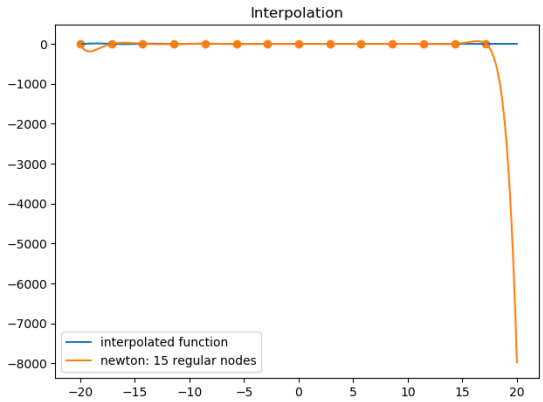


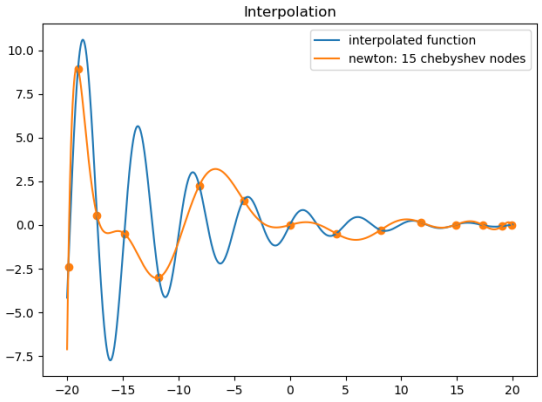


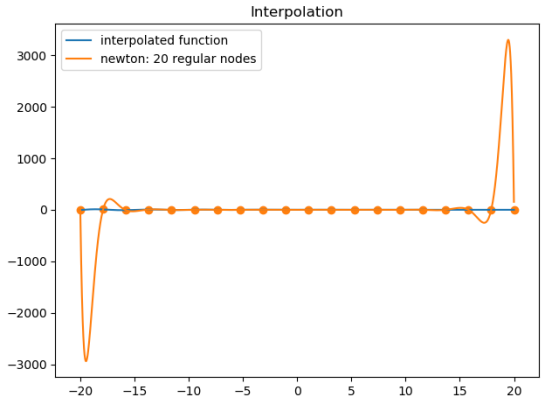


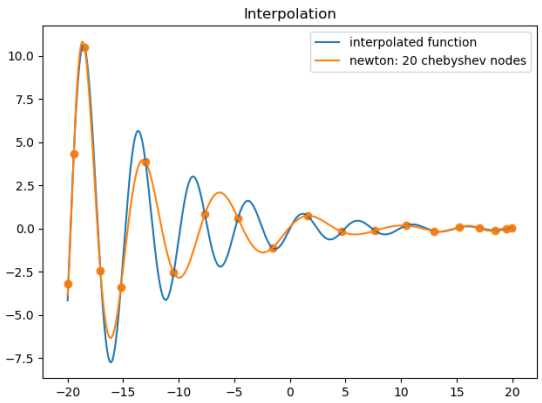


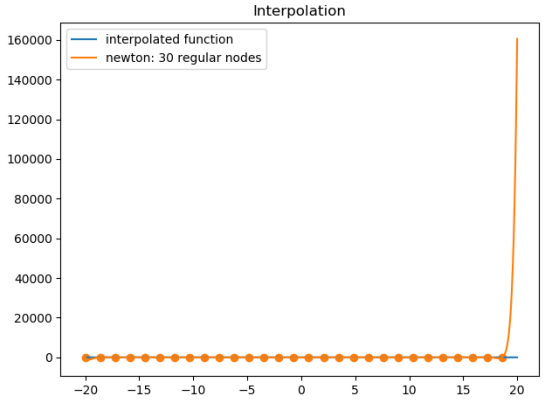


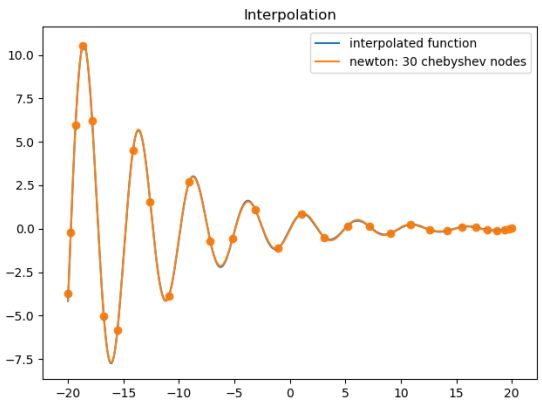






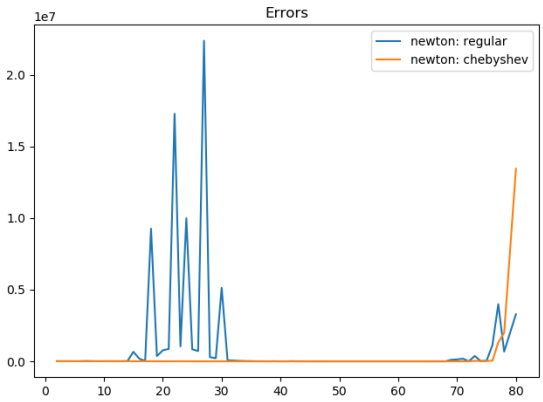


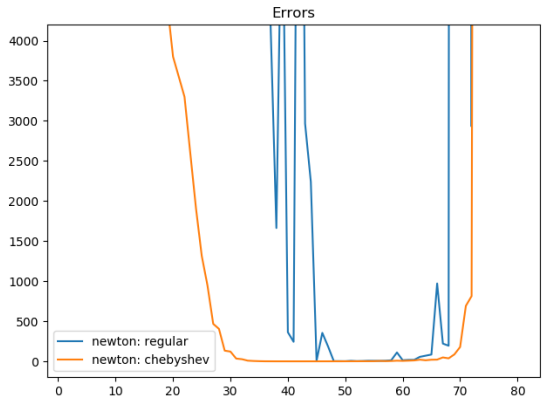




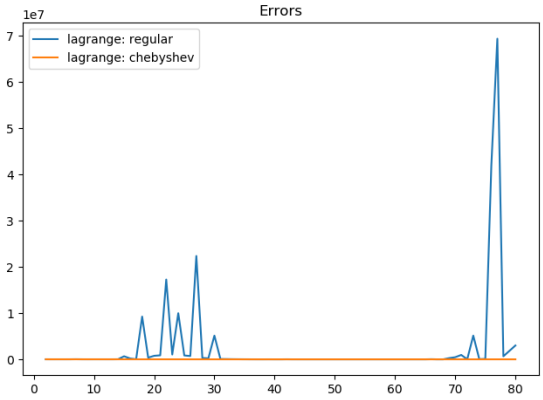
Podobnie jak w poprzednim przypadku, z powyższych wielomianów, funkcję testową najlepiej przybliża ten obliczony z wykorzystaniem 30 węzłów Czebyszewa. Warto zauważyć, że otrzymane wykresy są identyczne w porównaniu do tych otrzymanych dla metody Lagrange'a.

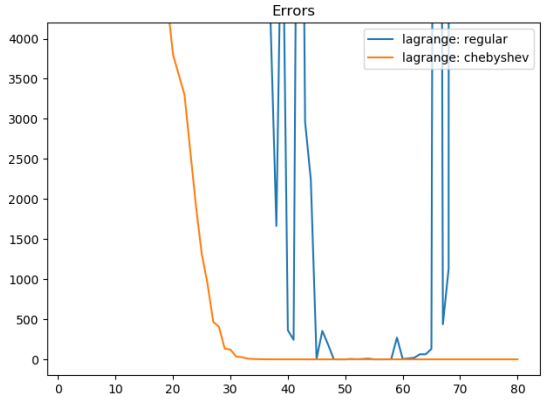
Wykresy z wartościami błędów w zależności od liczby węzłów przedstawiono poniżej.





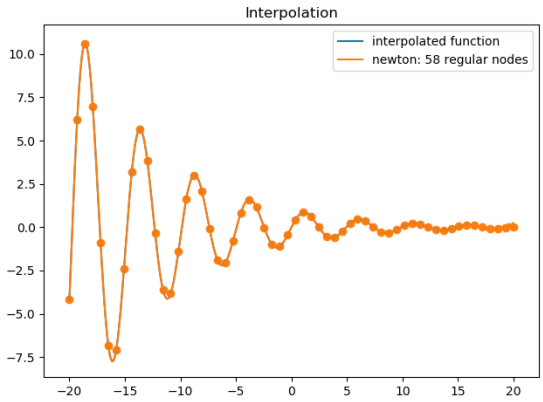
Dla porównania, ponownie przedstawiono także wykresy dla metody Lagrange'a

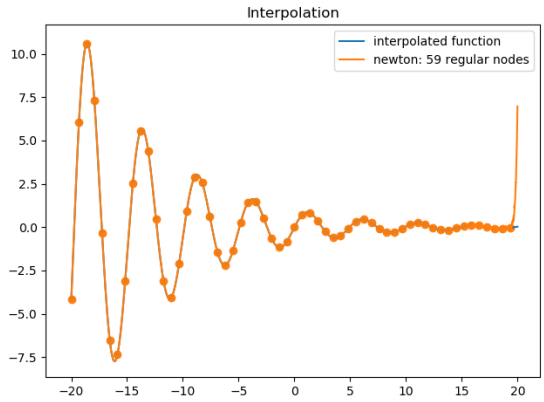




Wykresy dla węzłów równoległych są podobne. Ich kształt nie jest identyczny, natomiast argumenty, dla których wartość błędu jest bliska 0 należą do tego samego przedziału $[58{,}68]$. Wykresy dla węzłów Czebyszewa są identyczne do ok. 60 węzłów. W przeciwieństwie do metody Lagrange'a, gdzie dla węzłów Czebyszewa wartości błędów w miarę ze wzrostem liczby węzłów malały, tutaj od 40 węzłów zaczynają powoli rosnąć. Można także zauważyć, że wartości dla węzłów równoodległych mocno skaczą, natomiast wartości dla węzłów Czebyszewa albo rosną, albo maleją. Jest to spowodowane faktem, iż węzły Czebyszewa wyznaczane są cały czas według tego samego wzoru, natomiast węzły równoodległe rozmieszczone są na pewien sposób losowo. Raz może się przytrafić, że zostaną ułożone dobrze, natomiast już jeden węzeł więcej zmienia to rozmieszczenie i można zaobserwować efekt Rungego.

Efekt Rungego dla równomiernego rozmieszczenia węzłów występuje tu np. 58 węzłów (błąd: 12.95) i 59 węzłów (błąd: 110.52). Podobnie, jak w przypadku interpolacji metodą Lagrange'a, jeden dodatkowy węzeł wystarcza, by wartości na prawym krańcu przedziału znacznie wzrosły.





Wyniki dla obu przedstawionych metod są podobne. Metody te jednak się różnią. Metoda Lagrange'a polega na całkowicie osobnych obliczeniach dla każdego punktu. W metodzie Netwona najpierw zapamiętujemy pewne wartości, których nie trzeba liczyć osobno dla każdego punktu. Dlatego, metoda Newtona będzie lepsza, gdy trzeba aproksymować sporo nieznanych punktów danej funkcji np. w celu narysowania wykresu.

Metoda Lagrange'a natomiast jest nieco dokładniejsza. Ważne jest, że wartości błędu dla węzłów Czebyszewa cały czas maleją. Tę konfigurację więc najlepiej wykorzystać, kiedy chcemy aproksymować kilka nieznanych punktów danej funkcji.

4. Metoda Hermite'a

Zaimplementowano interpolację metodą Hermite'a.

```
std::vector<Point> hermiteInterpolation(const std::vector<double (*)(double)>
      &funs, const std::vector<Point> &nodes,
      const std::vector<double> &pointsX)
{
    int derNum = funs.size();
    int n = nodes.size() * derNum;

    std::vector<double> nodesX;
    std::vector<std::vector<double>> F(n, std::vector<double>());
    for (Point node : nodes)
    {
        for (int i = 0; i < derNum; i++)
        {
            nodesX.push_back(node.getX());
            F[0].push_back(node.getY());
        }
    }

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

```
            fact *= i;
        for (int p = 0; p < n - i; p++)
        {
            int q = p + i;
            if (nodesX[p] == nodesX[q])     // derivative
            {
                F[i].push_back(funs[i](nodesX[p]) / fact);
            }
            else                            // normal
            {
                double val = (F[i - 1][p + 1] - F[i - 1][p]) /
                             (nodesX[q] - nodesX[p]);
                F[i].push_back(val);
            }
        }
    }

    std::vector<Point> points;
    for (double x : pointsX)
    {
        double y = F[0][0];
        double factor = 1;
        for (int k = 1; k < n; k++)
        {
            factor *= (x - nodesX[k - 1]);
            y += (factor * F[k][0]);
        }
        points.push_back(Point(x, y));
    }
    return points;
}
```

Funkcja ta, oprócz charakterystycznych dla poprzednich funkcji argumentów przyjmuje także tablicę funkcji, która powinna zawierać adres funkcji testowej oraz jej pochodnych. W zależności od tego, ile pochodnych znajduje się w tej tablicy, tyle zostanie wykorzystanych przez funkcję interpolującą. Metoda ta jest w gruncie rzeczy podobna do metody Newtona. Wykorzystywane są tu jednak właśnie pochodne. Węzły zapisywane są do odpowiedniej tablicy tyle razy, ile pochodnych chcemy wykorzystać. Następnie, w kolejnych kolumnach, obliczane są nie tylko wartości funkcji, ale także – w odpowiednich miejscach – wartości odpowiednich pochodnych. Ja rozpatrzę dwa przypadki: metodę Hermite'a dla pierwszej pochodnej oraz metodę Hermite'a dla pierwszej i drugiej pochodnej. W tym celu zaimplementowano pochodne funkcji testowej.

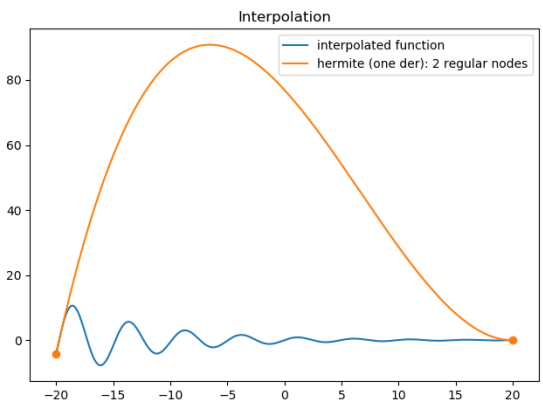
```
inline double df(double x)
{
    double trigArg = k * x / M_PI;
    return exp(-m * x / M_PI) / M_PI * (k * cos(trigArg) - m * sin(trigArg));
}
inline double ddf(double x)
{
```

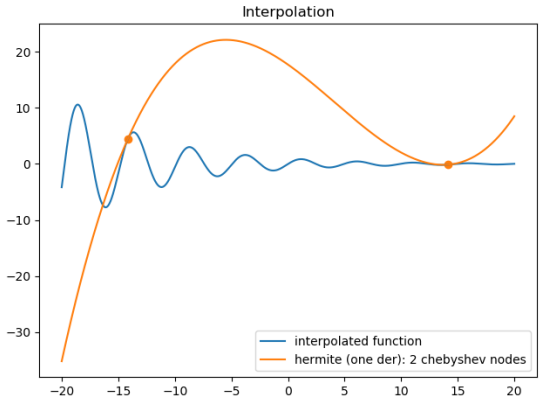
```
    double trigArg = k * x / M_PI;
    return exp(-
m * x / M_PI) / (M_PI * M_PI) * ((m * m - k * k) * sin(trigArg) - 2 * k * m *
cos(trigArg));
}
```

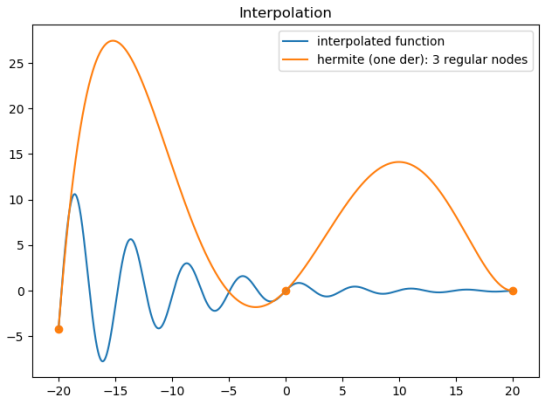
Pochodne można także przybliżać różnicą centralną, ale wówczas oczywiście dokładność będzie mniejsza.

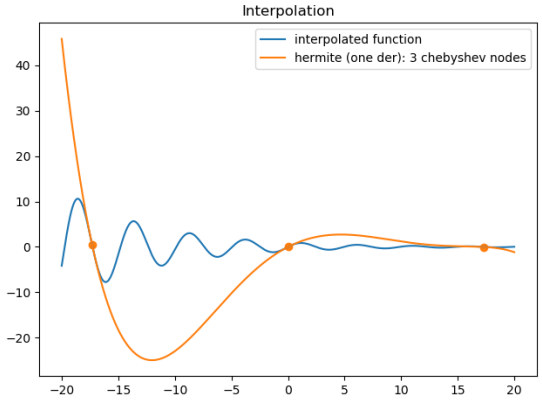
Najpierw rozpatrzę przypadek z wykorzystaniem tylko pierwszej pochodnej.

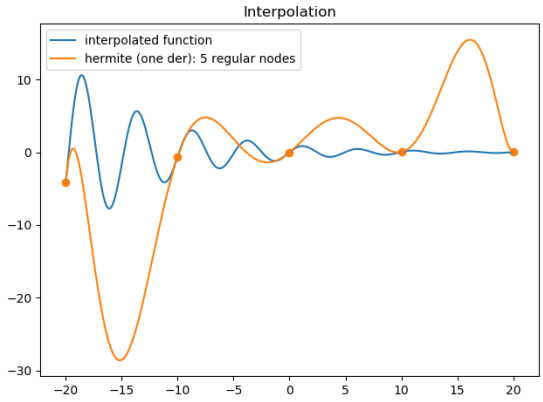
Poniżej zamieszczono przebiegi wielomianów obliczonych dla różnej liczby oraz rozmieszczenia węzłów.

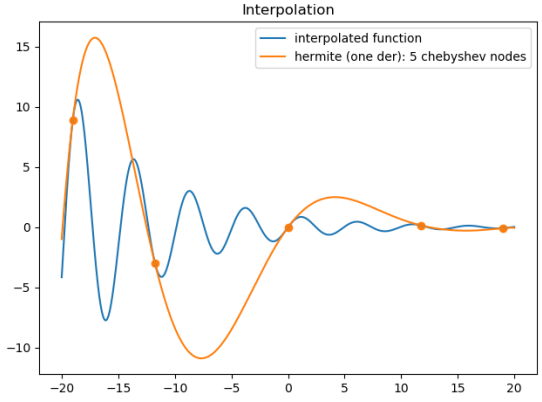


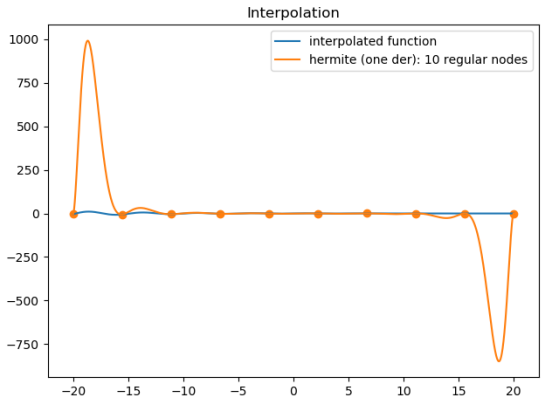


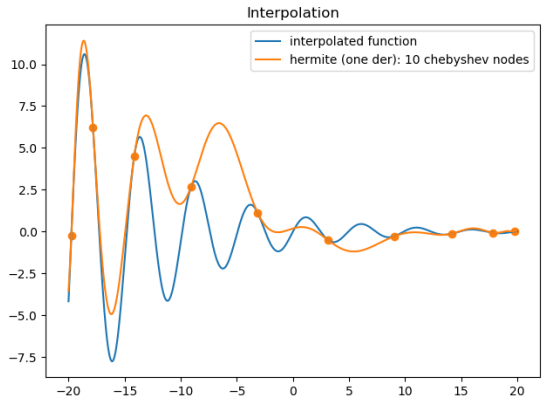


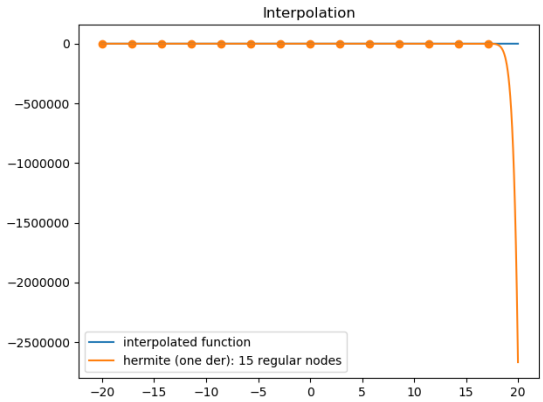


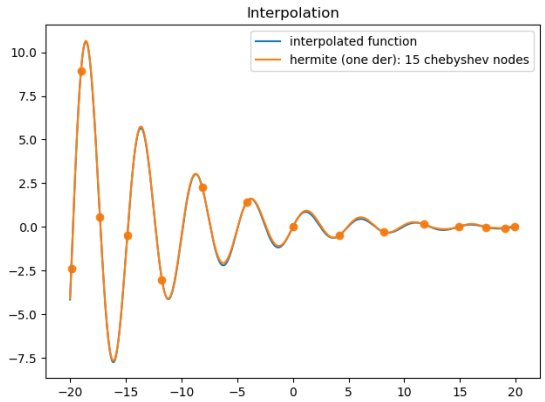


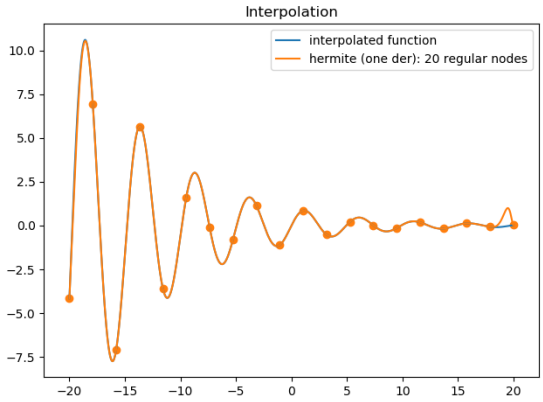


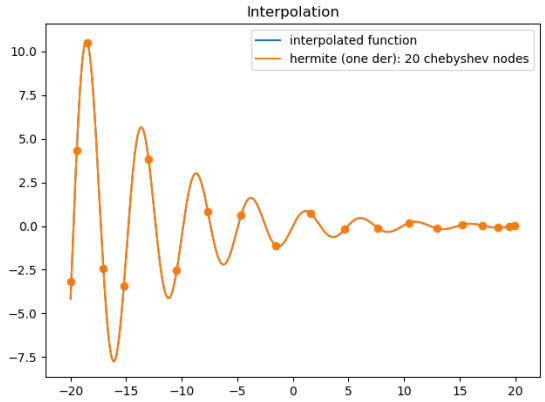


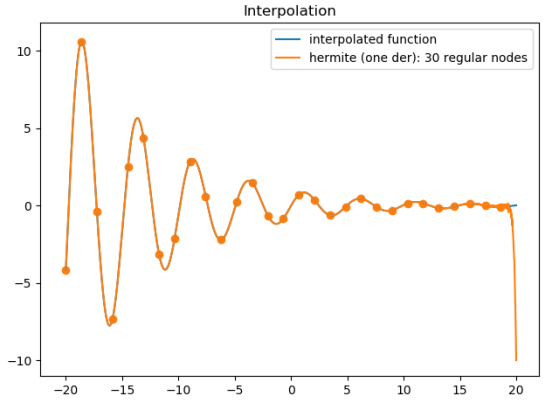


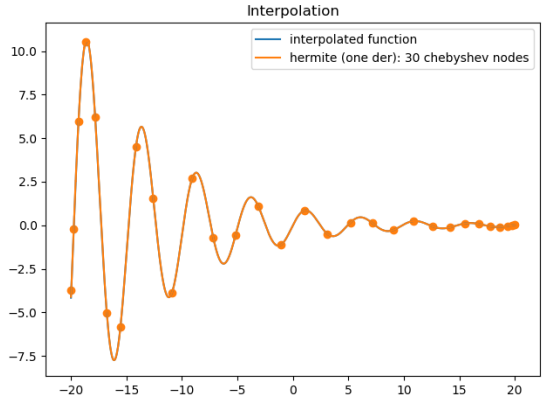






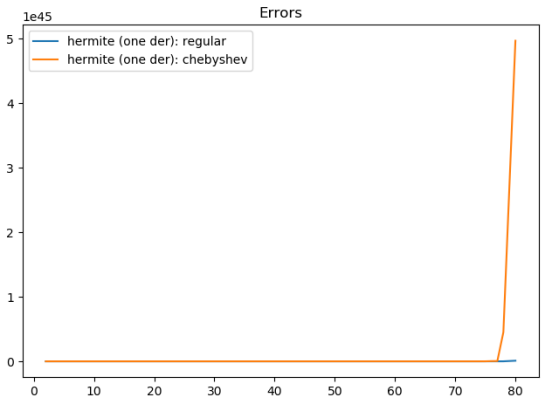


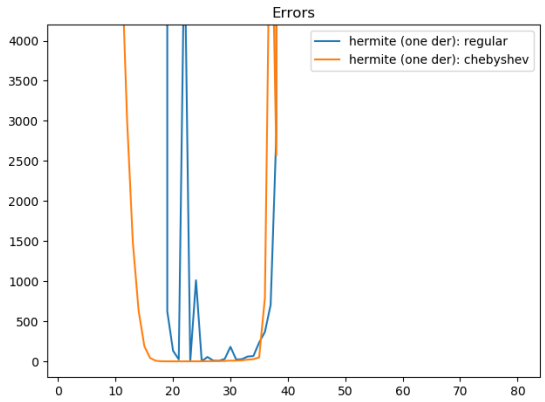




W przypadku węzłów równoodległych, nie udało się zilustrować poprawnego wielomianu, chociaż dla 20 węzłów było blisko. Dla węzłów Czebyszewa, od 15 do 30 węzłów, wykres dobrze ilustruje badaną funkcję (najlepiej – 20 węzłów).

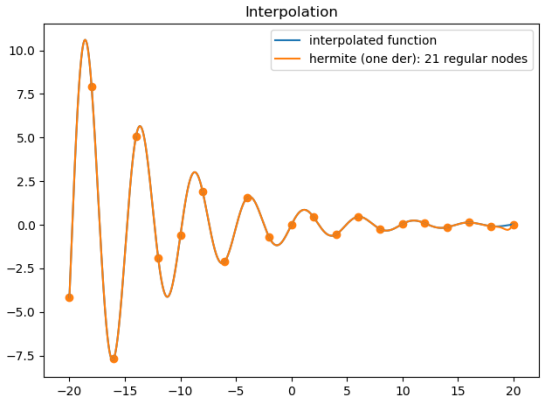
Sporządzono wykresy wartości błędów od liczby węzłów.

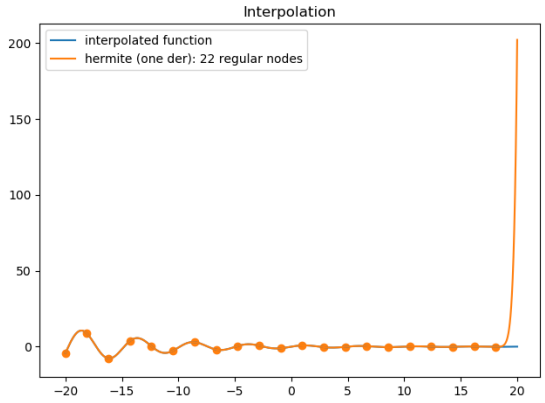




Z wykresów wynika, że praktycznie nie ma sensu używać węzłów równoodległych. Co prawda, kilka razy wartości zbliżają się do 0, natomiast jeden kolejny węzeł wszystko rujnuje (efekt Rungego). Dużo lepiej wypadają węzły Czebyszewa. Błąd jest bliski 0 od 18 do 27 węzłów. Co ciekawe, w tym przypadku wartości błędów od 40 węzłów z każdym nowym węzłem rosną o 1 rząd wielkości i dla 80 węzłów wartość ta jest o wiele większa niż dla węzłów równoodległych.

Przykładowe zajście efektu Rungego dla węzłów równoodległych: dla 21 węzłów wartość błędu wynosi 23.96, natomiast dla 22 - 5362.9.

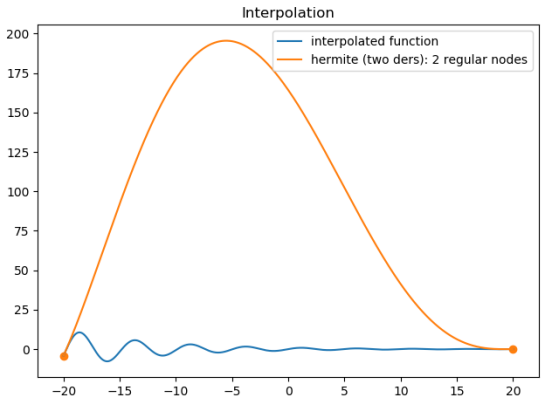


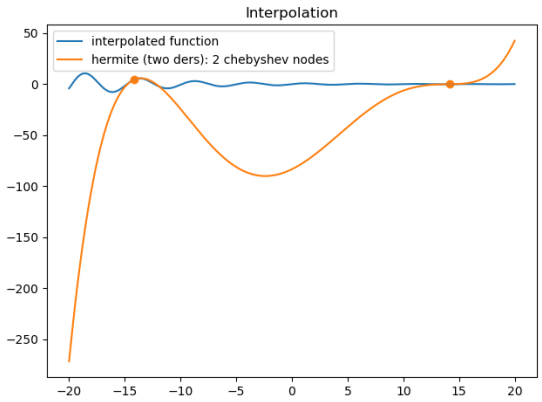


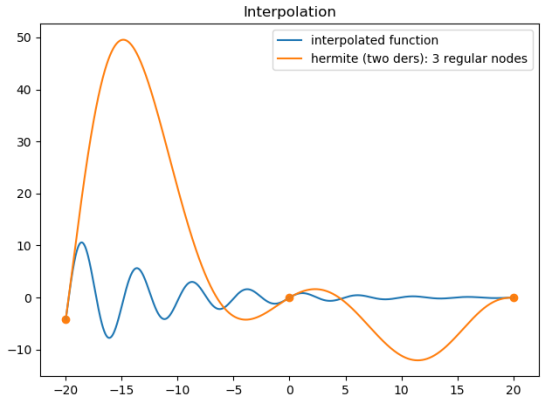
Podobnie, jak w przypadku poprzednich metod, jeden węzeł wystarcza, by funkcja „wystrzeliła” na prawym krańcu przedziału.

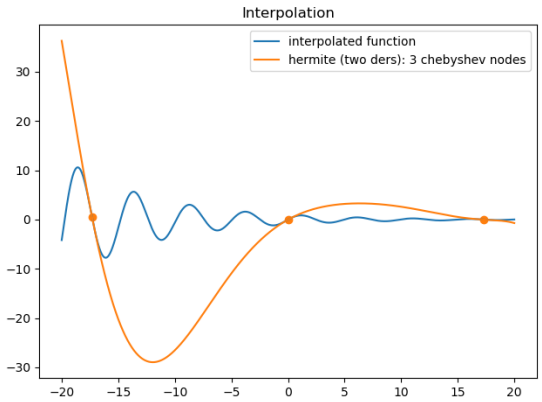
W porównaniu do poprzednich metod jednak, błąd dla węzłów Czebyszewa staje się bliski 0 dużo wcześniej. Metoda ta zatem będzie lepsza dla przypadków, w których znamy mniej wartości danej funkcji. Zaletą tej metody jest fakt, iż nie łączy ona po prostu punktów (tak z grubsza wyglądało to dla poprzednich metod - dla małej liczby węzłów), tylko tworzy wielomian styczny do funkcji w punktach. W przypadku funkcji szybko zmiennych może to być znaczące.

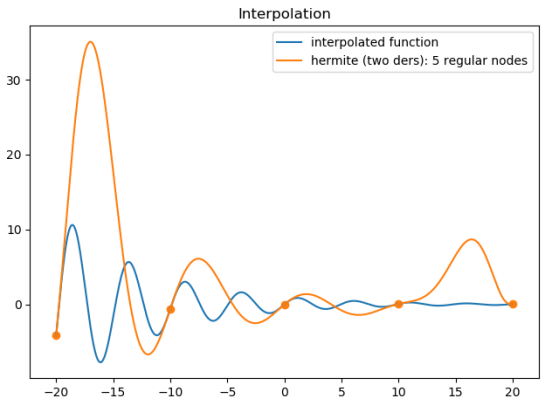
Rozpatrzymy jeszcze przypadek z wykorzystaniem pierwszej oraz drugiej pochodnej.

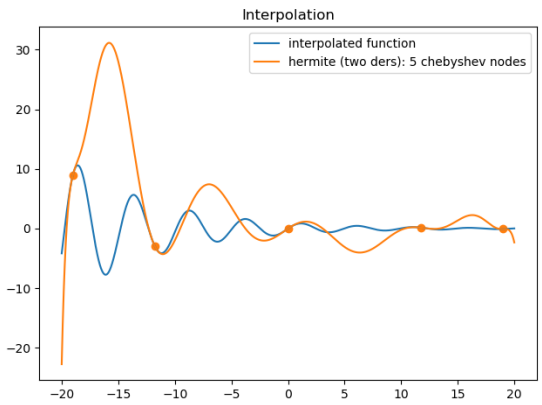

Interpolation
interpolated function
hermite (two ders): 2 regular nodes

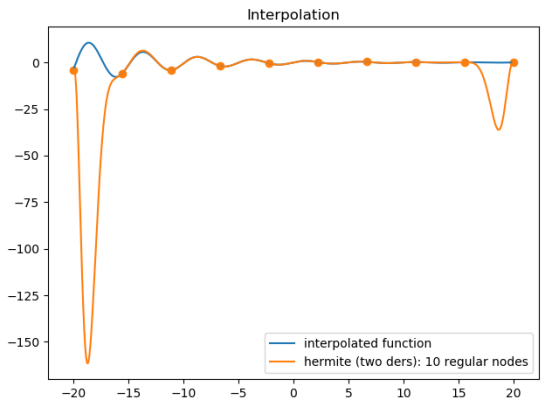

Interpolation
interpolated function
hermite (two ders): 2 chebyshev nodes

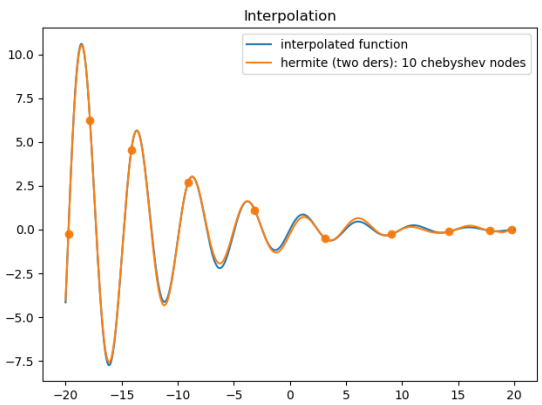

Interpolation
interpolated function
hermite (two ders): 3 regular nodes

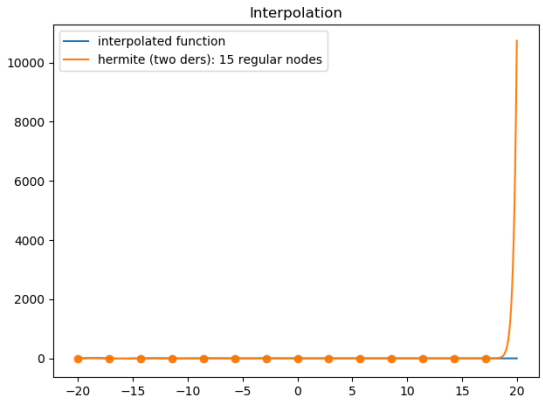

Interpolation
interpolated function
hermite (two ders): 3 chebyshev nodes

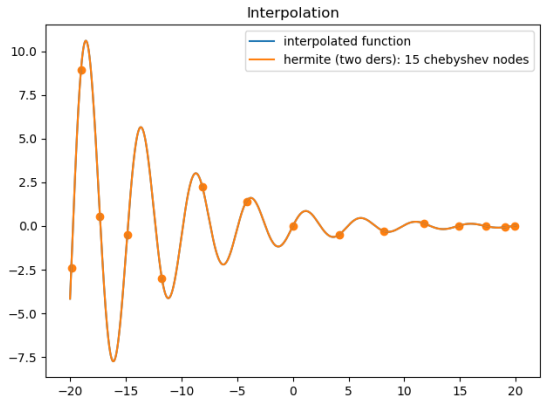

Interpolation
interpolated function
hermite (two ders): 5 regular nodes

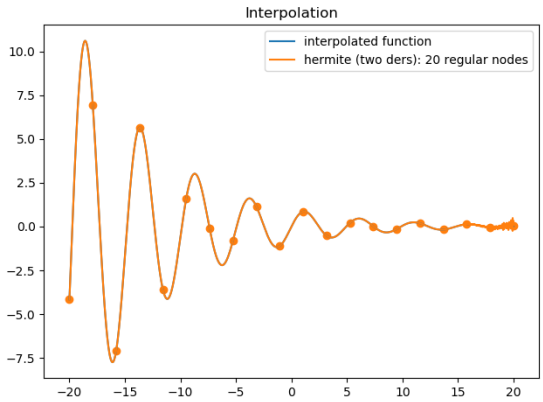

Interpolation
interpolated function
hermite (two ders): 5 chebyshev nodes

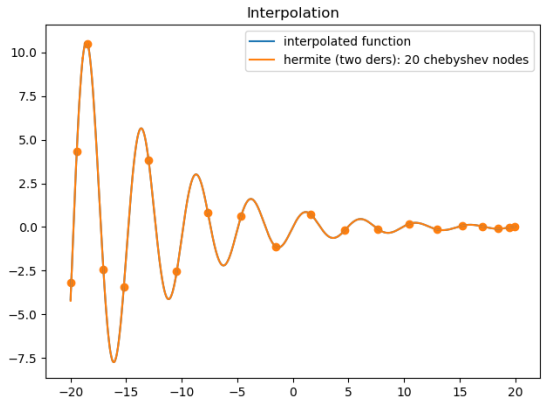

Interpolation
interpolated function
hermite (two ders): 10 regular nodes

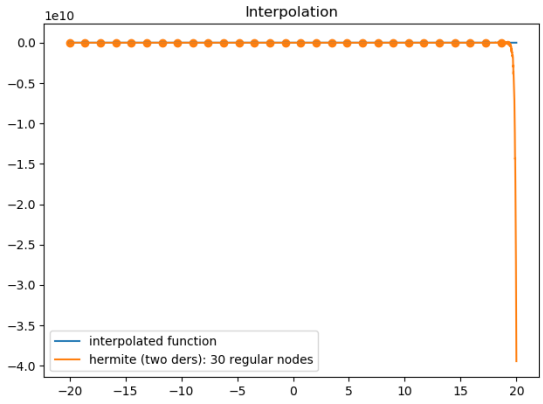

Interpolation
interpolated function
hermite (two ders): 10 chebyshev nodes

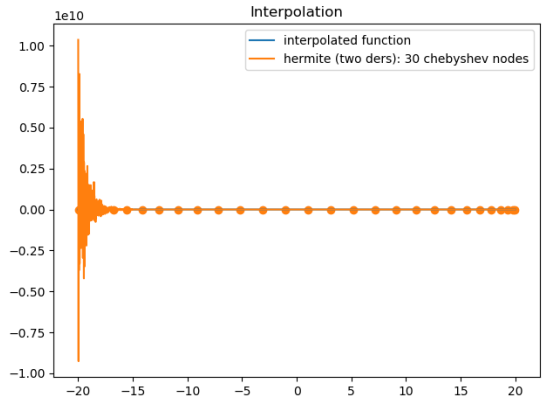




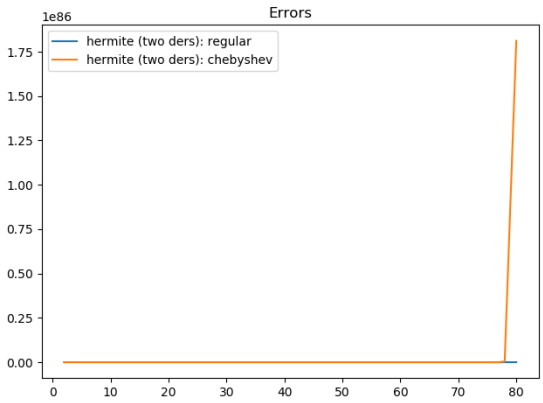


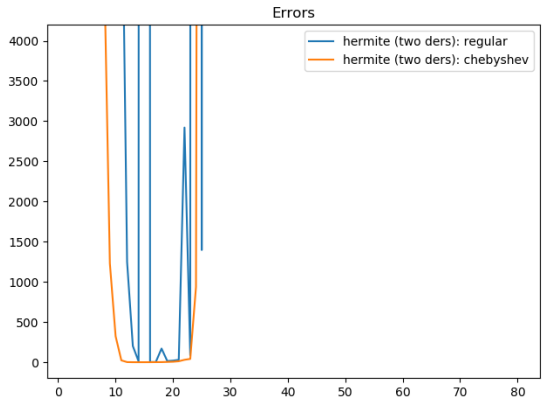






Wykresy są bardzo podobne do tych dla tylko pierwszej pochodnej. Wszystko jednak jest przesunięte w lewo na osi liczby węzłów, tj. przybliżenie węzłami Czebyszewa szybciej osiąga błąd bliski 0, ale też szybciej go traci. Można to zobaczyć na wykresach z wartościami błędów poniżej.





Już dla 13 węzłów Czebyszewa błąd wynosi tylko 0.16.

Ciekawą zależność widać dla 30 węzłów Czebyszewa. To samo jednak można zaobserwować z wykorzystaniem tylko pierwszej pochodnej, jednakże dla większej liczby węzłów.

Można wywnioskować, że im więcej pochodnych użyjemy, tym szybciej błąd z wykorzystaniem węzłów Czebyszewa zmaleje blisko 0. Użycie wielu pochodnych wydaje się idealne dla funkcji, w przypadku których znamy tylko kilka punktów. Problemem jest jednak konieczność znajomości pochodnych i fakt, że przybliżanie ich wartości może nie być wystarczająco dokładne.