2019

Metody numeryczne

PROJEKT

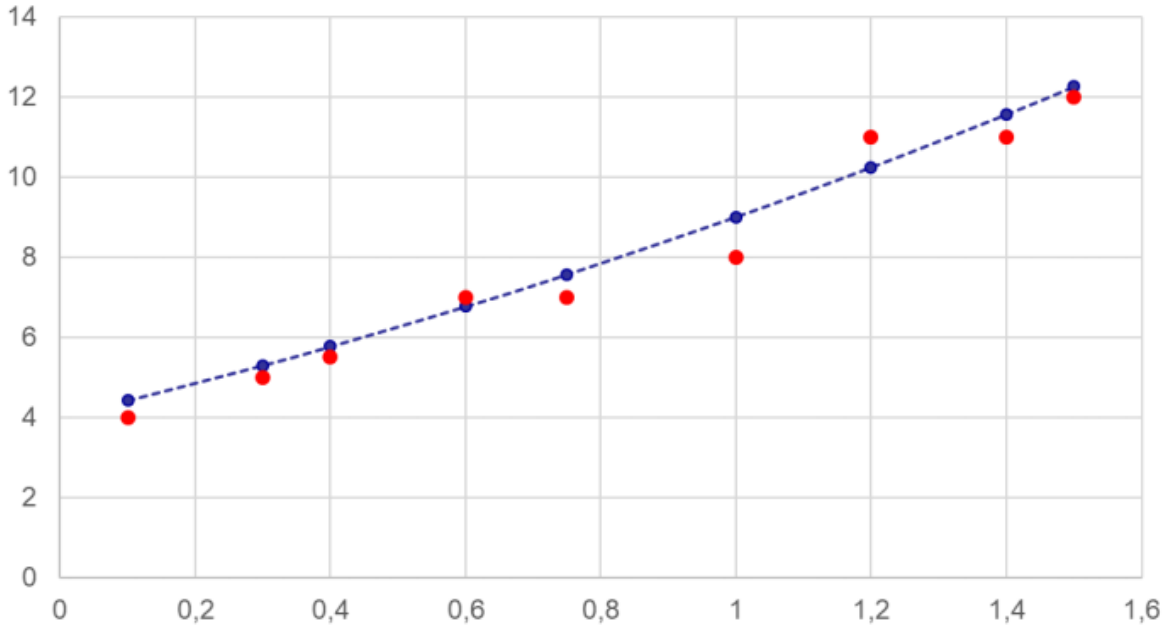
DAWID KRUCZEK

1. Wstęp

W moim projekcie skupiłem się na implementacji dwóch metod numerycznych: *aproksymacji średniokwadratowej* oraz *metody Brenta*. Do rozwiązania problemu posłużyłem się językiem C++ oraz frameworkiem QT, dzięki któremu stworzyłem interfejs graficzny (GUI). Sprawozdanie zostało podzielone na 3 obszary związane z każdą metodą: **opis teoretyczny metody**, **testowanie metody**, **wnioski**.

2. Aproksymacja średniokwadratowa – teoria

Aproksymacja to inaczej przybliżanie funkcji intuicyjnie:



Aproksymacja średniokwadratowa wielomianami:

- Aproksymacja funkcji w przestrzeni $L_p^2[a,b]$ nazywana jest aproksymacją średniokwadratową
- **Aproksymacja funkcji w przestrzeni $l_{p,N}^2$ nazywana jest aproksymacją średniokwadratową dyskretną**

W moim projekcie skupiłem się na aproksymacji średniokwadratowej dyskretnej.

Założenie:

Przybliżamy funkcję wielomianami

Wielomian optymalny

Element optymalny dla funkcji f względem przestrzeni W_n wielomianów stopnia nie wyższego niż n, czyli wielomian w_n^*, dla którego zachodzi równość:

$$\|f - w_n^*\| = \inf_{w \in W_n} \|f - w\|$$

gdzie:

$$\|f - w\| := \begin{cases} \left(\int_a^b (f(x) - w(x))^2 p(x) dx \right)^{1/2} & dla \quad L_p^2[a,b] \\ \sum_{i=1}^{N} (f(x_i) - w(x_i))^2 p(x_i) & dla \quad l_{p,N}^2 \end{cases}$$

Nazywamy n-tym wielomianem optymalnym w sensie aproksymacji średniokwadratowej z funkcją wagową odpowiednio na przedziale [a,b] lub zbiorze dyskretnym $x_1, x_2, \ldots, x_n$.

Przestrzeń unitarna

W rozważaniu aproksymacji średniokwadratowej jest również ważne pojęcie przestrzeni unitarnej – jest to przestrzeń liniowa V, w której jest określona funkcja zwana iloczynem skalarnym, która parze elementów f, g € V przyporządkowuje liczbę rzeczywistą (f,g) spełniającą następujące warunki:

1. $(f,f) \geq 0; (f,f) = 0 \Leftrightarrow f = 0$
2. $(f,g) = (g,f)$
3. $(\alpha f + \beta g, h) = \alpha(f,h) + \beta(g,h) \; \forall f,g,g \in V \; i \; liczb \; rzeczywistych \; \alpha \; i \; \beta$

Układ ortogonalny

- Jeśli $(f,g) = 0$ to mówimy, że dwa elementy są wzajemnie prostopadłe (ortogonalne). Skończony lub nieskończony zbiór niezerowych elementów $f_1, f_2, \ldots, f_n, \ldots$przestrzeni unitarnej nazywamy ortogonalnym, jeśli każde dwa różne od siebie elementy są do siebie ortogonalne.
- Jeśli dodatkowo$(f_k, f_k) = 1$, to układ nazywamy ortonormalnym.
- Element jest ortogonalny do przestrzeni, jeśli jest ortogonalny do każdego elementu tej przestrzeni.

Aproksymacja w przestrzeniach unitarnych

W przestrzeni unitarnej, gdzie normę definiujemy poprzez iloczyn skalarny $||f|| = \sqrt{(f,f)}$ można wykazać, że jeśli V jest przestrzenią unitarną, U jej podprzestrzenią, a f dowolnym elementem z V, to element h* € U jest optymalnym dla f względem podprzestrzeni U wtedy i tylko wtedy gdy f-h* jest ortogonalny do U.

Element optymalny

- Jeśli h^* jest elementem optymalnym, to $f - h^*$ jest ortogonalny do wszystkich elementów f_i bazy podprzestrzni U.
- Z tego wynika zależność: $(f - h^*, f_i) = 0$ a więc $(f, f_i) = (h^*, f_i)$.
- Z kolei h^* należy do podprzestrzeni U z bazą f_i, a więc można go zapisać jako: $h^* = \sum_{j=1}^{n} \alpha_j f_j$
- Czyli: $(h^*, f_i) = \sum_{j=1}^{n} \alpha_j (f_j, f_i) = (f, f_i)$ co można zapisać w postaci układu równań liniowych:

$$\begin{aligned}
&\alpha_1(f_1, f_1) + \alpha_2(f_2, f_1) + \ldots + \alpha_n(f_n, f_1) = (f, f_1) \\
&\alpha_1(f_1, f_2) + \alpha_2(f_2, f_2) + \ldots + \alpha_n(f_n, f_2) = (f, f_2) \\
&\quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
&\alpha_1(f_1, f_n) + \alpha_2(f_2, f_n) + \ldots + \alpha_n(f_n, f_n) = (f, f_n)
\end{aligned}$$

Jeśli baza rozpatrywanej przestrzeni jest bazą ortogonalną, to układ równań jest układem z macierzą diagonalną i ma rozwiązanie:

$$\alpha_i = \frac{(f, f_i)}{(f_i, f_i)}$$

Element optymalny wówczas możemy zapisać jako:

$$h^* = \sum_{j=1}^{n} \frac{(f, f_j)}{(f_j, f_j)} f_j$$

Implementacja metody w C++

```
class Punkt {
public:
    double x, y;
};

class Wielomian {
public:
    Punkt* tab;
    int n;

    Wielomian(int N) {
        this->n = N;
        tab = new Punkt[n];
    }
};
```

Rysunek 1 Klasy bazowe z których korzystam podczas tworzenia punktów i wielomianu

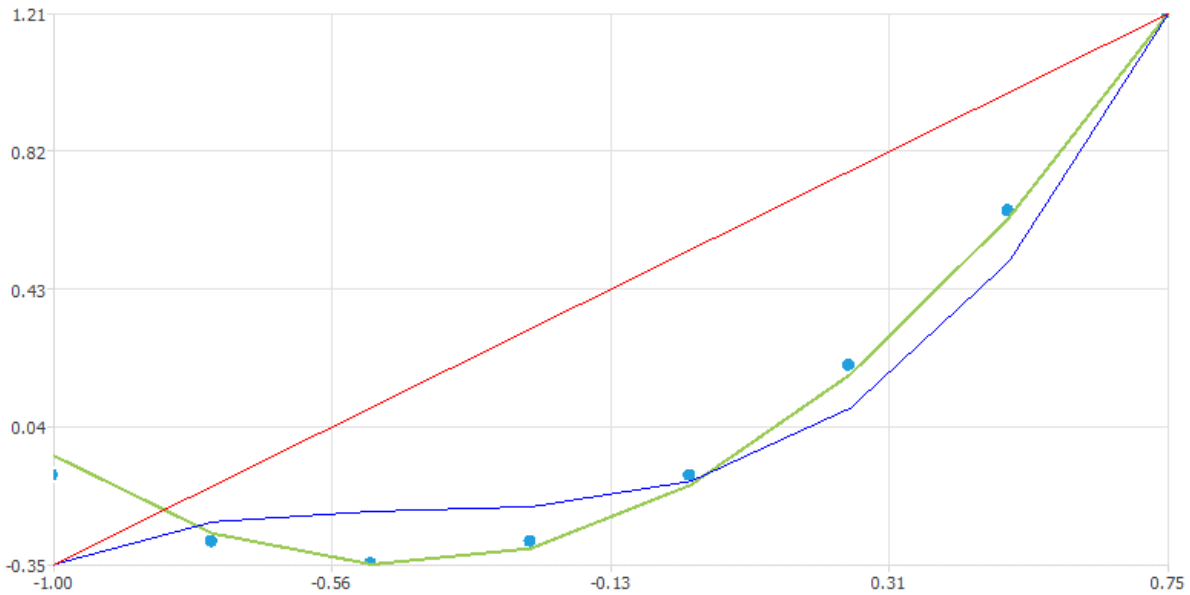
```
double Legendre(int k, double x) {
    if (k == 0) {
        return 1;
    }
    if (k == 1) {
        return x;
    }
    double K = (double)k;
    return ((((2 * K) - 1) / K) * x * Legendre(k - 1, x)) - (((K - 1) / K)* Legendre(k - 2, x));
}
```

Rysunek 2 Funkcja obliczająca wielomian Legendre'a

```
double elementOptymalny(Wielomian a, double x, int n) {
    double returnValue = 0.0;

    for (int i = 0; i < n; i++) {
        double licznik = 0.0;
        double mianownik = 0.0;
        for (int j = 0; j < a.n; j++) {
            double legendrei = Legendre(i, a.tab[j].x);
            licznik += a.tab[j].y*legendrei;
            mianownik += legendrei * legendrei;
        }
        returnValue += (licznik / mianownik)*Legendre(i, x);
    }
    return returnValue;

}
```

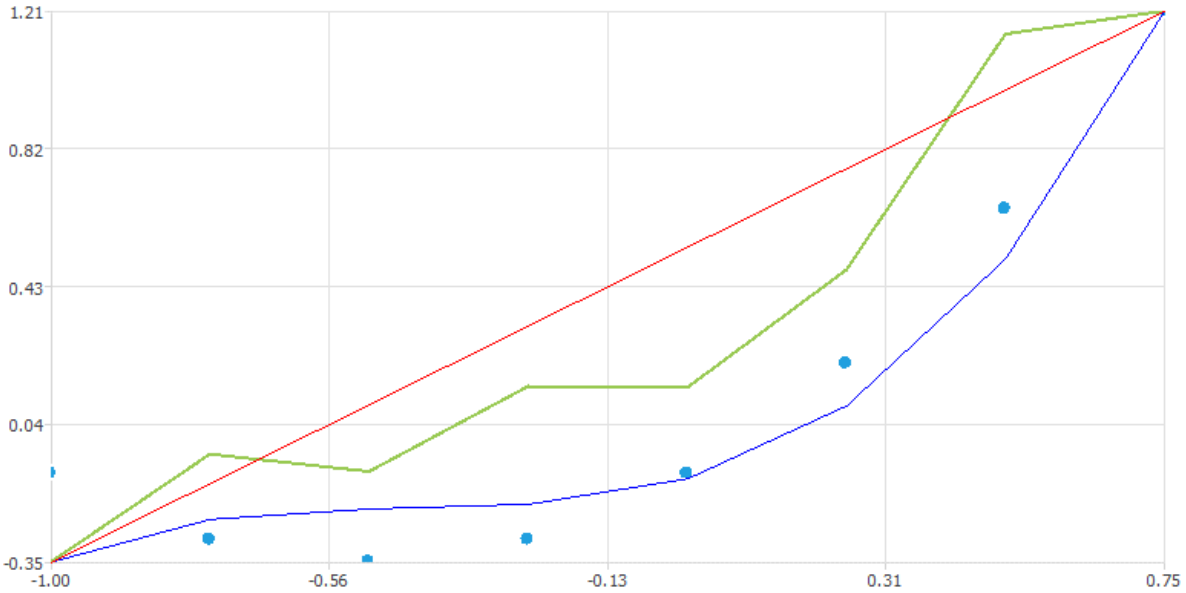
Rysunek 3 Funkcja wyznaczająca eleemnt optymalny

3. Aproksymacja średniokwadratowa –testy
    a. Funkcja (x*(x + 1) - 0.1) określana wielomianem st. 3 (dla porównana określono również wielomian st.2 i st.4)



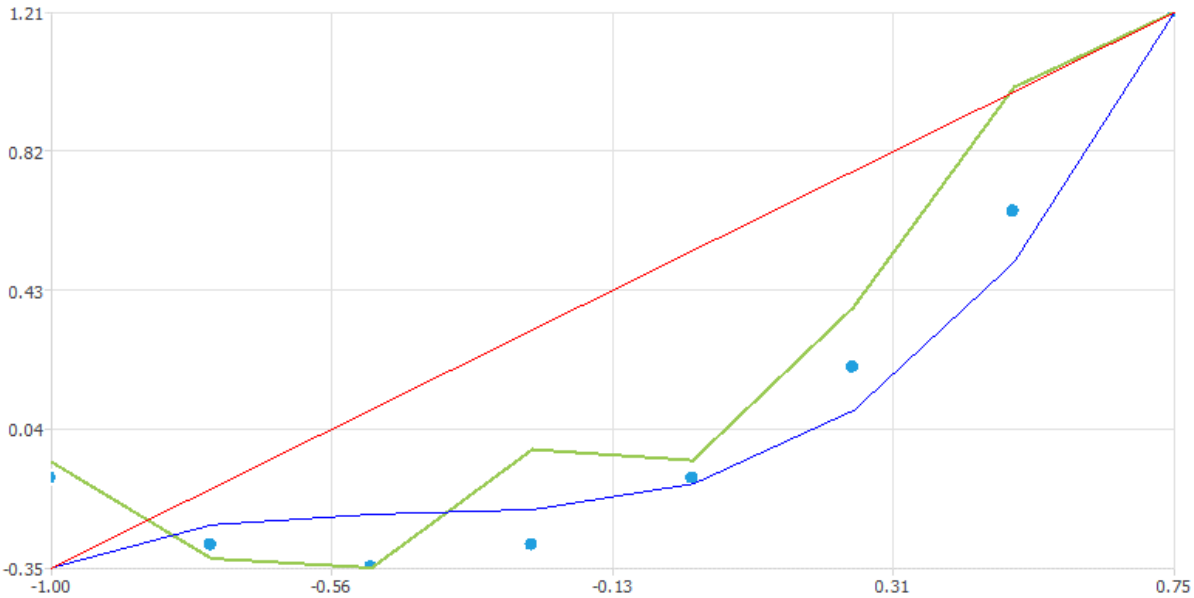
Algorytm wykonywał się 79ms

b. Funkcja (x*(x + 1) - 0.1) określana wielomianem st. 8 (dla porównana określono również wielomian st.7 i st.9)



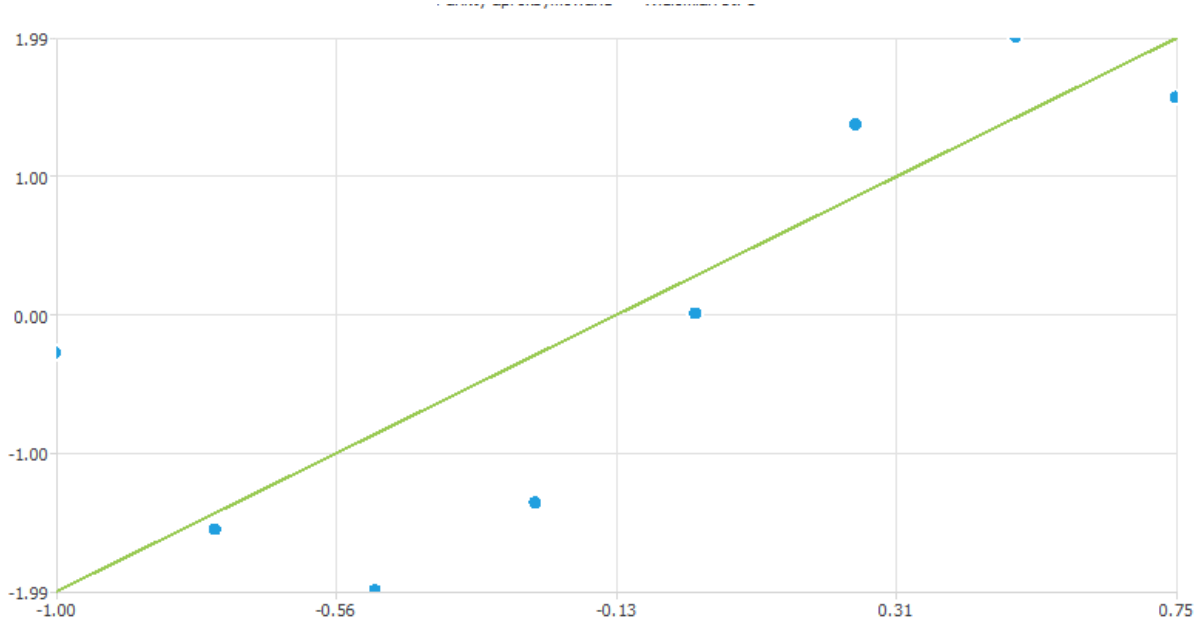
Algorytm wykonywał się 173ms

c. Funkcja (x*(x + 1) - 0.1) określana wielomianem st. 15 (dla porównana określono również wielomian st.14 i st.16)



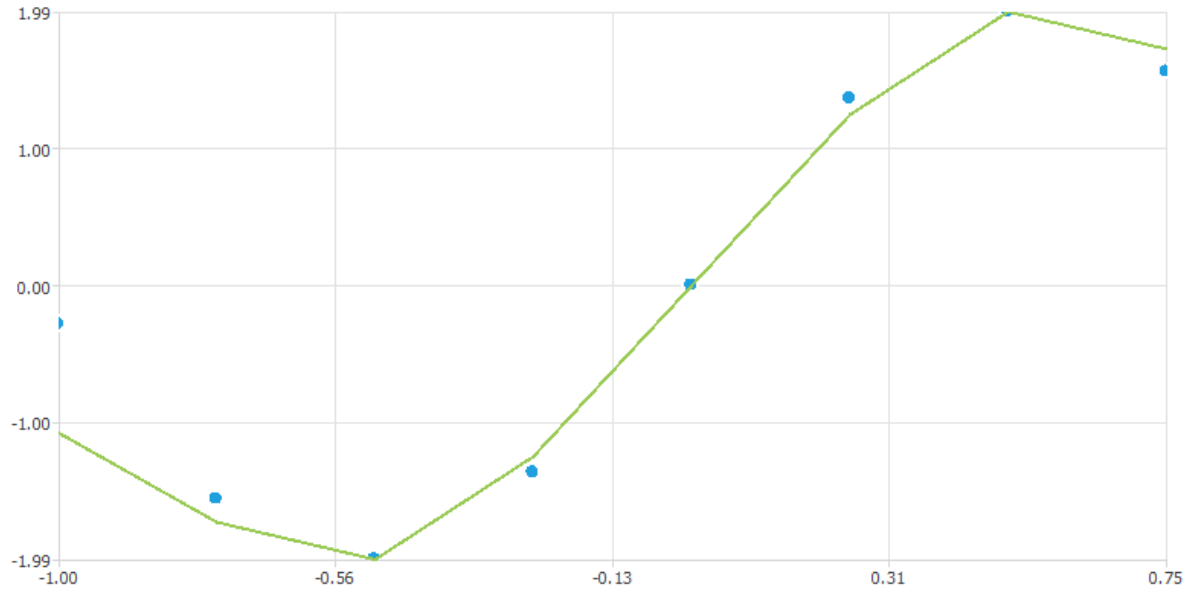
Algorytm wykonywał się 5661ms

d. Funkcja sin(3x)*2cos((4/5)x) określana wielomianem st. 3



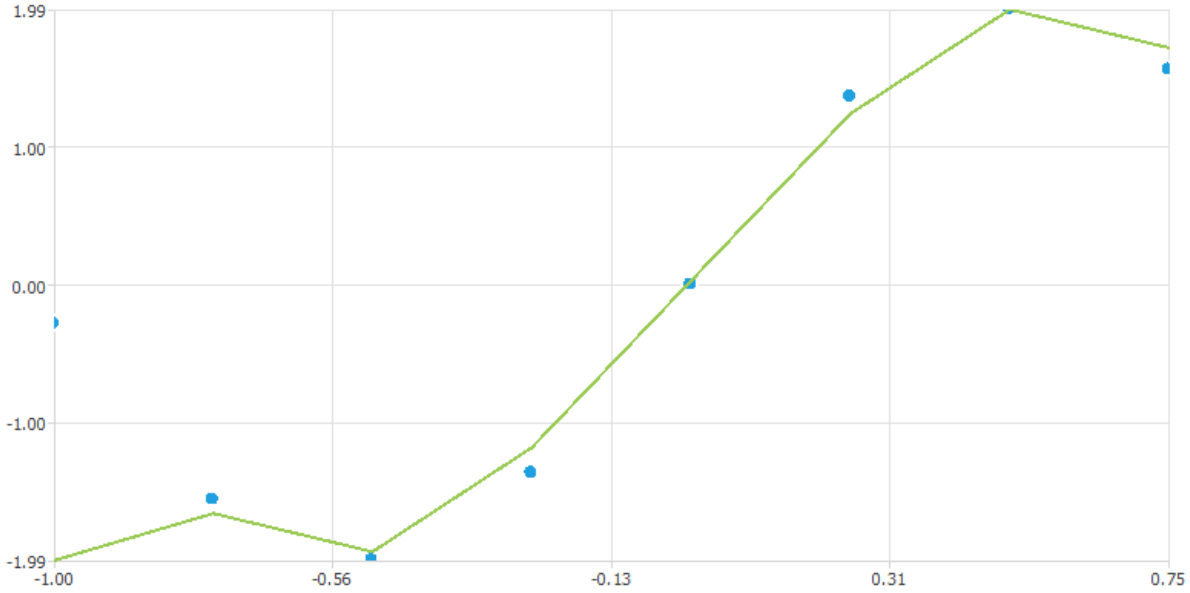
Algorytm wykonywał się 11ms

e. Funkcja sin(3x)*2cos((4/5)x) określana wielomianem st. 8



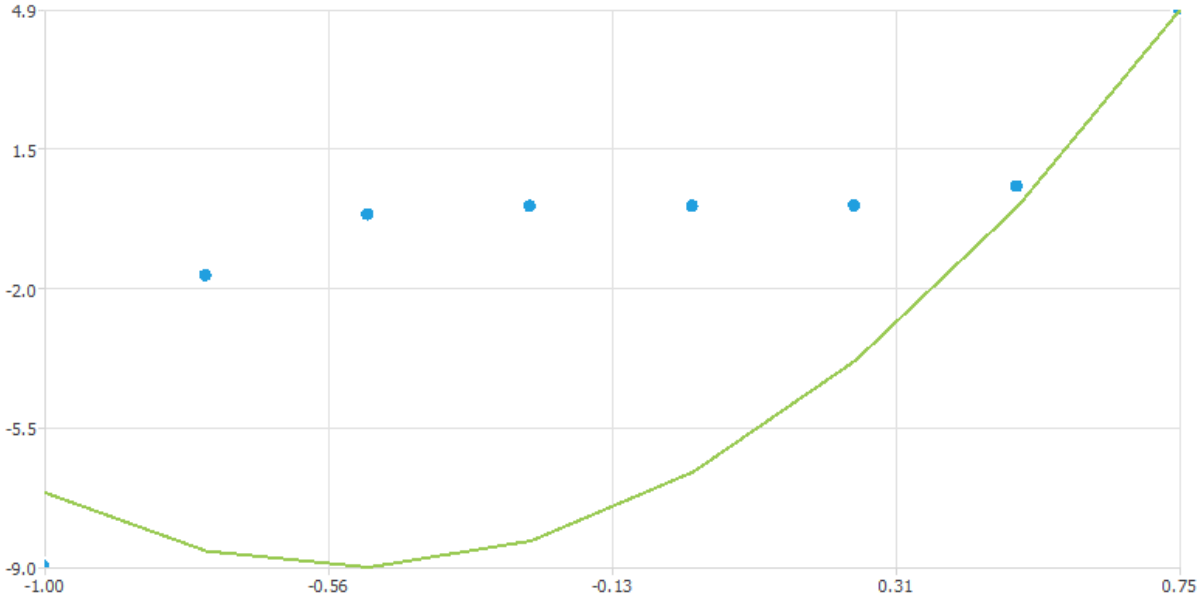
Algorytm wykonywał się 205ms

f. Funkcja sin(3x)*2cos((4/5)x)) określana wielomianem st. 15



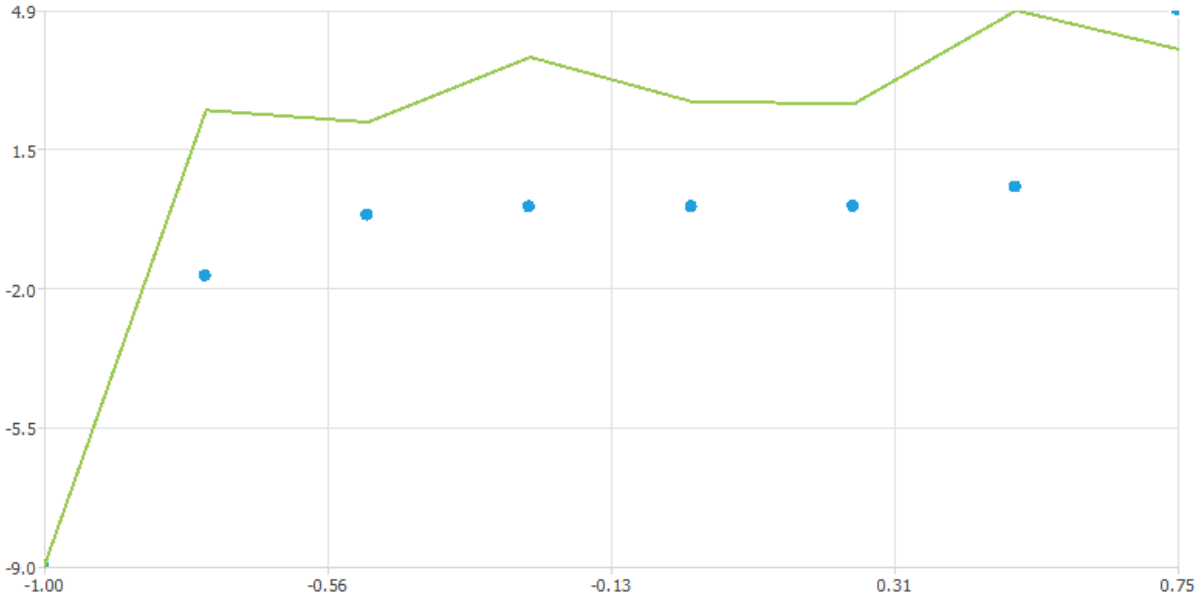
Algorytm wykonywał się 5490ms

g. Funkcja $(3x + 3x^2 + 3x^3) * 3x^4$ określana wielomianem st. 3



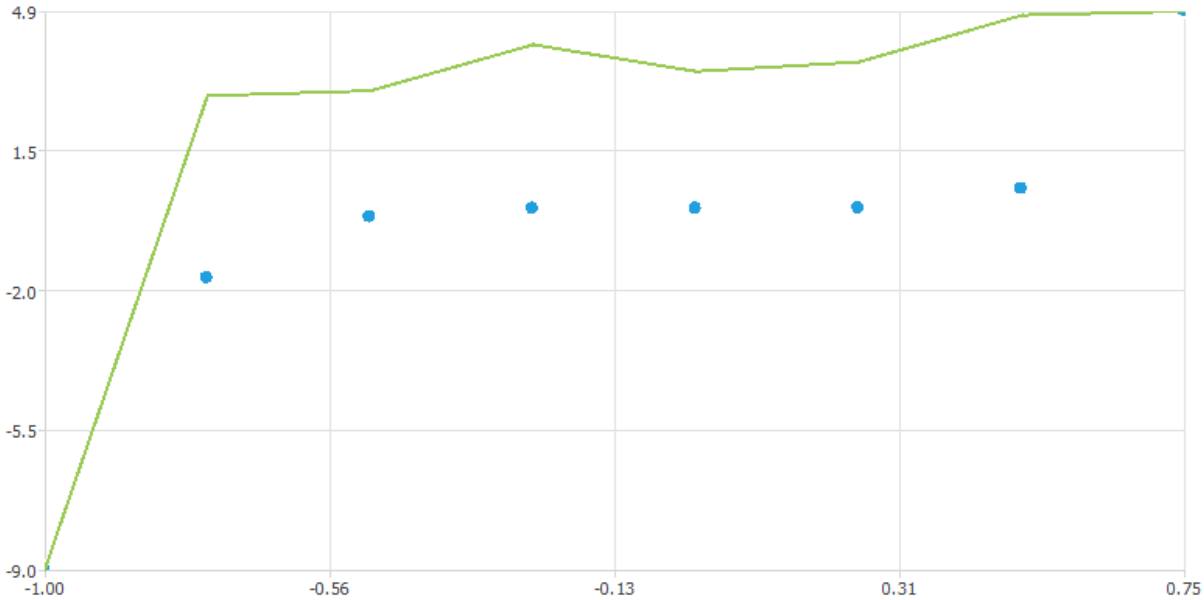
Algorytm wykonywał się 10ms

h. Funkcja $(3x + 3x^2 + 3x^3) * 3x^4$ określana wielomianem st. 8



Algorytm wykonywał się 182ms

i. Funkcja $(3x + 3x^2 + 3x^3) * 3x^4$ określana wielomianem st. 15



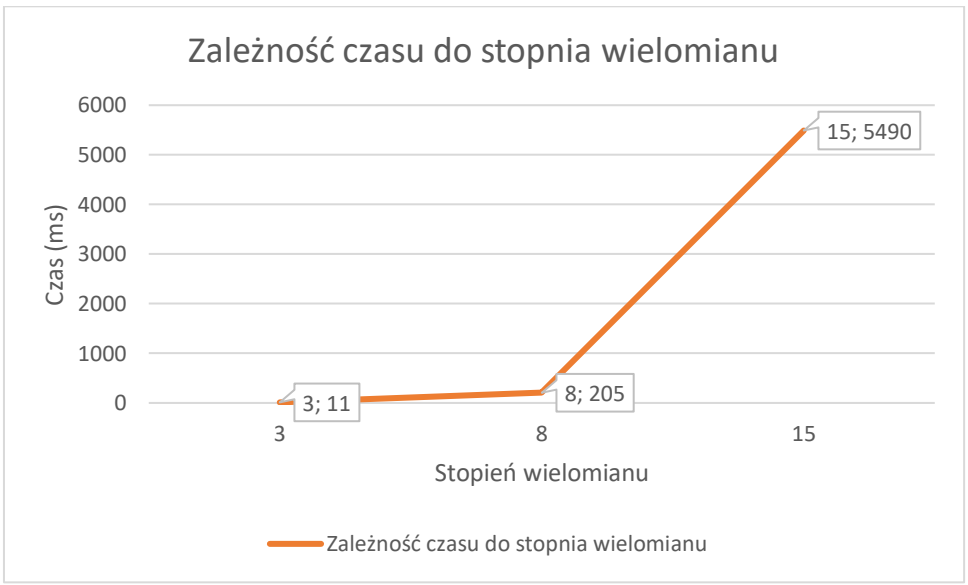
Algorytm wykonywał się 5412ms

Podsumowanie dla aproksymacji średniokwadratowej

Wraz z zwiększaniem się stopnia wielomianu określającego funkcję na przedziale < -1,1> rośnie liczba rekurencji dlatego czas wykonywania algorytmu drastycznie rośnie

a. Funkcja (x*(x + 1) - 0.1)

b. Funkcja sin(3x)*2cos((4/5)x)



c. Funkcja $(3x + 3x^2 + 3x^3) * 3x^4$

4. Aproksymacja średniokwadratowa – wnioski

Jest użyteczna jeśli potrzebujemy znaleźć krzywe, które możliwie jak najbliżej przybliżają punkty (np. w doświadczeniach). Często wykorzystuje się ją, gdy znamy funkcję ale potrzebujemy znaleźć prostszą funkcję (np. wielomian), której można użyć do określenia przybliżonych wartości funkcji danej.

5. Metoda Brenta – teoria

Metoda Brenta to metoda znajdowania pierwiastków, która łączy metodę bisekcji, metodę siecznych oraz odwrotną interpolację kwadratową. Ma niezawodność bisekcji, ale może być tak szybka jak te mniej wiarygodne metody. Algorytm próbuje użyć potencjalnie szybko zbieżnych metod jak metodę siecznych czy odwrotną interpolację kwadratową jeśli to możliwe, ale wraca do bardziej solidnej metody bisekcji jeśli to konieczne. Metodę tę wymyślił Richard Brent opierając się na wcześniejszym algorytmie Theodorusa Dekkera.

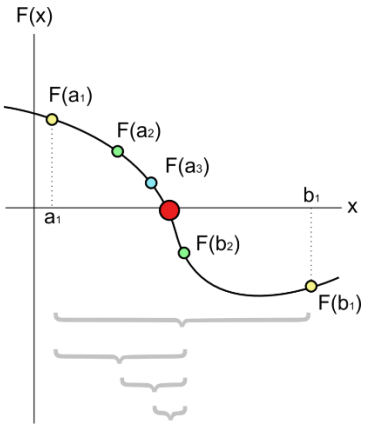
Metoda bisekcji

Metoda bisiekcii to metoda połowienia przedziału jest jedną z metod rozwiązywania równań nieliniowych, opiera się na twierdzeniu Bolzana-Cauchy'ego:

Jeżeli funkcja ciągła f(x) ma na końcach przedziału domkniętego wartości różnych znaków, to wewnątrz tego przedziału, istnieje co najmniej jeden pierwiastek równania f(x)=0.

Aby można zastosować metodę muszą zostać spełnione założenia:

1. *Funkcja f(x) jest ciągła w przedziale domkniętym [a,b]*
2. *Funkcja przyjmuje różne znaki na końcach przedziału: f(a)f(b) <0;*



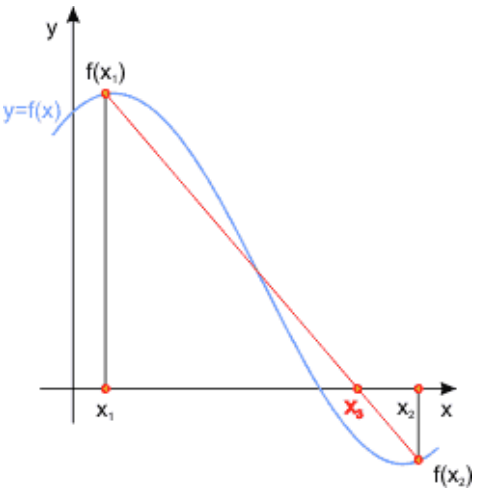
Metoda siecznych

Metoda służąca do rozwiązywania równania nieliniowego z jedną niewiadomą, jest to algorytm interpolacji liniowej, polega na przyjęciu, że funkcja ciągła na dostatecznie małym odcinku w przybliżeniu zmienia w się sposób liniowy. Możemy wtedy na odcinku <a,b> krzywą y=f(x) zastąpić sieczną. Za przybliżoną wartość pierwiastka przyjmujemy punkt przecięcia siecznej z osią OX.

Metodę siecznych dla funkcji f(x), mającej pierwiastek w przedziale <a,b> można zapisać następującym wzorem iteracyjnym:

$$\begin{cases} x_0 = a \\ x_1 = b \\ x_{n+1} = \frac{f(x_n)x_{n-1} - f(x_{n-1})x_n}{f(x_n) - f(x_{n-1})} \end{cases}$$

Metoda siecznych ma tę zaletę, że do wykonania interpolacji za jej pomocą niepotrzebna jest znajomość pochodnej danej funkcji, gdyż przybliżamy ją za pomocą powyższego wzoru.



Odwrotna interpolacja kwadratowa

Odwrotna interpolacja kwadratowa jest metodą znajdowania pierwiastków, to znaczy jest algorytmem rozwiązywania równań postaci f(x) = 0. Pomysłem jest użycie interpolacji

kwadratowej do aproksymacji funkcji odwrotnej do f. Algorytm jest definiowany przez równanie rekurencyjne:

$$x_{n+1} = \frac{f_{n-1}f_n}{(f_{n-2}-f_{n-1})(f_{n-2}-f_n)}x_{n-2} + \frac{f_{n-2}f_n}{(f_{n-1}-f_{n-2})(f_{n-1}-f_n)}x_{n-1} + \frac{f_{n-2}f_{n-1}}{(f_n-f_{n-2})(f_n-f_{n-1})}x_n,$$

gdzie $f_k = f(x_k)$. Jak można zauważyć z zależności rekurencyjnej, ta metoda wymaga trzech początkowych wartości: x_0, x_1 i x_2.

Metoda Dekkera na, której bazuje metoda Brenta dobrze spisuje jeśli funkcja f stosunkowo dobrze się zachowuje, jednak czasem istnieją okoliczności, w których iteracja wykorzystuje metodę siecznych, ale iteracje zbiegają bardzo powoli, w tym przypadku metoda Dekkera wymaga więcej iteracji niż metoda bisekcji, dlatego pojawił się dodatkowy warunek który musi być spełniony, aby w następnej iteracji zaakceptowano rezultat metody siecznych:

$$|\delta| < |b_k - b_{k-1}|$$

oraz

$$|s - b_k| < \tfrac{1}{2}|b_k - b_{k-1}|$$

Powoduje to pewność, że na k-tej iteracji krok bisekcji zostanie wykonany w najwyżej dodatkowych iteracjach ponieważ powyższy warunek zmusza rozmiar kolejnego kroku interpolacji do połowienia się przy każdych dwóch iteracjach i po najwyżej iteracjach wielkość kroku będzie mniejsza niż które woła krok bisekcji.

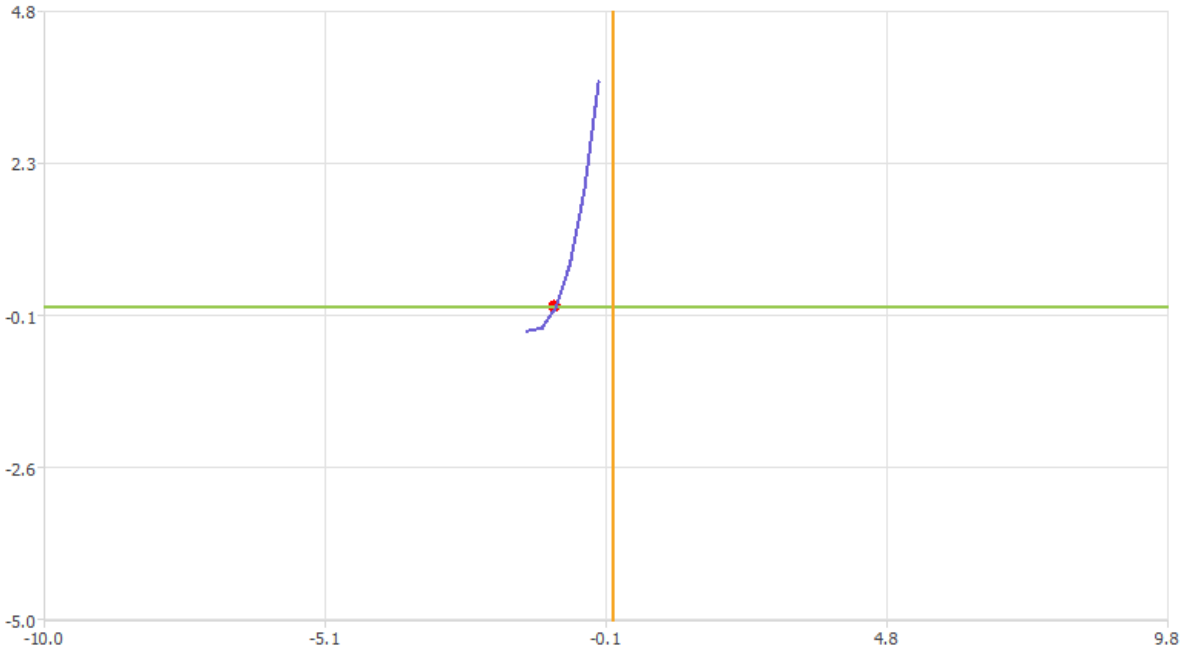
Brent dowiódł, że jego metoda wymaga co najwyżej N^2 iteracji gdzie N oznacza ilość iteracji bisekcji.

Implementacja w C++

```
double brents_fun(double dolna_granica, double gorna_granica, double TOL, double MAX_ITER)
{
    double a = dolna_granica;
    double b = gorna_granica;
    double fa = f(a);   // obliczenie funkcji dla a
    double fb = f(b);   // obliczenie funkcji dla b
    double fs = 0;

    if (!(fa * fb < 0))
    {
        std::cout << "Wartość f(dolna_granica) i f(gorna_granica) muszą być przeciwnych znaków" << std::endl;
        return 0;
    }

    if (std::abs(fa) < std::abs(b)) // jesli wartosc fa jest mniejsza od gornej granicy
    {
        std::swap(a,b);
        std::swap(fa,fb);
    }
```

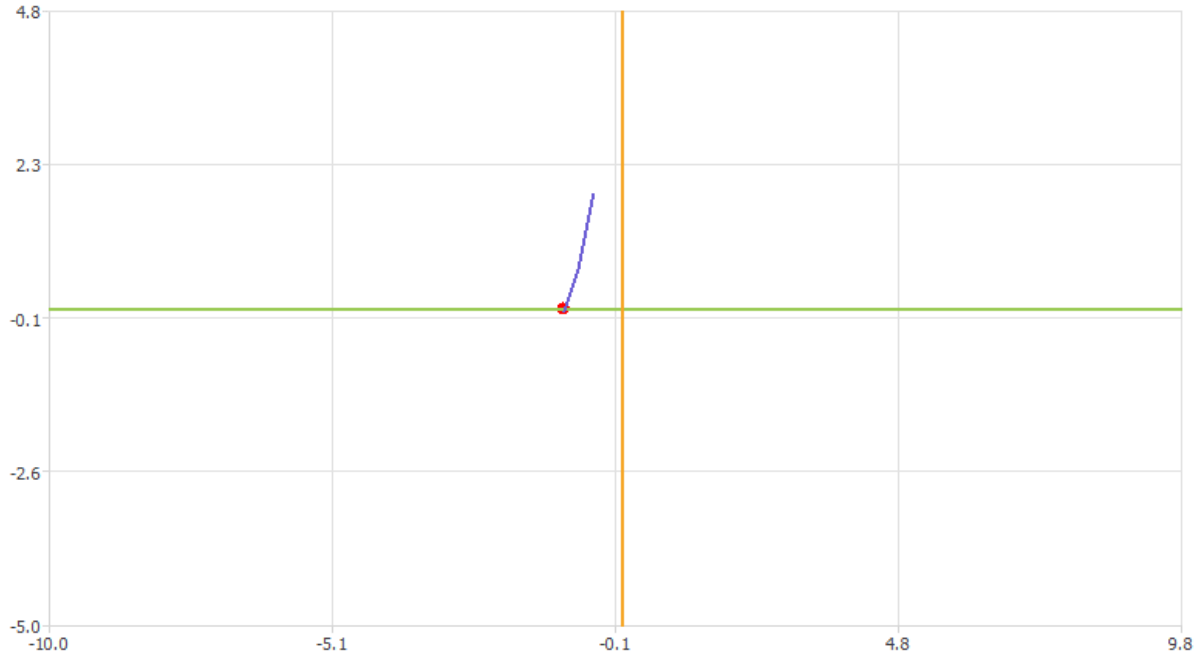
```
    double c = a;
    double fc = fa;
    bool mflag = true;      // uzywam do oceny warunkow
    double s = 0;           // zwracana wartosc
    double d = 0;           // uzywam jesli mglaga jest nieustawiona

    for (unsigned int iter = 1; iter < MAX_ITER; ++iter)
    {

        if (std::abs(b-a) < TOL)
        {
            std::cout << "Po " << iter << " iteracjach pierwiastek to: " << s << std::endl;
            return s;
        }

        if (fa != fc && fb != fc)                                   ⚠ comparing floating point w
        {
            // metoda interpolacji odwrotnej
            s =   ( a * fb * fc / ((fa - fb) * (fa - fc)) )
                + ( b * fa * fc / ((fb - fa) * (fb - fc)) )
                + ( c * fa * fb / ((fc - fa) * (fc - fb)) );
        }
        else
        {
            // metoda siecznych
            s = b - fb * (b - a) / (fb - fa);
        }

          if (    ( (s < (3 * a + b) * 0.25) || (s > b) ) ||
                  ( mflag && (std::abs(s-b) >= (std::abs(b-c) * 0.5)) ) ||
                  ( !mflag && (std::abs(s-b) >= (std::abs(c-d) * 0.5)) ) ||
                  ( mflag && (std::abs(b-c) < TOL) ) ||
                  ( !mflag && (std::abs(c-d) < TOL))  )
          {
              //metoda bisekcji
              s = (a+b)*0.5;
              mflag = true;
          }
          else
          {
              mflag = false;
          }

          fs = f(s);
          d = c;
          c = b;
          fc = fb;

          if ( fa * fs < 0)   // fa i fs maja przeciwny znak
          {
              b = s;
              fb = fs;    // jesli tak
          }
          else
          {
              a = s;
              fa = fs;    // jesli nie
          }

          if (std::abs(fa) < std::abs(fb)) // jesli wartosc fa jest mniejsza od wartosc fb
          {
              std::swap(a,b);
              std::swap(fa,fb);
          }

      }

      std::cout<< "Rozwiązanie nie jest zbieżne lub przekroczono ilosc iteracji" << std::endl;

}
```

6. Metoda Brenta – testy
    a. Funkcja (x+1)(x+2)(x+3) na przedziale a=-1,5; b=0



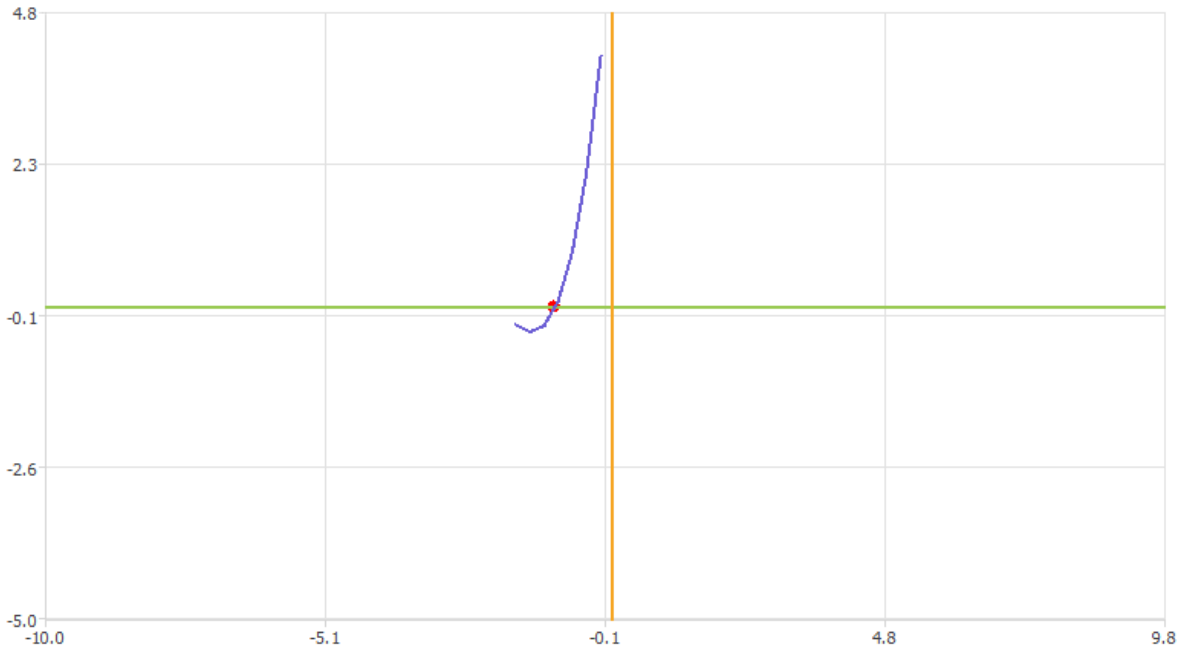
Obliczono pierwiastek -1 po 14 iteracjach zajęło to 79ms

    b. Funkcja (x+1)(x+2)(x+3) na przedziale a=-1,01; b=-0,3



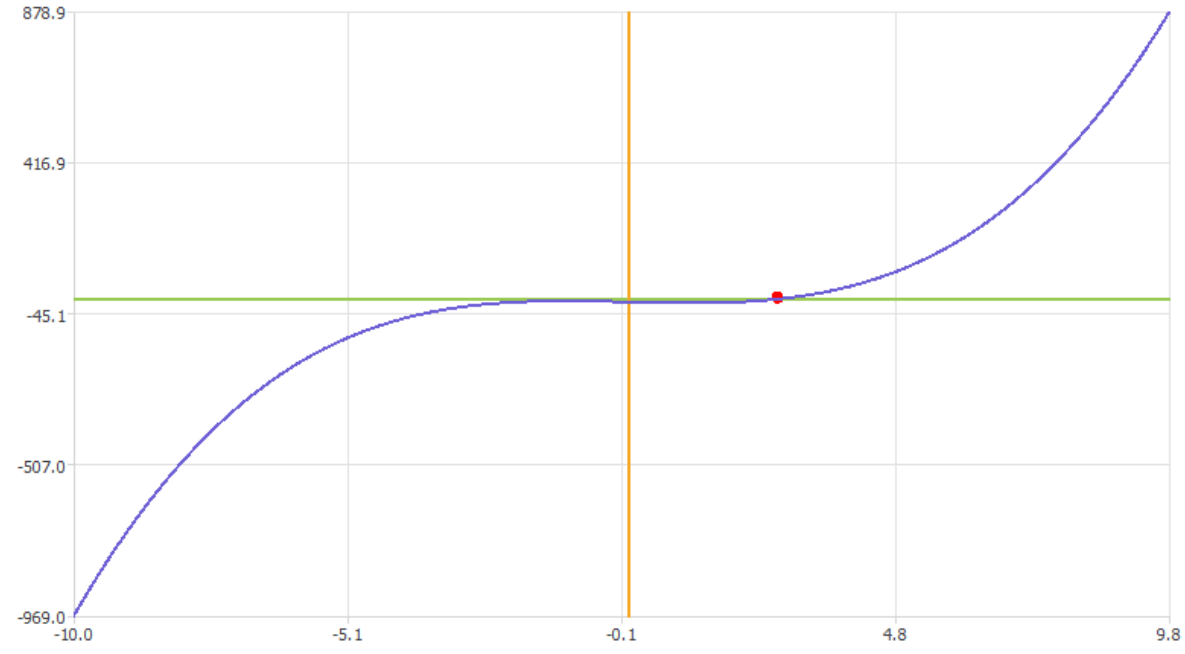
Obliczono pierwiastek -0,9999947 po 12 iteracjach i zajęło to 77ms

c. Funkcja (x+1)(x+2)(x+3) na przedziale a-1,01; b=-0,099 => wartości skrajne



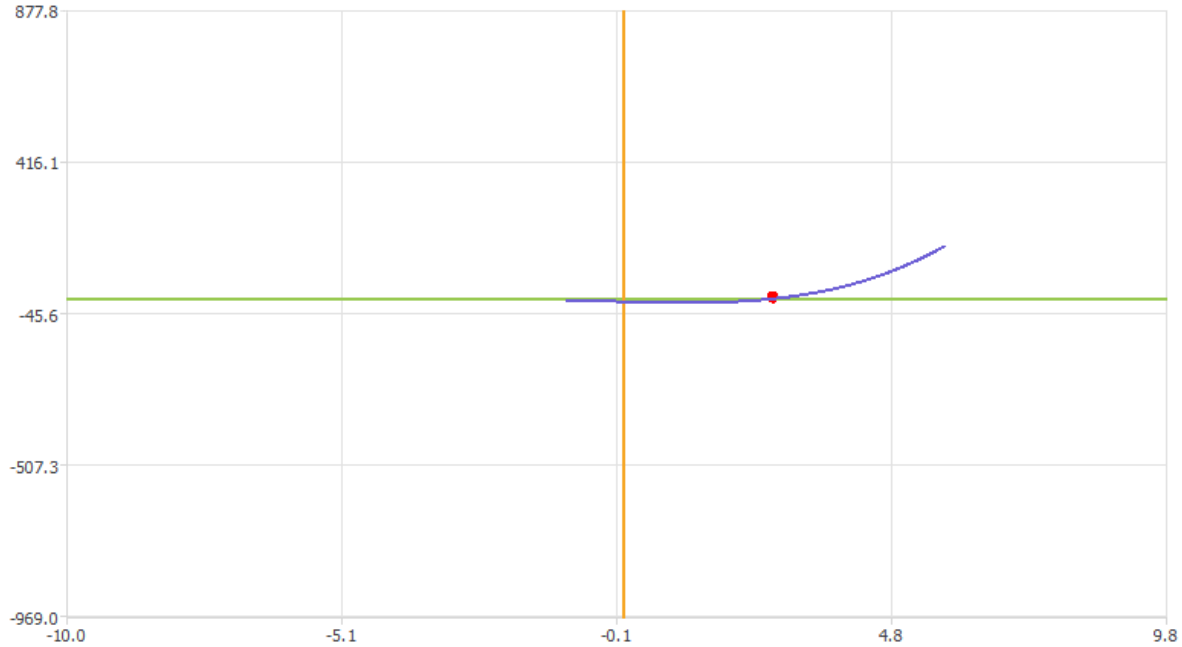
Obliczono pierwiastek -1 po 3 iteracjach zajęło to 69 ms

d. Funkcja (x^3-4x-9) na przedziale a=-10; b=10



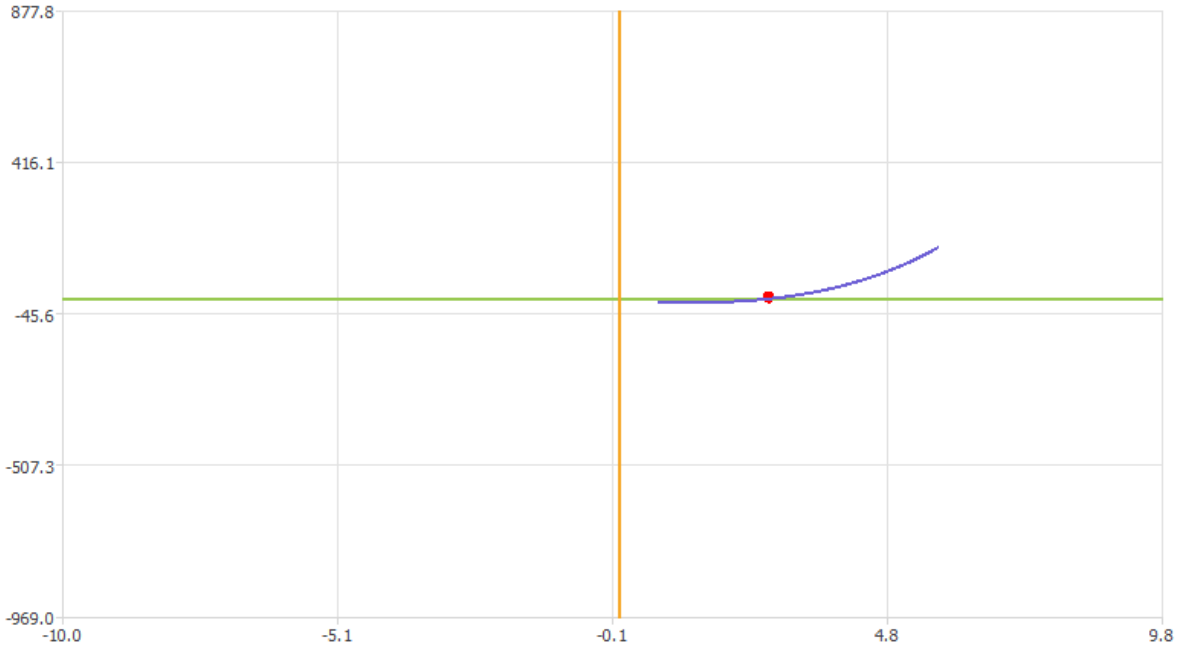
Obliczono pierwiastek 2.70653 po 14 iteracjach zajęło to 70 ms

e. Funkcja (x^3-4x-9) na przedziale a=1; b=3



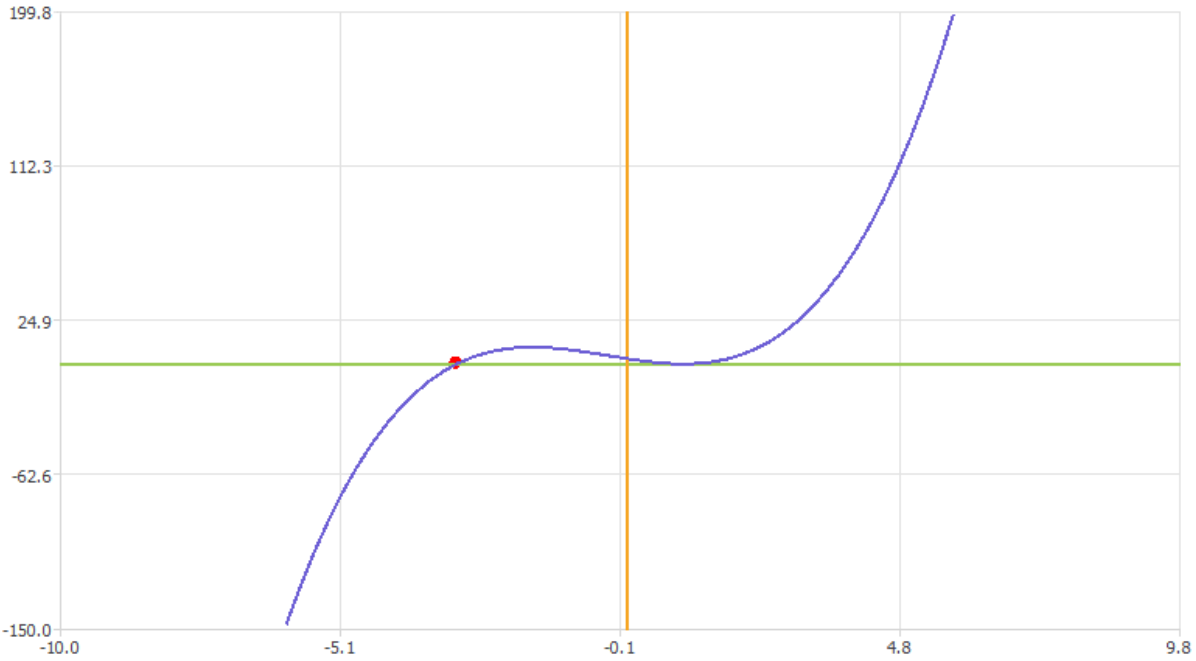
Obliczono pierwiastek 2.70655 po 12 iteracjach zajęło to 82ms

f. Funkcja (x^3-4x-9) na przedziale a=2.706; b=2.707 =>wartości skrajne



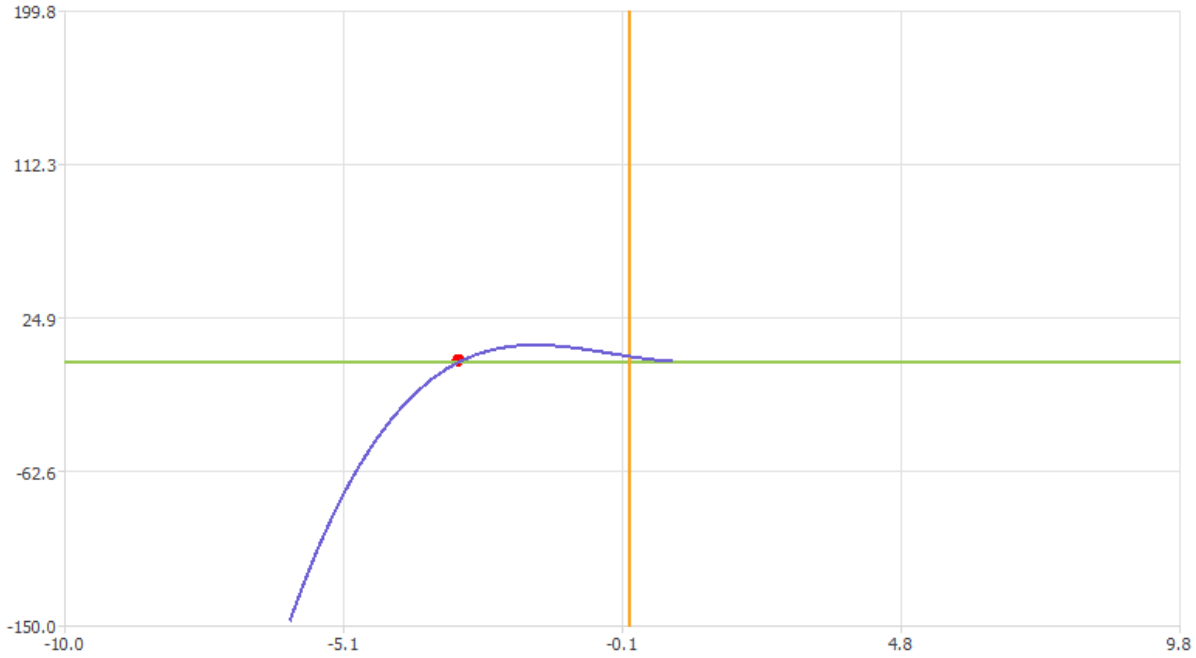
Obliczono pierwiastek -2.70656 po 5 iteracjach zajęło to 63 ms

g. Funkcja (x+3)*(x-1)*(x-1) na przedziale a=-4; b=3



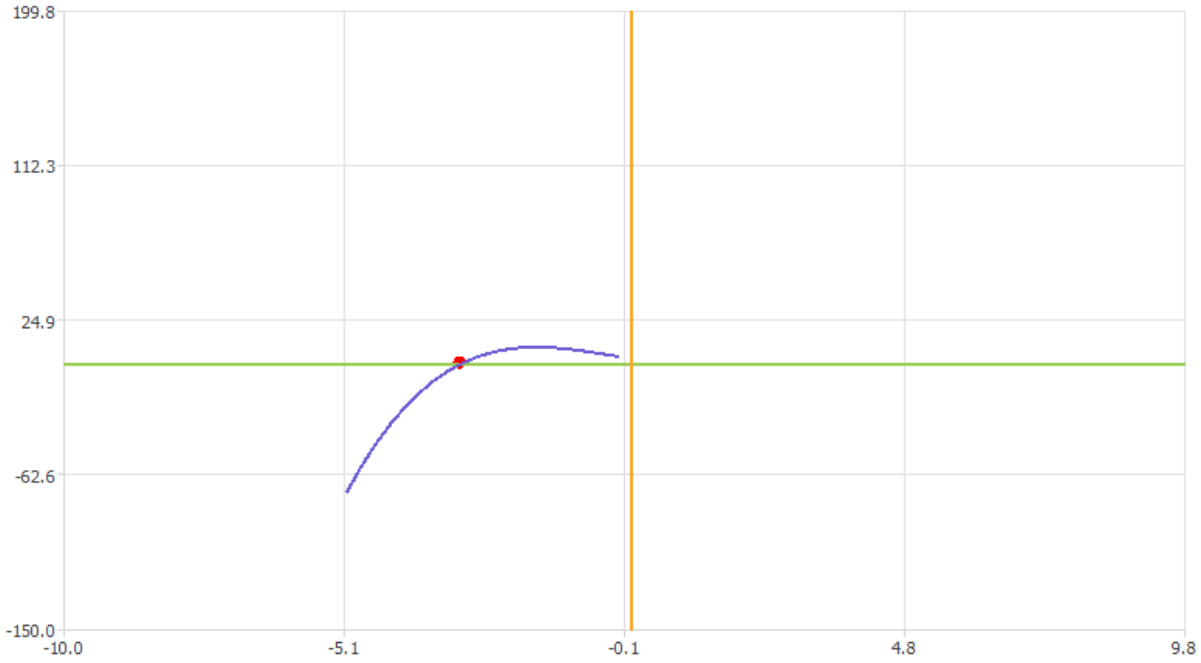
Obliczono pierwiastek -3 po 13 iteracjach zajęło to 71 ms

h. Funkcja (x+3)*(x-1)*(x-1) na przedziale a=-4; b=-2



Obliczono pierwiastek -3 po 10 iteracjach zajęło to 70 ms

i. Funkcja (x+3)*(x-1)*(x-1) na przedziale a=-2.999; b=-3.0001

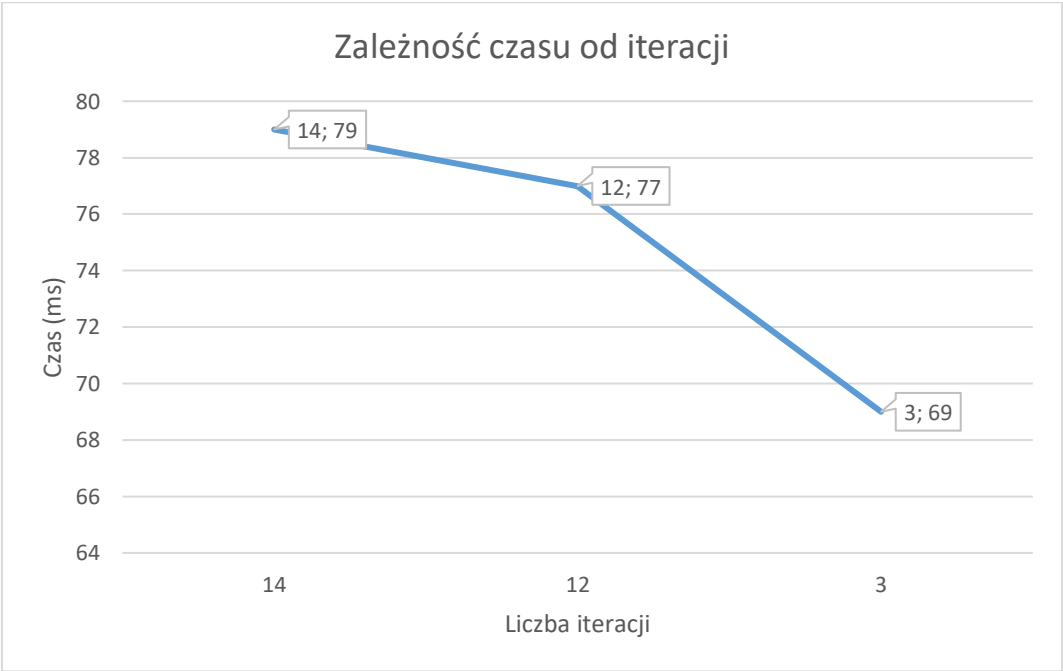


Obliczono pierwiastek -3 po 6 iteracjach zajęło to 64ms

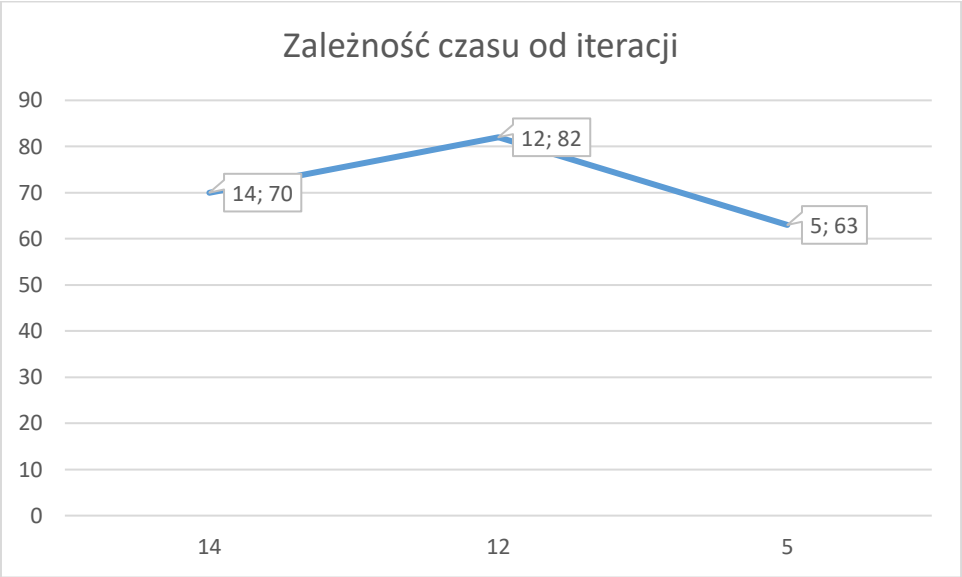
Podsumowanie dla metody Brenta

Liczba iteracji spada wraz z zmniejszaniem się zakresu szukania miejsca zerowego a co za tym idzie skraca się czas wykonywanego algorytmu

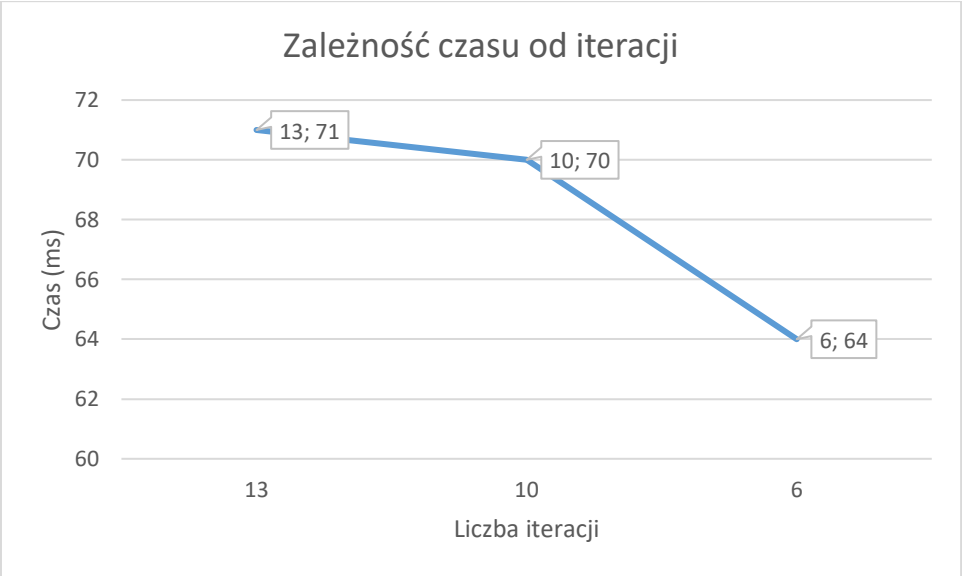
a. Funkcja (x+1)(x+2)(x+3)



b. Funkcja (x^3-4x-9)



c. Funkcja (x+3)*(x-1)*(x-1)



d. Metoda Brenta –wnioski

Metoda Brenta zachowuje się znacznie lepiej niż metoda Dekkera (algorytm A), jednak przewyższa ją metoda Dekkera (algorytm R).

Użycie odwrotnej interpolacji kwadratowej zamiast liniowej interpolacji nieznacznie zwiększa wydajność algorytmu.