Metody Obliczeniowe w Nauce i Technice

Interpolacja

Oscar Teeninga

Wielomian interpolacyjny Lagrange'a

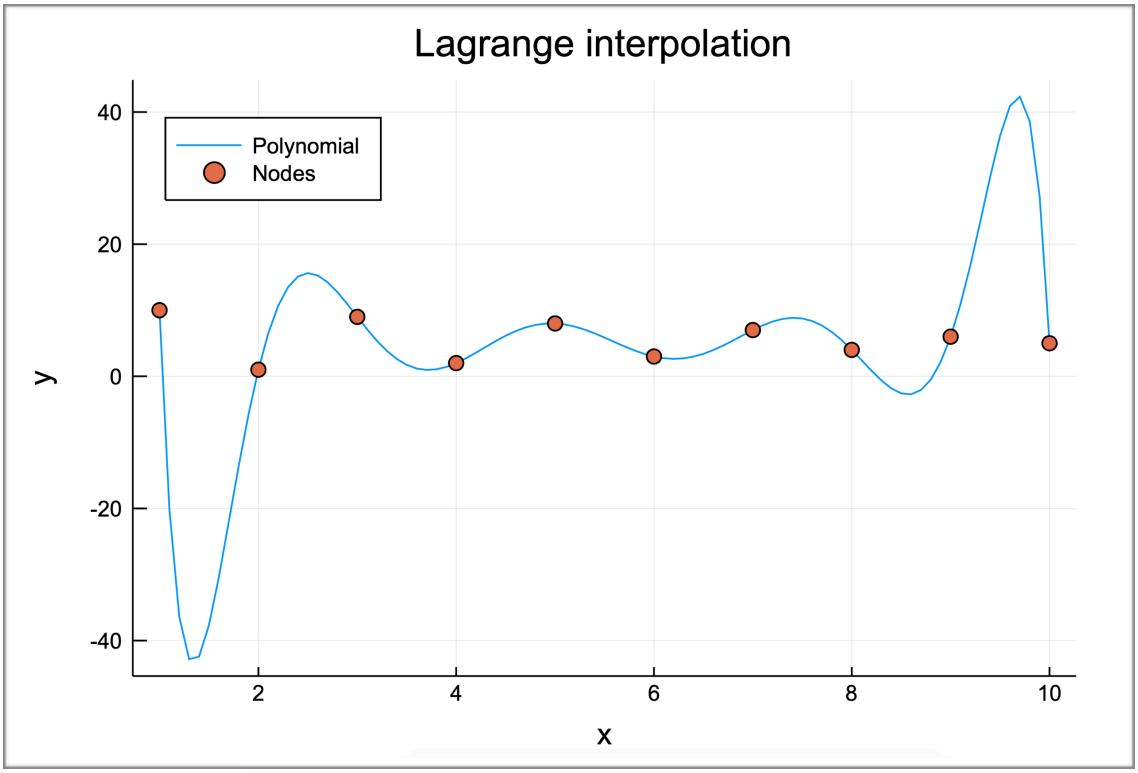
Algorytm zaimplementowałem w języku Julia. Jest to najprostsza możliwa wersja obliczania wartości funkcji interpolującej w każdym punkcie. Złożoność algorytmu to O(n^2*m), gdzie n - ilość węzłów, m - ilość punktów funkcji interpolującej.

```
function Lagrange(x, xval, yval)
    if (length(xval) != length(yval))
        return Nothing
    end
    size = length(xval)
    range = length(x)
    result = zeros(range)
    for k = 1:range
        value = 0
        for i = 1:size
            a = 1
            for j = 1:size
                if (j != i)
                    a *= (x[k] - xval[j])/(xval[i] - xval[j])
                end
            end
            value += a*yval[i]
        end
        result[k] = value
    end
    return result
end

xval = [1,2,3,4,5,6,7,8,9,10]
yval = [10,1,9,2,8,3,7,4,6,5]
```

Kod 1: Interpolacja Lagrange'a

Dla przykładowych danych wygenerowałem wykres z naniesionymi węzłami. Jak widać funkcja poprawnie generuje wartości funkcji interpolującej



Wykres 1: Interpolacja Lagrange'a

Metoda ilorazów różnicowych

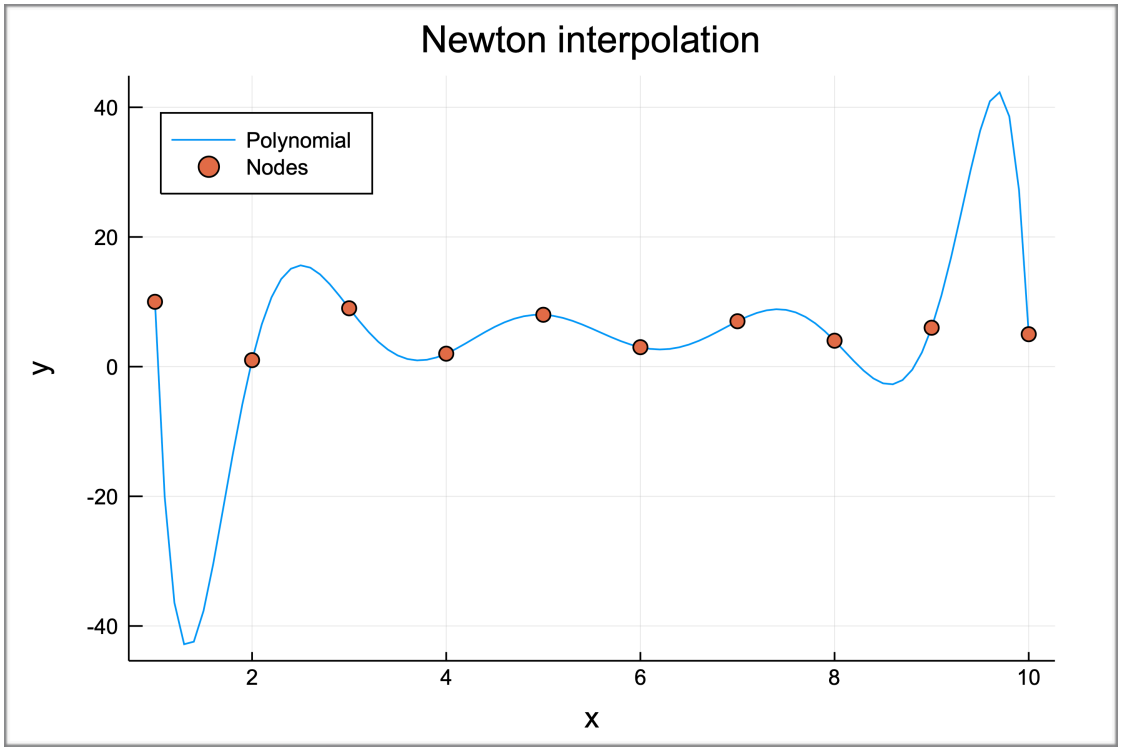
Algorytm zaimplementowałem w języku Julia. W przeciwieństwie do poprzedniego algorytmu, tym razem generujemy jednorazowo odpowiednie współczynniki, a następnie wyliczamy wartości funkcji interpolującej za pomocą algorytmu Hornera co również przyśpiesza obliczenia. Pozwala to osiągnąć znacząco lepszą złożoność O(n^2 + m).

```
function Horner(a, xval, x, size)
    fx = a[size]
    k = size-1
    while k > 0
        fx = fx * (x - xval[k]) + a[k]
        k -= 1
    end
    return fx
end

function Newton(x, xval, yval)
    if (length(xval) != length(yval))
        return Nothing
    end
    size = length(xval)
    range = length(x)
    result = zeros(range)
    a = zeros(size)
    r = zeros(size)
    for i = 1:size
        r[i] = yval[i]
        j = i-1
        while j > 0
            r[j] = (r[j+1]-r[j])/(xval[i]-xval[j])
            j -= 1
        end
        a[i] = r[1]
    end
    for k = 1:range
        result[k] = Horner(a, xval, x[k], size)
    end
    return result
end
```

Kod 2: Interpolacja Newtona + algorytm Hornera

Dla tych samych danych jak uprzednio wygenerowałem funkcję interpolującą. Również widać, że funkcja została wygenerowana poprawnie.



Wykres 2: Interpolacja Newtona

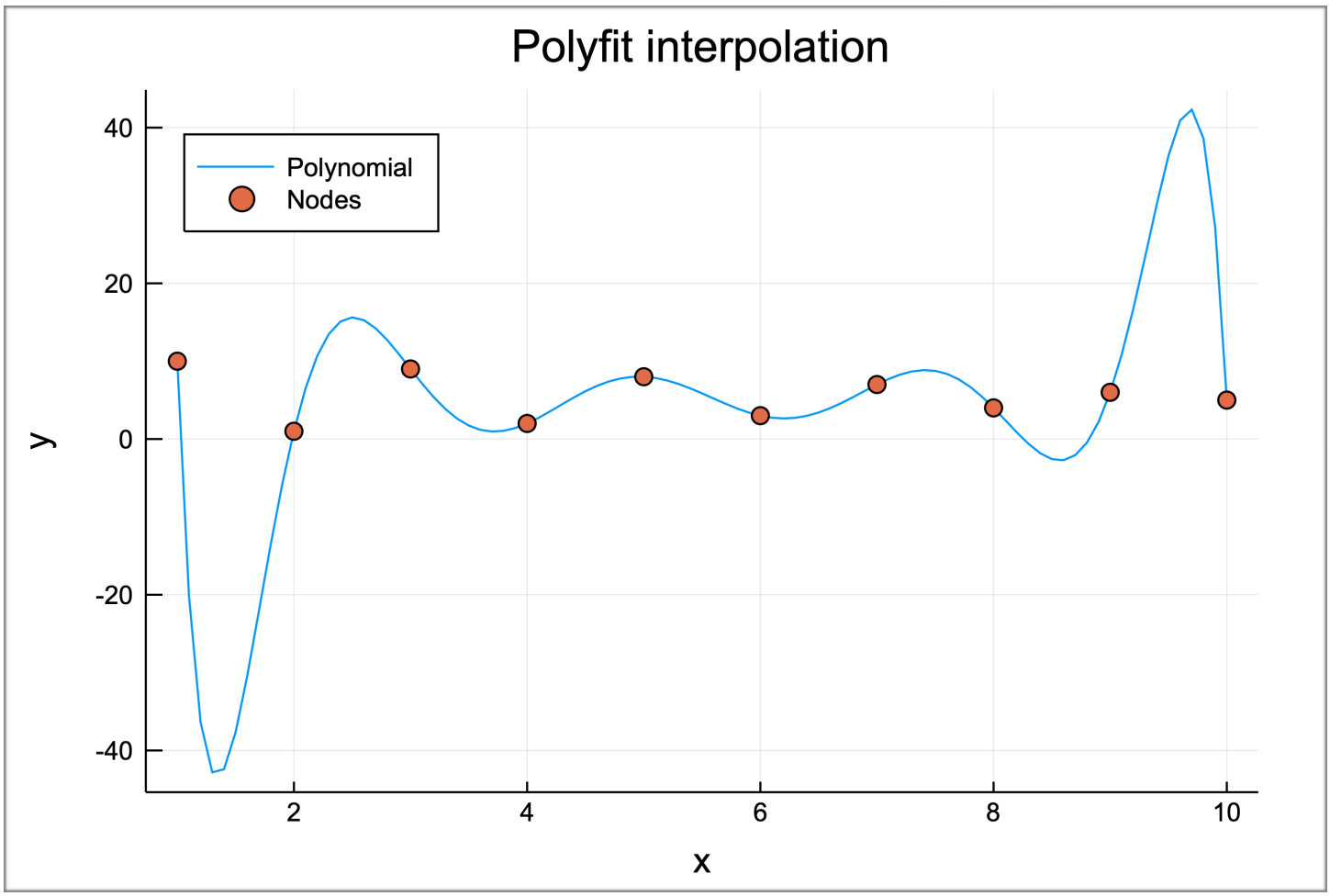
Interpolacja za pomocą funkcji **polyfit**

Implementując metodę skorzystałem z funkcji polyfit, dostarczaną z pakietem *Polynomials*. Można się spodziewać, że będzie ona najszybsza, jednak jej dokładne działanie jest nieznane. Widać jedynie, że współczynniki generowane są jednorazowo.

```
function Poly(xs, xval, yval)
    fit=polyfit(xval,yval)
    fxs=[fit(x) for x in xs]
    return fxs
end
```

Kod 3: Interpolacja polyfit

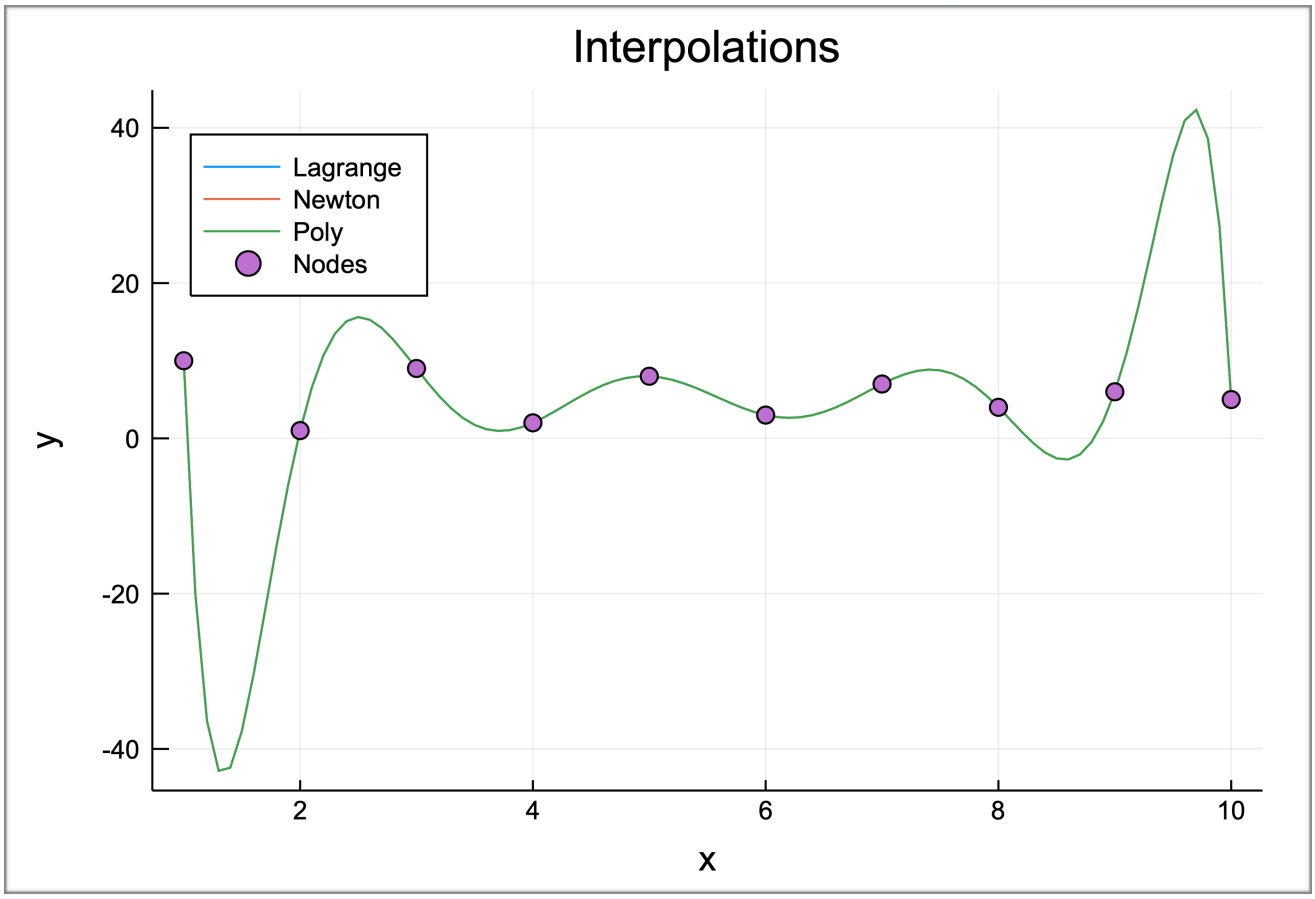
Po raz kolejny dla tych samych danych wygenerowałem wykres, ty razem za pomocą funkcji Poly. Funkcja również generuje poprawną interpolację przykładowych węzłów.



Wykres 3: Interpolacja funkcją polyfit

Porównanie algorytmów

Korzystając z tych samych przykładowych danych wygenerowałem dla każdego algorytmu własną funkcję interpolującą, a następnie nałożyłem wszystkie na ten sam wykres.



Wykres 4: Porównanie interpolacji

Widać, że wszystkie algorytmy wygenerowały identyczną funkcję interpolacyjną. Stało się tak, ponieważ zawsze istnieje dokładnie jedna funkcja interpolująca stopnia n dla n+1 węzłów. Jest to zgodne z twierdzeniem:

„Dla danych $n+1$ punktów pomiarowych, parami różnych od siebie, istnieje jedyny wielomian stopnia co najwyżej n interpolujący te punkty.”

Łatwo udowodnić, że faktycznie niemożliwe jest otrzymanie innej funkcji interpolującej. Załóżmy, że mamy dwa różne wielomiany co najwyżej n-stopnia $W_1^n(x)$, $W_2^n(x)$, gdzie każdy interpoluje dowolny zbiór $n+1$ węzłów. Tworzymy wielomian $W_3^n(x) = W_2^n(x) - W_1^n(x)$. Zauważmy, że dla każdego węzła $W_3^n(x_i) = 0$, a to oznacza, że ma co najmniej $n+1$ miejsc zerowych, będąc wielomianem co najwyżej n stopnia. Jest to sprzeczne zakładając, że $W_1^n(x) \neq W_2^n(x)$, gdyż byłoby to niezgodne z zasadniczym twierdzeniem algebry.
A więc $W_1^n(x) = W_2^n(x)$.

Pomiary czasu dla algorytmów interpolacji

Do wygenerowania pomiarów oraz wykresów skorzystałem z Julii. Zastosowałem stałą ilość punktów wyznaczających $n = 100$. Większe wartości znacząco spowalniały algorytm Lagrange'a, więc stworzyłem oddzielny test dla algorytmu Newtona i Poly.

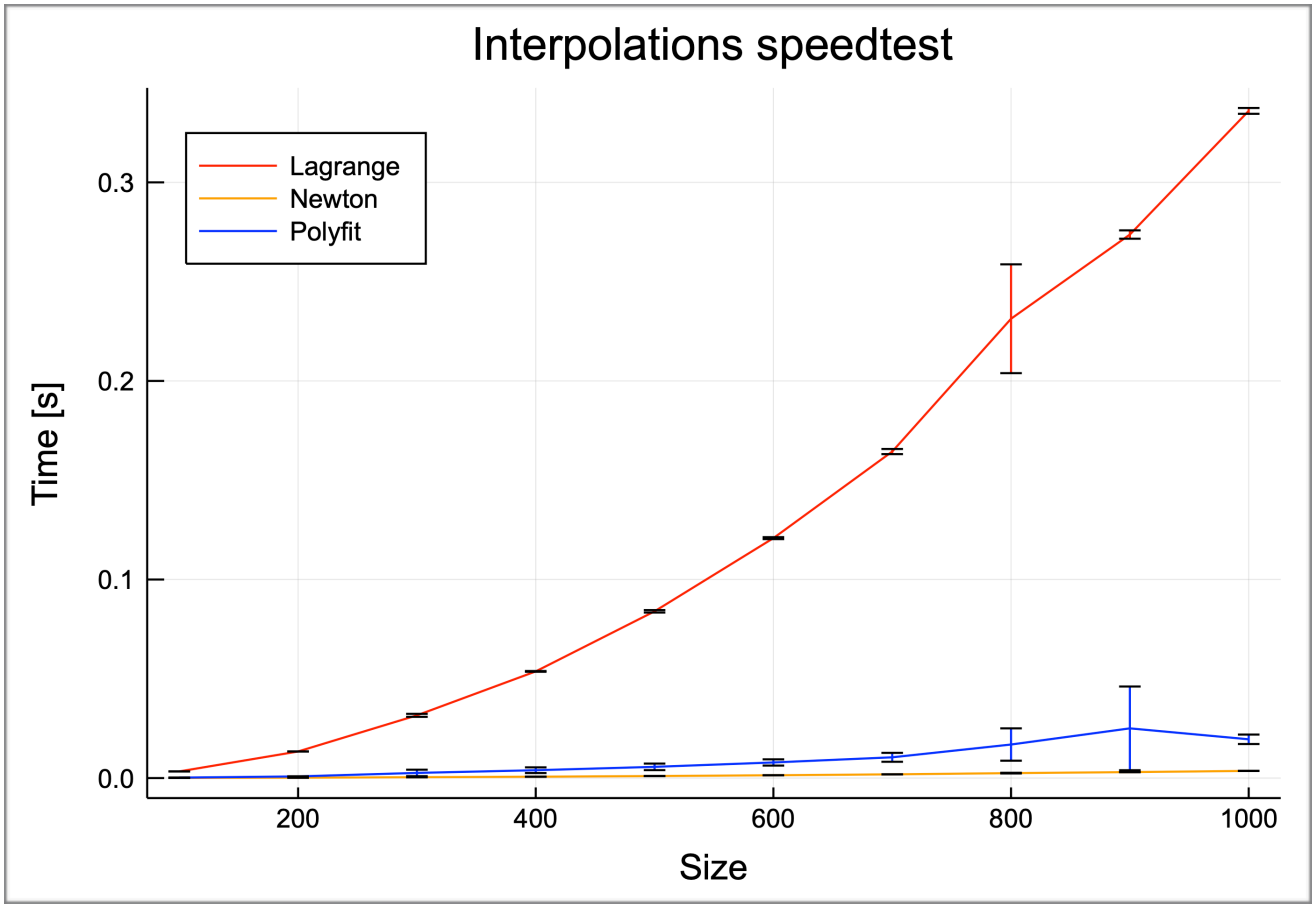
```
function gen_table(S, V)
    max = 9*V + S
    df = DataFrame(Size = [], Time_lagrange = [], Time_newton = [], Time_poly = [])
    while S <= max
        xs = 1:S
        x = 1:S/100:S
        fxs = [rand() for x in xs]
        for i = 1:10
            time1 = @elapsed Lagrange(x, xs, fxs)
            time2 = @elapsed Newton(x, xs, fxs)
            time3 = @elapsed Poly(x, xs, fxs)
            push!(df, (S, time1, time2, time3))
        end
        S += V
    end
    return df
end
```

Kod 4: Porównanie interpolacji

Skorzystałem z pakietu DataFrames i zapisałem wszystko do pliku .csv. Następnie odczytałem dane, wyliczyłem wartość średnią oraz odchylenie standardowe i narysowałem wykres.

```
df = CSV.read("times_interpolation.csv")
df_lagrange = by(df, [:Size], Time_avg = :Time_lagrange => mean, Time_std = :Time_lagrange => std)
df_newton = by(df, [:Size], Time_avg = :Time_newton => mean, Time_std = :Time_newton => std)
df_poly = by(df, [:Size], Time_avg = :Time_poly => mean, Time_std = :Time_poly => std)
```

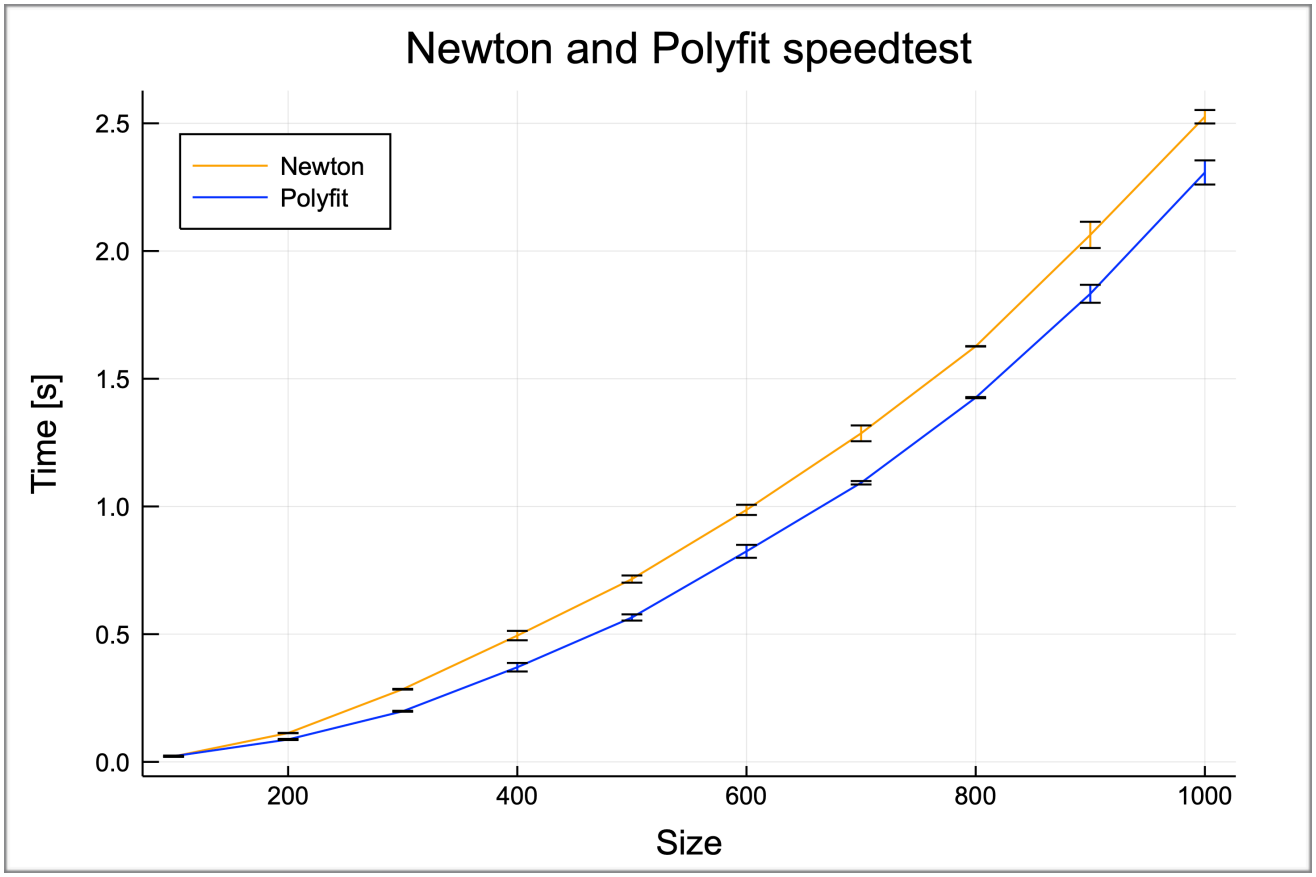
Kod 5: Załadowanie tabel do testów



Wykres 5: Porównanie szybkości działania wszystkich interpolacji

Z wykresu wynika, że algorytm Lagrange'a jest najmniej optymalny, co było oczekiwane. Natomiast nieoczekiwane był fakt, że metoda równań różnicowych okazała się być szybsza niż polyfit. Wynika to z stałego zagęszczenia punktów wyznaczających funkcję interpolującą. Dla pewności przeprowadziłem dodatkowe testy dla dwóch szybszych funkcji dla zwiększającej się gęstości punktów $n = 100S$, gdzie S to liczba węzłów(Size).

Z wykresu widać, że funkcję są tak samo szybkie, z niewielką przewagą funkcji polyfit.



Wykres 6: Porównanie szybkości działania interpolacji polyfit i Netwona

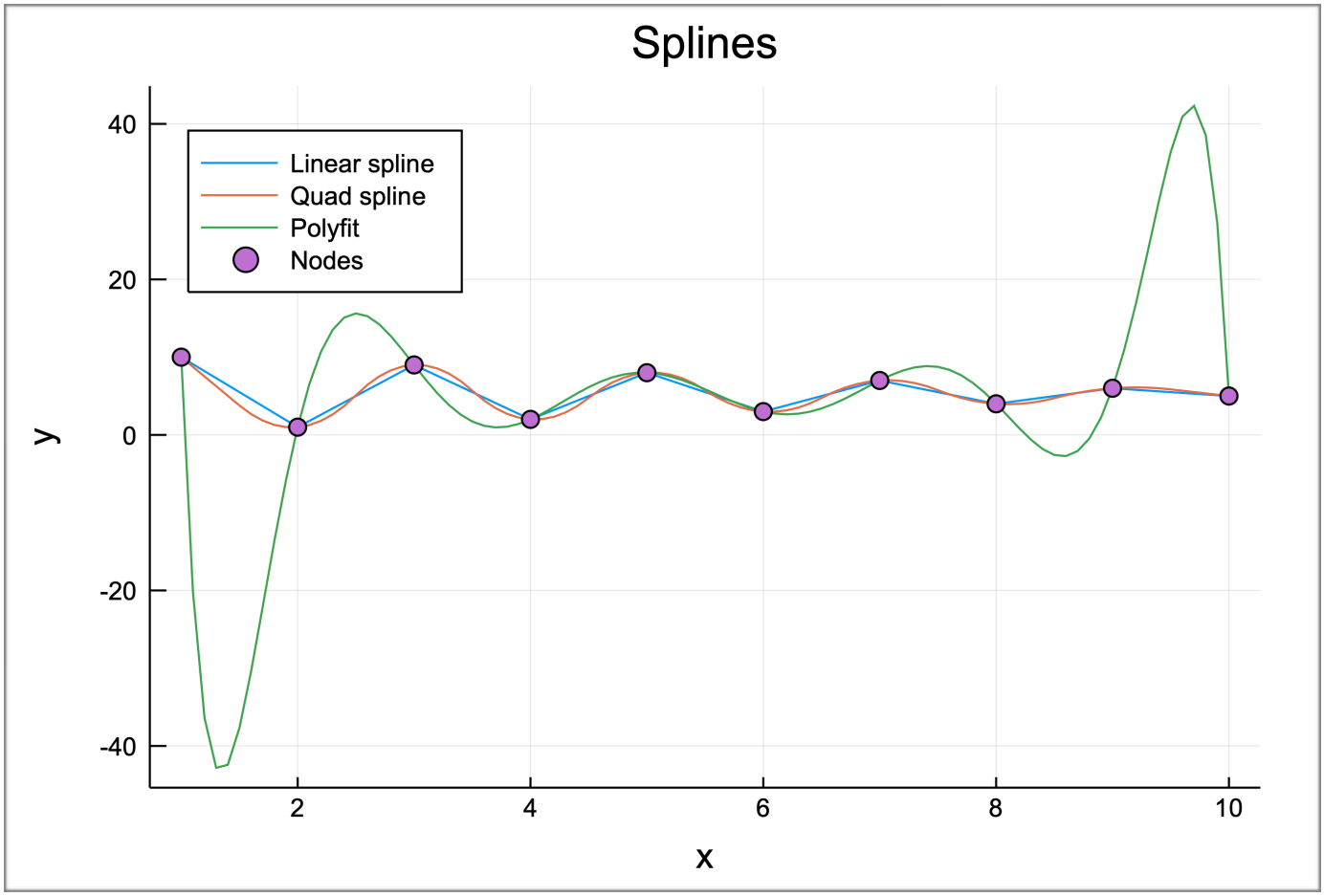
Funkcje sklejane

Ponownie skorzystałem z języka Julia oraz z pakietu *Interpolations*. Stworzyłem dwie funkcje sklejane: jedna interpoluje węzły za pomocą funkcji liniowych; druga za pomocą funkcji 4 stopnia. Skorzystałem z funkcji *LinearInterpolation(), interpolate() oraz BSpline.*

```
linear = LinearInterpolation(xval, yval)
quad = interpolate(yval, BSpline(Quadratic(Line(OnCell()))))
r_lin = [linear(x) for x in xs]
r_cub_ext = [quad(x) for x in xs]
r_poly = Poly(xs, xval, yval)
```

Kod 7: Generowanie funkcji sklejanych

Na podstawie wygenerowanych danych stworzyłem wykres z funkcji sklejanych oraz dla porównania dorysowałem funkcję wygenerowaną przez Poly. Zgodnie z oczekiwaniami, funkcje sklejane precyzyjniej i w sposób bardziej kontrolowany interpolują węzły. Sam stopień funkcji sklejanych zwiększa gładkość całej funkcji interpolującej, jednak zwiększa ilość obliczeń i zmniejsza wydajność.



Wykres 7: Porównanie funkcji sklejanych z interpolacją polyfit

Efekt Rungego

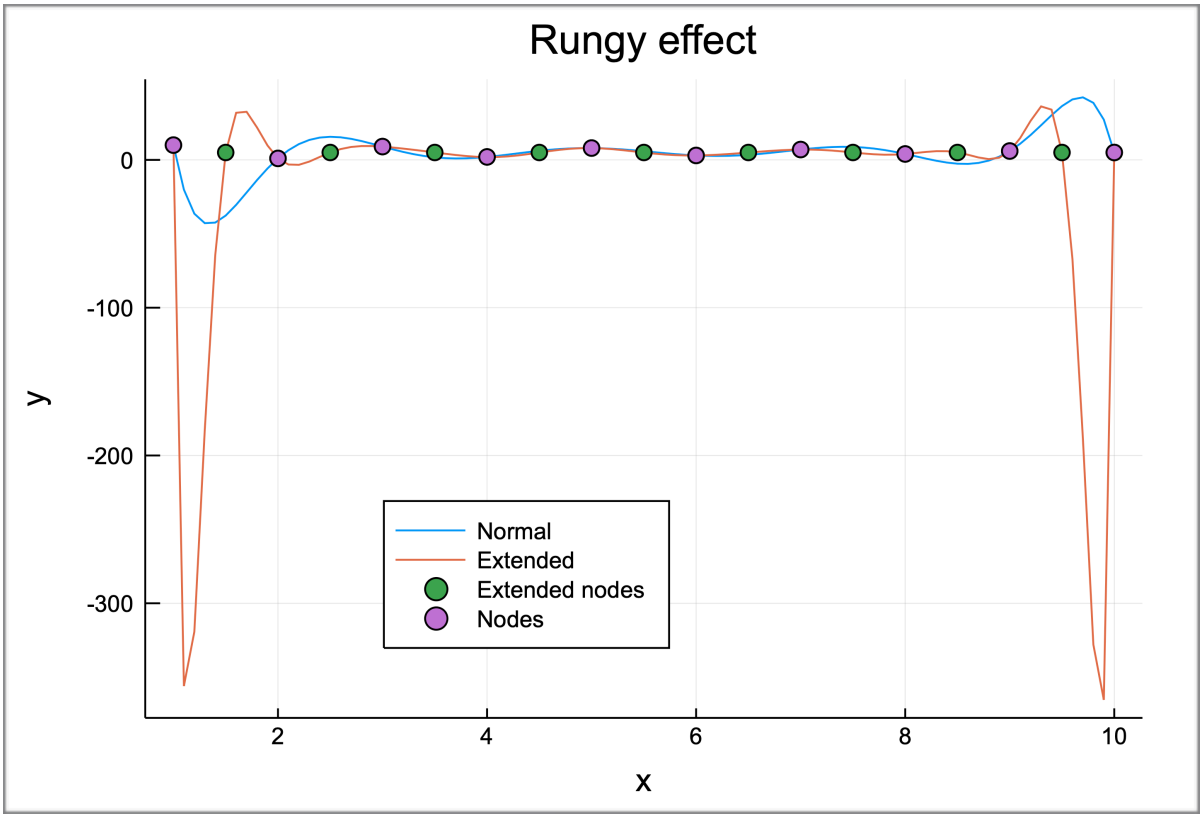
Korzystając z języka Julia stworzyłem dwa zestawy węzłów:

```
xval = [1,2,3,4,5,6,7,8,9,10]
yval = [10,1,9,2,8,3,7,4,6,5]
xval_ext = [1,1.5,2,2.5,3,3.5,4,4.5,5,5.5,6,6.5,7,7.5,8,8.5,9,9.5,10]
yval_ext = [10,5,1,5,9,5,2,5,8,5,3,5,7,5,4,5,6,5,5]
```

Kod 8: Wygenerowanie węzłów interpolacji

Drugi jest rozszerzeniem pierwszego, przy czym było starałem się rozmieścić dodatkowe węzły dokładnie pomiędzy pozostałe tworząc w ten sposób regularną siatkę węzłów oddalonych od siebie o 0.5. Każdy z nowych węzłów spełniał wzór $f(x_{2i+1}) = 5$. Taki dobór węzłów pozwolił uwypuklić problem z równomiernie zagęszczonymi węzłami nazwany *efektem Rungego:*

*„**Efekt Rungego** – pogorszenie jakości interpolacji wielomianowej, mimo zwiększenia liczby jej węzłów. Początkowo ze wzrostem liczby węzłów n przybliżenie poprawia się, jednak po dalszym wzroście n, zaczyna się pogarszać, co jest szczególnie widoczne na końcach przedziałów.” ~ Wikipedia.org*



Wykres 8: Pokazanie efektu Rungego

Zgodnie z przytoczoną definicją, zwiększając liczbę węzłów, udało się osiągnąć funkcję interpolującą mniej kontrolowalną, co jest niezgodne z intuicją. Efekt jest szczególnie widoczny na krańcach, natomiast funkcja jest precyzyjniejsza w środku, więc zagęszczanie punktów interpolacji w niektórych przypadkach może okazać się przydatne (np. analizujemy tylko „środkową” część wykresu).

Algorytmy interpolacji stosowane w grafice komputerowej

Interpolacja dwuliniowa

Metoda interpolacji dla funkcji dwóch zmiennych. Zgodnie z intuicją jest to złożenie dwóch interpolacji liniowych. Służy do skalowania obrazów graficznych. W celu wyznaczenia interpolacji dwuliniowej, należy wyznaczyć interpolacje liniowe dla każdego kierunku. Przeprowadzając najpierw interpolacje wzdłuż osi OX, otrzymujemy:

$$f(R_1) = \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \quad \text{,gdzie } R_1 = (x, y_1)$$

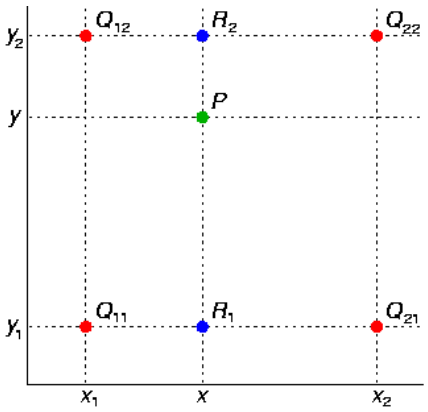
$$f(R_2) = \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \quad \text{,gdzie } R_2 = (x, y_2)$$

Następnie dla osi OY:

$$f(P) = \frac{y_2 - y}{y_2 - y_1} f(R_1) + \frac{y - y_1}{y_2 - y_1} f(R_2)$$

Zakładamy ponadto, że:

$Q_{11} = (x_1, y_1)$
$Q_{12} = (x_1, y_2)$
$Q_{21} = (x_2, y_1)$
$Q_{22} = (x_2, y_2)$



Wykres 9: Graficzne przedstawienie konceptu

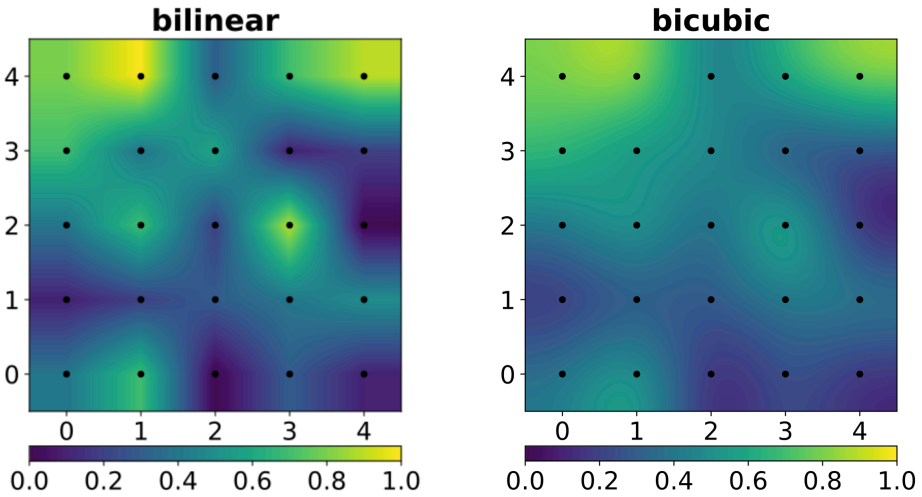
Wówczas funkcja będzie wyglądała:

$$f(x,y) = f(x_1,y_1)(x - x_2)(y - y_2) + f(x_2,y_1)(x - x_1)(y - y_2) + f(x_1,y_2)(x - x_2)(y - y_1) + f(x_2,y_2)(x - x_1)(y - y_1)$$

Za pomocą tej funkcji możemy wyznaczać dodatkowe punkty przy zwiększaniu rozdzielczości obrazu. Wartość funkcji będzie reprezentować kolor nowego piksela.
Skalowanie za pomocą interpolacji dwuliniowej jest dostępna w programie GIMP 2.10.10.

Interpolacja sześcienna

Wykorzystuje 16 pikseli sąsiadujących i opiera się na użyciu funkcji kwadratowej. Mówiąc troszkę bardziej dokładnie funkcja kwadratowa zostaje dopasowana do 8 pikseli otoczenia. Wartość przyjmowana przez piksel jest zależna od wartości pikseli w otoczeniu. Interpolacja kwadratowa powoduje najmniejszy efekt zniekształcenia i jednocześnie jest najbardziej złożona numerycznie. Poniżej porównanie:



Wykres 9,10: Porównanie jakości interpolacji stosowanej w grafice