Metody Obliczeniowe w Nauce i Technice

Interpolacja

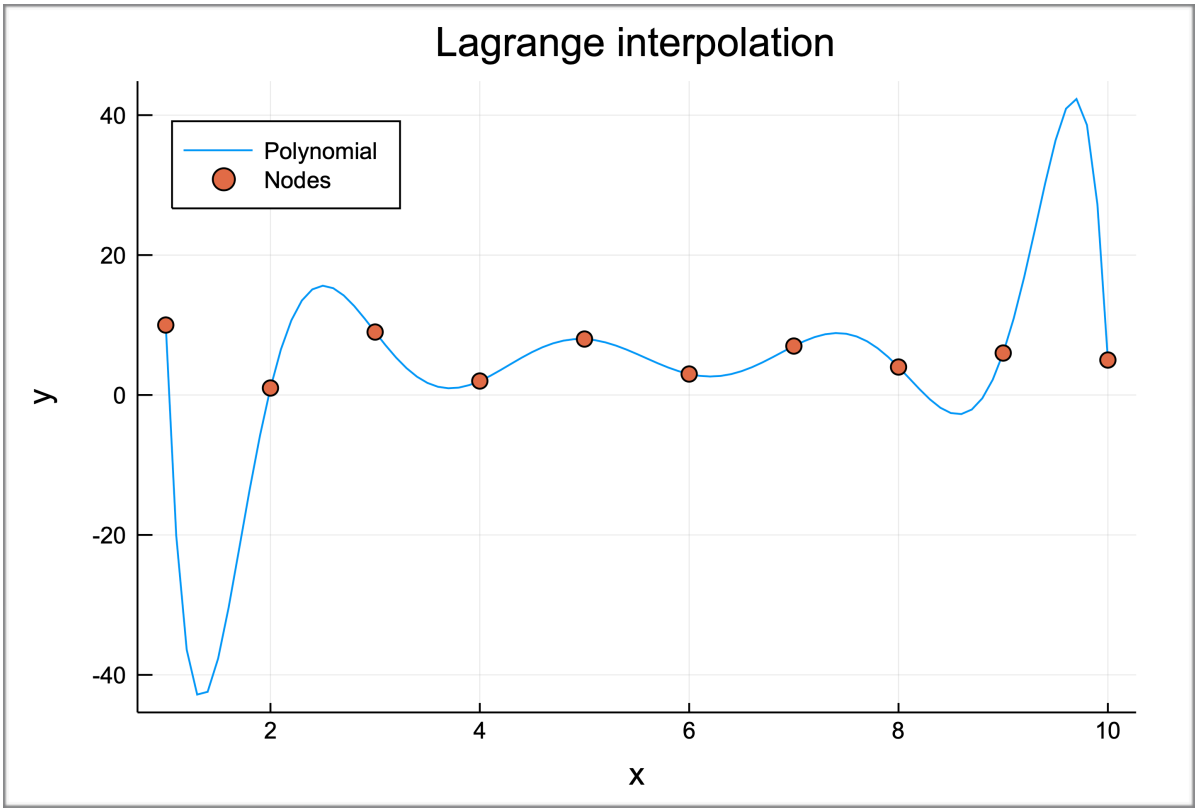
Oscar Teeninga

Wielomian interpolacyjny Lagrange'a

Algorytm zaimplementowałem w języki 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.

```
3  function Lagrange(x, xval, yval)
4      if (length(xval) != length(yval))
5          return Nothing
6      end
7      size = length(xval)
8      range = length(x)
9      result = zeros(range)
10     for k = 1:range
11         value = 0
12         for i = 1:size
13             a = 1
14             for j = 1:size
15                 if (j != i)
16                     a *= (x[k] - xval[j])/(xval[i] - xval[j])
17                 end
18             end
19             value += a*yval[i]
20         end
21         result[k] = value
22     end
23     return result
24 end
25
26 xval = [1,2,3,4,5,6,7,8,9,10]
27 yval = [10,1,9,2,8,3,7,4,6,5]
28 xs=1:0.1:10
29 fxs = Lagrange(xs,xval, yval)
```

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

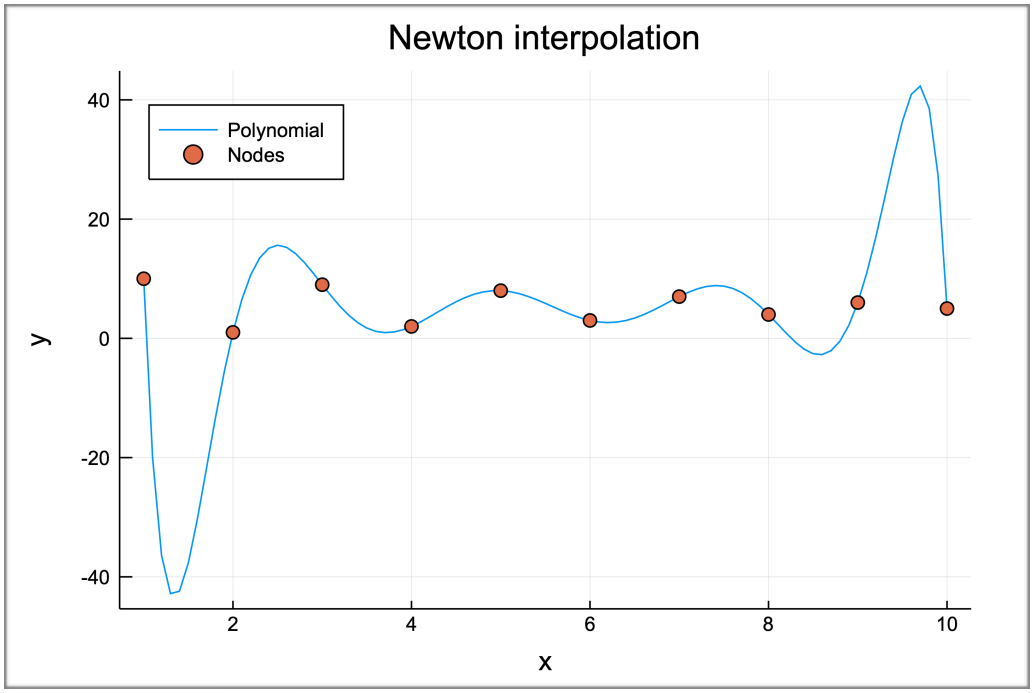


Metoda ilorazów różnicowych

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).

```
3   function Horner(a, xval, x, size)
4       fx = a[size]
5       k = size-1
6       while k > 0
7           fx = fx * (x - xval[k]) + a[k]
8           k -= 1
9       end
10      return fx
11  end
12
13  function Newton(x, xval, yval)
14      if (length(xval) != length(yval))
15          return Nothing
16      end
17      size = length(xval)
18      range = length(x)
19      result = zeros(range)
20      a = zeros(size)
21      r = zeros(size)
22      for i = 1:size
23          r[i] = yval[i]
24          j = i-1
25          while j > 0
26              r[j] = (r[j+1]-r[j])/(xval[i]-xval[j])
27              j -= 1
28          end
29          a[i] = r[1]
30      end
31      for k = 1:range
32          result[k] = Horner(a, xval, x[k], size)
33      end
34      return result
35  end
```

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

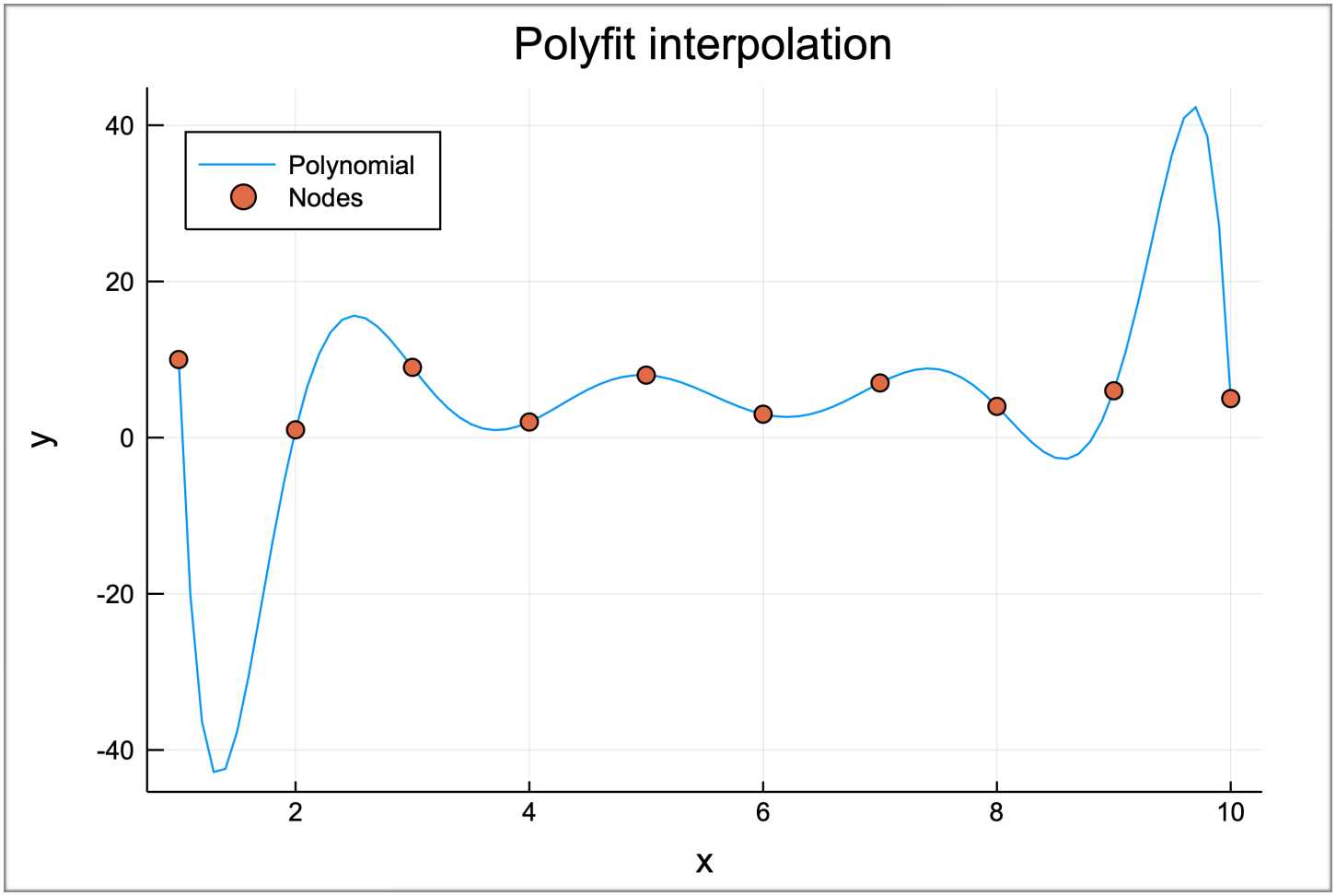


Interpolacja za pomocą funkcji **polyfit**

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.

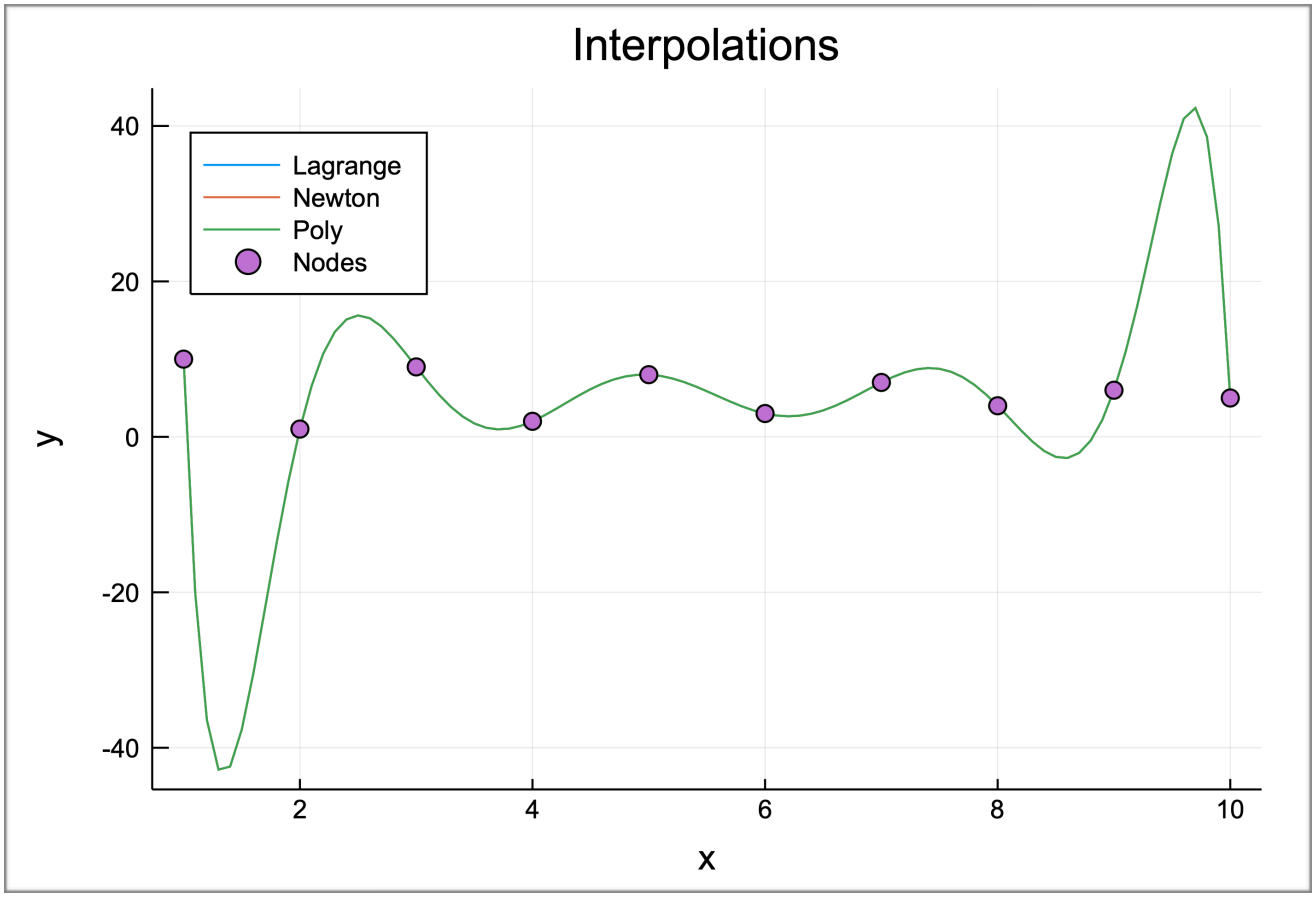
```
4   function Poly(xs, xval, yval)
5       fit=polyfit(xval,yval)
6       fxs=[fit(x) for x in xs]
7       return fxs
8   end
```

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



Porównanie algorytmów

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



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 $W_1^n(x) = W_2^n(x)$.

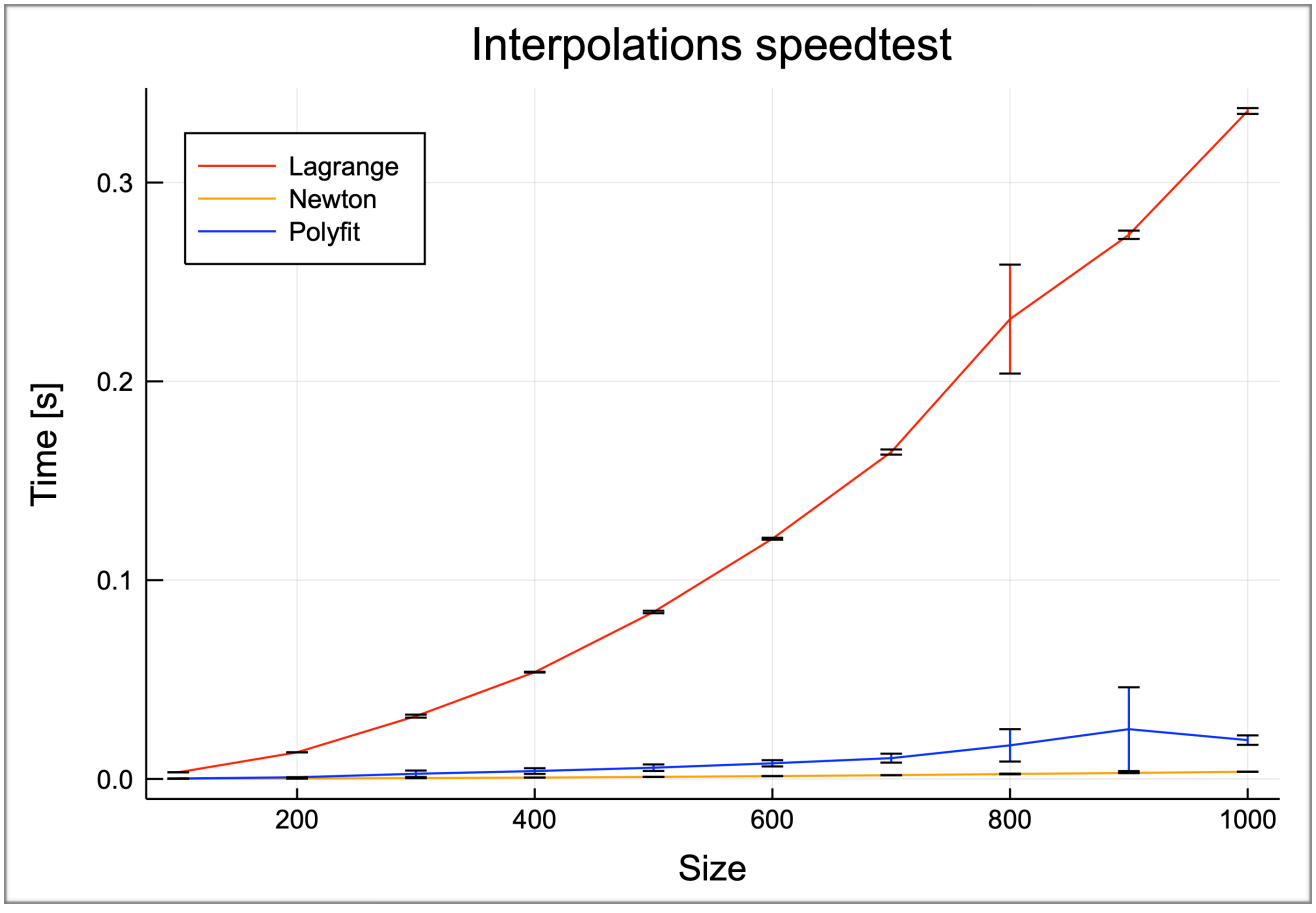
Pomiary czasu dla algorytmów interpolacji

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.

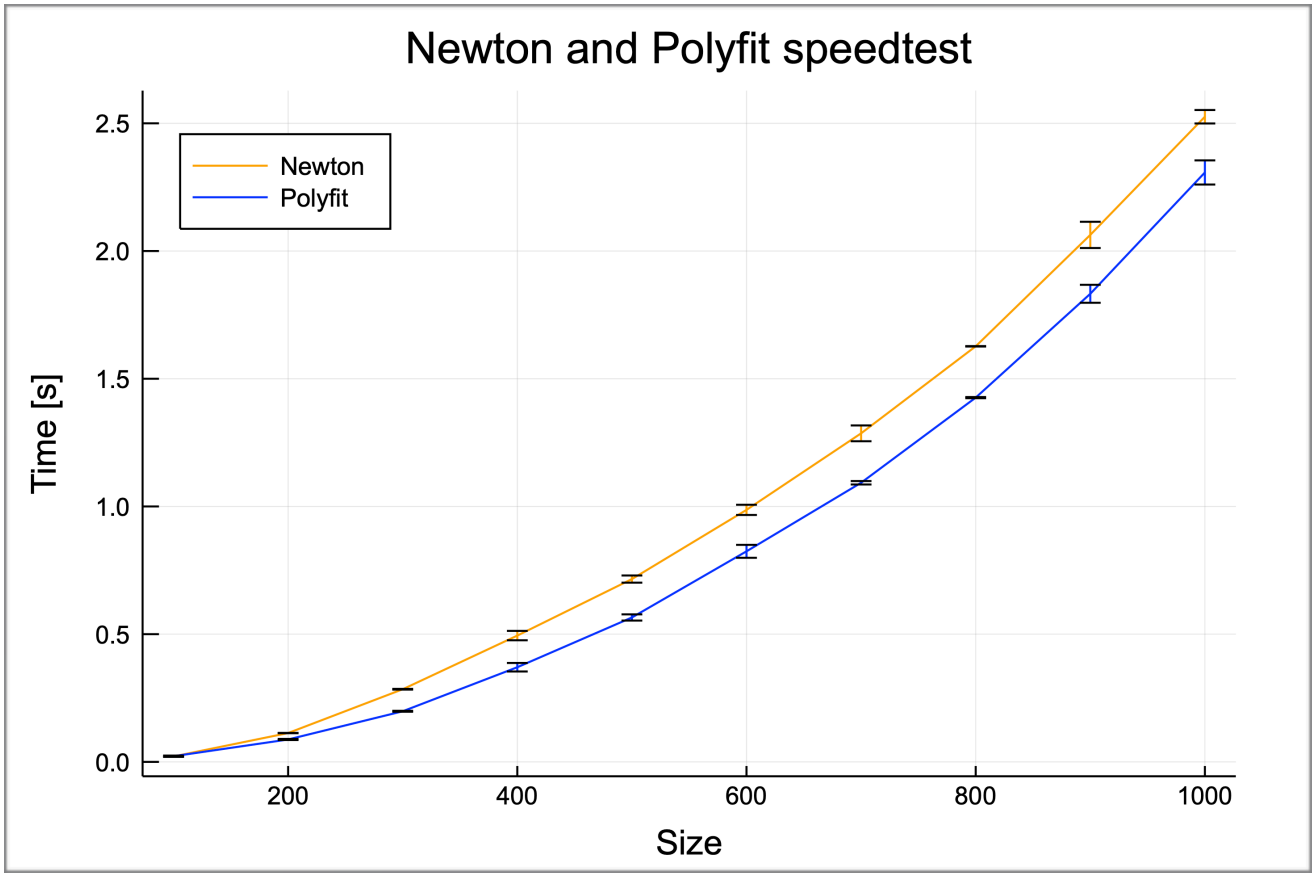
```
10  function gen_table(S, V)
11      max = 9*V + S
12      df = DataFrame(Size = [], Time_lagrange = [], Time_newton = [], Time_poly = [])
13      while S <= max
14          xs = 1:S
15          x = 1:S/100:S
16          fxs = [rand() for x in xs]
17          for i = 1:10
18              time1 = @elapsed Lagrange(x, xs, fxs)
19              time2 = @elapsed Newton(x, xs, fxs)
20              time3 = @elapsed Poly(x, xs, fxs)
21              push!(df, (S, time1, time2, time3))
22          end
23          S += V
24      end
25      return df
26  end
```

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

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



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).



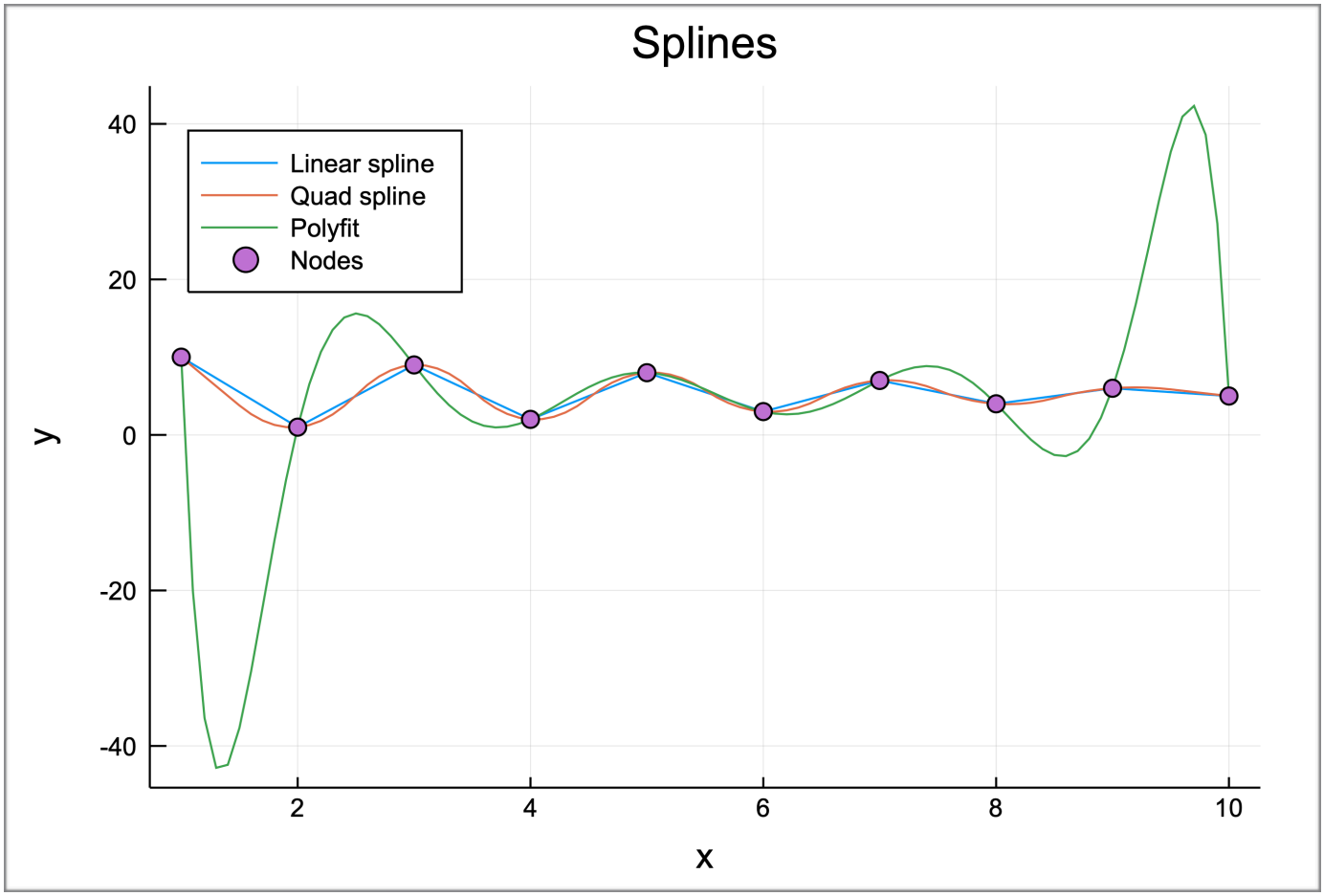
Z wykresu widać, że funkcję są tak samo szybkie, z niewielką przewagą funkcji polyfit.

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.*

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

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.



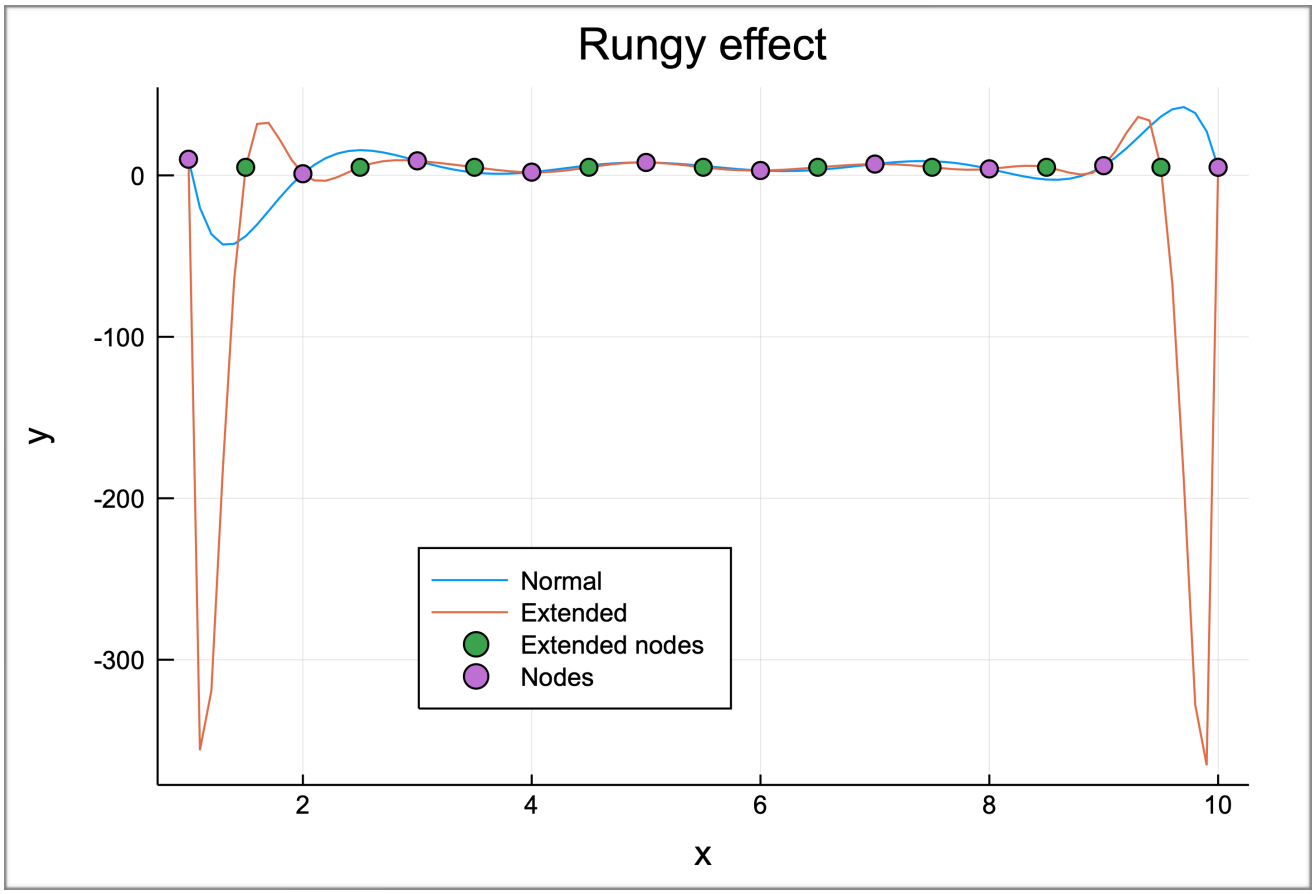
Efekt Rungego

Korzystając z języka Julia stworzyłem dwa zestawy węzłów:

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

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*



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

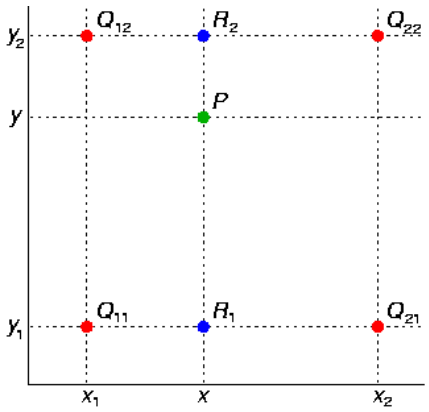
Metoda interpolacji dla funkcji dwóch zmiennych. Zgodnie z intuicją jest to złożenie dwóch interpolacji liniowych. Służy do skalowania obrazów graficznych. W celu wyznaczenia interpolacji dwuliniowej, należy wyznaczyć interpolacje liniowe dla każdego kierunku. Przeprowadzając najpierw interpolacje wzdłuż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

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

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

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:

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

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: