Rozdział 5

Interpolacja wielomianowa i splajnowa

5.1 Wielomiany w octavie

W octavie istnieje cała gamma funkcji związanych z wielomianami:

- **polyval**( ) - funkcja pozwalająca obliczać wartość wielomianu zadanego w bazie potęgowej dla danej wartości x, czy całej macierzy wartości
- **polyfit**( ) - funkcja znajdująca współczynniki wielomianu zadanego stopnia najlepiej dopasowanego do zadanej tabelki punktów.
- polyinteg( ) - zwraca współczynniki wielomianu będącego całką nieoznaczoną z danego wielomianu
- polyderiv( ) - zwraca współczynniki wielomianu będącego pochodną z danego wielomianu
- **roots**( ) - zwraca zera wielomianu
- **conv**(a,b) - zwraca współczynniki wielomianu będącego iloczynem wielomianów o współczynnikach odpowiednio z wektorów a, b

Będziemy korzystali przede wszystkim z dwóch pierwszych funkcji.

We wszystkich tych funkcjach przyjmowane jest, że rozpatrujemy współczynniki wielomianu w bazie potęgowej, ale w następującej kolejności:

$$(x^n, x^{n-1}, \ldots, 1).$$

Współczynniki indeksowane są od jedynki, tzn. wielomian stopnia nie większego od n:

$$w(x) = \sum_{k=1}^{n+1} a_k x^{n+1-k}$$

jest reprezentowany przez wektor współczynników: $(a_1, \ldots, a_{n+1})$.

Jeśli chcemy policzyć wartość w w danym punkcie x, czy ewentualnie dla tablicy punktów umieszczonych w macierzy X, wywołujemy **polyval**(a,x) lub **polyval**(a,X).

Tak więc, aby narysować wykres funkcji $x^3 - 2x + 1$ na $[-3, 4]$ można wykonać następującą sekwencję komend :

```
a=[1,0,-2,1];   #definiujemy wsp.
x=linspace(-3,4);#siatka
wx=polyval(a,x); #wartosc w na siatce
plot(x,wx); #wykres
```

5.2 Interpolacja wielomianowa

Interpolacja wielomianowa polega na tym, że szukamy funkcji p z pewnej przestrzeni wielomianów (zazwyczaj wielomianów stopnia nie większego od n), które w tych punktach spełniają odpowiednie warunki interpolacyjne, tzn. p oraz jej pochodne mają w tych punktach zadane wartości.

5.2.1 Interpolacja Lagrange'a

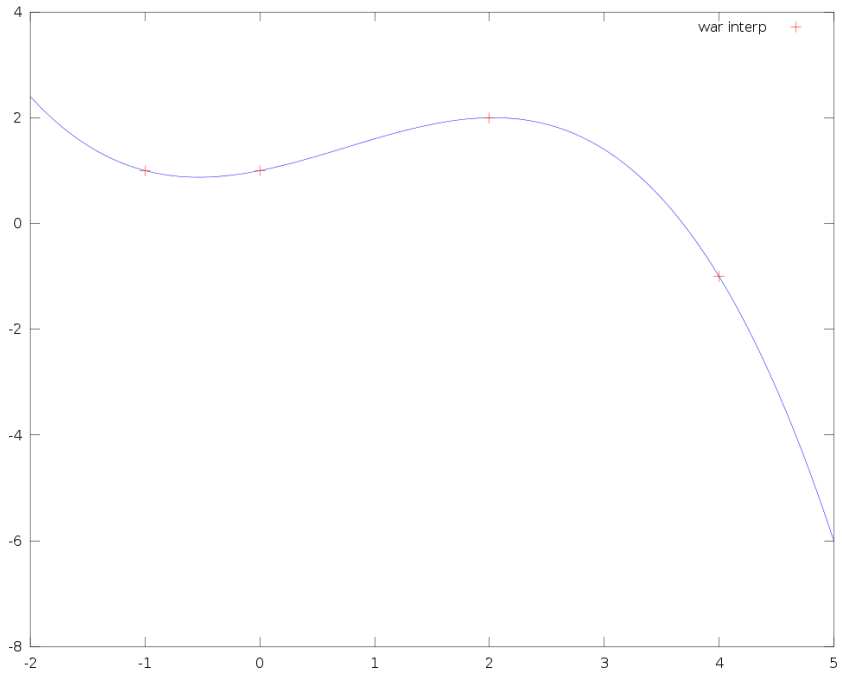
W interpolacji wielomianowej Lagrange'a dla zadanych $n+1$ różnych punktów x_k i wartości $y_k = f(x_k)$ szukamy wielomianu $p(x)$ stopnia co najwyżej n takiego, że spełnione są następujące warunki interpolacyjne:

$$f(x_k) = p(x_k) = y_k \qquad k = 1, \ldots, n+1.$$

Funkcja **polyfit**(x,y,n) znajduje współczynniki wielomianu p w bazie potęgowej dla wektorów x, y długości $n+1$. Ważne aby wartości w x były różne.

Przetestujmy tę funkcję dla następujących danych: $x = (-1, 0, 2, 4)$, $y = (1, 1, 2, -1)$:

```
x=[-1,0,2,4];
y=[1,1,2,-1];
a=polyfit(x,y,3);
s=linspace(-2,5);
plot(s,polyval(a,s),x,y,"r+;war_interp;")
```



Rysunek 5.1: Wykres wielomianu interpolacyjnego. Czerwone plusy - warunki interpolacyjne

Widać na rysunku 5.1, że wielomian spełnia warunki interpolacyjne. Możemy to sprawdzić też obliczeniowo:

```
max(abs(y-polyval(a,x)))
```

Otrzymaliśmy wynik: błąd w węzłach jest równy prawie zero.

Przetestujmy **polyfit**(x,y,n) dla dużych n dla węzłów równoodległych na $[0,1]$ i losowe wartości y z $[-1,1]$, następnie policzmy błąd w węzłach w zależności od N:

```
N=100;
er=zeros(N+1,1);
for k= 0:N,
   x=linspace(0,1,k+1);
   y=2*(rand(size(x))-0.5);
   a=polyfit(x,y,k);
   er(k+1)=max(abs(y-polyval(a,x)));
endfor
```

```
max(er)
```

Widzimy, że możemy otrzymać niepoprawny wynik dla dużych N. Wynika to ze złych własności algorytmu stosowanego przez octave'a ze względu na zaburzenia powodowane przez niedokładne obliczenia w arytmetyce zmiennopozycyjnej.

Zbadajmy błąd pomiędzy funkcją, a jej wielomianem interpolacyjnym. Na początek rozpatrzmy analityczną funkcję $f(x) = \sin(x)$ i węzły równoodległe, oraz normę supremum na odcinku $[-4, 6]$. Normę supremum:

$$\|g\|_{\infty,[a,b]} := \sum_{t\in[a,b]} |g(t)|$$

przybliżymy poprzez dyskretną normę na siatce N równomiernie rozłożonych punktów na tym odcinku $[a, b]$, czyli

$$\|f\|_{h,\infty,[a,b]} = \max_k |f(a + k * h)|$$

z $h = (b - a)/N$:

```
x=[-1,0,2,4];
y=[1,1,2,-1];
a=polyfit(x,y,3);
s=linspace(-2,5);
plot(s,polyval(a,s),x,y,"r+;war_interp;")
```

5.2.2 Interpolacja Hermite'a

Rozpatrzmy teraz zadanie interpolacji Hermite'a. W interpolacji wielomianowej Hermite'a dla zadanych $n + 1$ różnych punktów x_k i naturalnych krotności węzłów p_k szukamy wielomianu $w(x)$ stopnia co najwyżej N dla $N = \sum_{k=0}^{n} p_k - 1$ takiego, że spełnione są następujące warunki interpolacyjne:

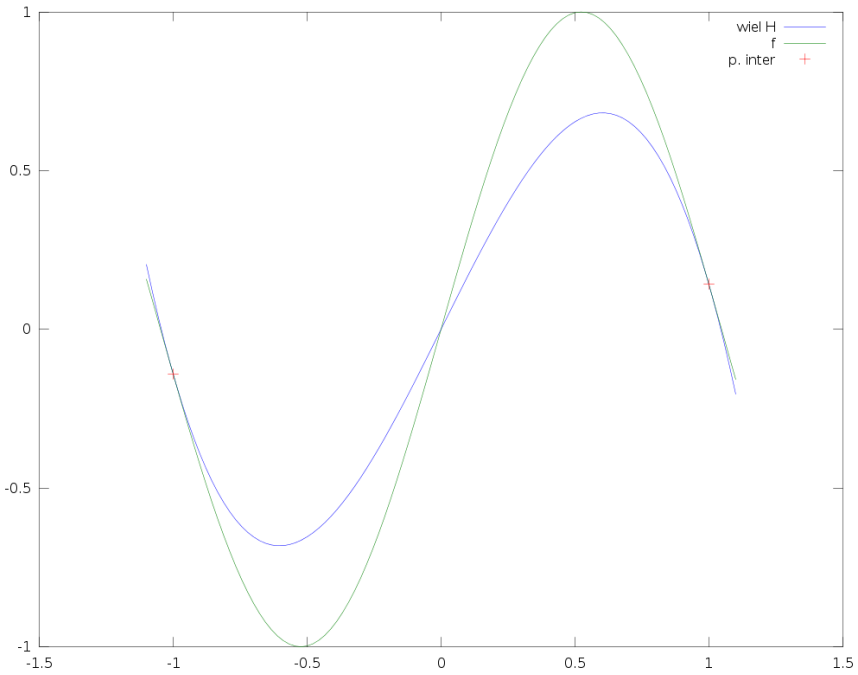
$$w^{(j)}(x_k) = y_{k,j} \qquad k = 1, \ldots, n + 1, \qquad j = 0, \ldots, p_k - 1.$$

Tutaj $y_{k,j}$ to $N + 1$ zadanych wartości.

W octavie nie ma funkcji realizującej interpolacje Hermite'a.

Ale czy w szczególnych przypadkach nie możemy łatwo rozwiązać tego problemu korzystając z odpowiednich funkcji octave'a?

Rozpatrzmy przypadek węzłów tej samej krotności; np. n różnych dwukrotnych węzłów.



Rysunek 5.2: Interpolacja Hermite'a $\sin(3x)$ - dwa dwukrotne węzły $-1, 1$

Chcemy znaleźć współczynniki wielomianu $\sum_{k=1}^{N} a_k x^k$ dla $N = 2(n+1)$ takie, że

$$w(x_k) = f(x_k), \qquad w'(x_k) = f'(x_k).$$

Czyli rozwiązanie spełnia następujący układ równań liniowych:

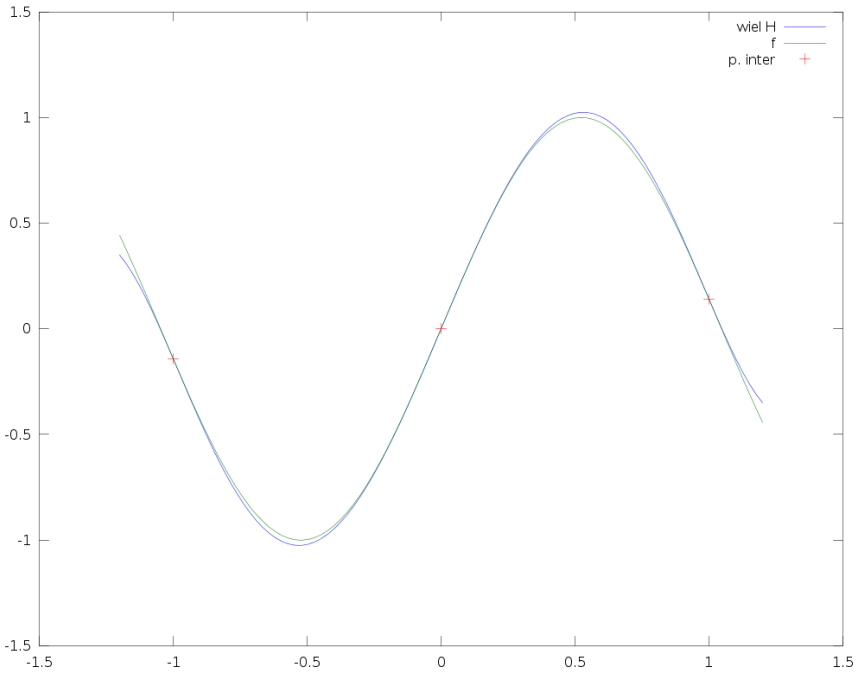
$$\sum_{k=0}^{N} a_k x_j^k = f(x_k) \qquad j = 0, \ldots, n$$

$$\sum_{k=1}^{N} k a_k x_j^{k-1} = f'(x_k) \qquad j = 0, \ldots, n.$$

Utwórzmy macierz tego układu z pomocą funkcji **vander**(x), która tworzy macierz Vandermonde'a dla węzłów podanych w wektorze x:

```
function [a,C,f]=interpolyH(x,y,dy)
#tworzy wspolczynniki wielomianu Hermite'a dla
#wezlow dwukrotnych z x
#y= wartosci w wezlach x (wektor pionowy)
```

```
#wartosci pochodnej w wezlach x (wektor pionowy)
#output: wspolczynniki wielomainu w bazie potegowej
# (Zgodne z polyval())
#C- opcja - macierz ukladu
n=length(x);
N=2*n-1;
A=vander(x,N+1);
D=diag(N:-1:0);
B=zeros(size(A));
B(:,1:N)=A(:,2:N+1);
B=B*D;
C=[A;B];
f=[y;dy];
a=C\[y;dy];
endfunction
```



Rysunek 5.3: Interpolacja Hermite'a $\sin(3x)$ - trzy dwukrotne węzły $-1, 0, 1$

Na początku przetestujmy tę funkcję dla węzłów $-1, 1$ dwukrotnych i funkcji $\sin(3 * x)$.

```
x=[-1;1];
f=@(x) sin(3*x);
df=@(x) 3*cos(3*x);
y=f(x);
dy=df(x);
a=interpolyH(x,y,dy);
z=linspace(-1.1,1.1);
plot(z,polyval(a,z),";wiel_H;",z,f(z),";f;",x,y,"r+");
```

Z wykresu funkcji i wielomianu widzimy, że wielomian przecina wykres i jest styczny w punktach $-1, 1$, por. rysunek 5.2.

Zobaczmy co się stanie w przypadku trzech węzłów. Testujemy funkcję dla węzłów $-2, 0, 2$ dwukrotnych i $\sin(3 * x)$.

```
x=[-1;0;1];
f=@(x) sin(3*x);
df=@(x) 3*cos(3*x);
y=f(x);
dy=df(x);
a=interpolyH(x,y,dy);
z=linspace(-1.2,1.2);
plot(z,polyval(a,z),";wiel_H;",z,f(z),";f;",x,y,"r+");
```

Z rysunku 5.3 widać, że funkcja działa poprawnie również w tym przypadku.

5.3 Interpolacja zespolona. Algorytm FFT

W tym rozdziale omówimy krótko interpolację zespoloną, ale tylko w przypadku określonych węzłów zespolonych równomiernie rozmieszczonych na sferze jednostkowej.

Chcemy znaleźć współczynniki zespolone $a_k \in \mathbb{C}$, $k = 0, \ldots, N$ takie, że

$$\sum_{k=0}^{N} a_k z_j^k = b_k \quad j = 0, \ldots, N$$

dla danych zespolonych b_k i $z_j = \exp\left(\frac{2\pi * i * j}{N+1}\right)$ dla $j = 0, \ldots, N$, czyli pierwiastków z jedynki stopnia $N + 1$.

Równoważnie możemy to zadanie sformułować jako zadanie znalezienia a_k takich, że

$$\sum_{k=0}^{N} a_k \exp\left(\frac{2\pi * i * j * k}{N+1}\right) = b_k \quad j = 0, \ldots, N.$$

Okazuje się, że rozwiązanie możemy wyrazić poprzez operator dyskretnej transformaty Fouriera, czy równoważnie jako mnożenie przez macierz $\mathcal{F}_{N+1} = \frac{1}{N+1}(\omega_{N+1}^{k*l})_{k,l=0}^{N}$:

$$\vec{a} = \mathcal{F}_{N+1}\vec{b}$$

gdzie $\vec{a} = (a_k)_{k=0}^{N}, \vec{b} = (b_k)_{k=0}^{N}$, a $\omega_{N+1} = \exp\left(\frac{-2\pi*i}{N+1}\right)$.

Mnożenie przez macierz $\mathcal{F}_{N+1}$ można szybko wykonać z wykorzystaniem algorytmu szybkiej transformacji Fouriera, czyli FFT (Fast Fourier Transform). Odpowiednia wersja algorytmu FFT służy też szybkiemu mnożeniu przez macierz odwrotną do $\mathcal{F}_{N+1}$: $\mathcal{F}_{N+1}^{-1} = (N+1)\overline{\mathcal{F}}_{N+1} = (\overline{\omega}_{N+1}^{k*l})_{k,l=0}^{N}$.

Obliczając wartość wielomianu $\sum_k a_k z^k$ w punktach z_j, czy równoważnie wartości wielomianu trygonometrycznego $\sum_{k=0}^{N} a_k \exp(i*k*x)$ w punktach $x_j = \frac{2\pi*j}{N+1}$, to przyjmując $b_j = \sum_k a_k z_j^k$ otrzymujemy:

$$\vec{b} = \mathcal{F}_{N+1}^{-1}\vec{a}.$$

Warto dodać, że często macierz DFT definiuje się bez czynnika $\frac{1}{N+1}$, oczywiście wtedy macierz odwrotna też musi być odpowiednio przeskalowana.

Funkcja

fft ()

służy w octave'ie mnożeniu przez macierz $\mathcal{F}_{N+1}$. Jej najprostsze wywołanie to

a=**fft** (b)

Sprawdźmy, czy funkcja **fft** () oblicza wartość DFT zgodnie z naszą definicją. Policzmy $\mathcal{F}_4(4,0,0,0)^T$, powinniśmy otrzymać wektor samych jedynek:

```
a=fft ( [ 4 ; 0 ; 0 ; 0 ] )
```

a otrzymaliśmy:

```
a =

   4 + 0 i
   4 + 0 i
   4 + 0 i
   4 - 0 i
```

To oznacza, że funkcja **fft** () oblicza wartość mnożenia przez $(N+1)*\mathcal{F}_{N+1}$.

Z kolei funkcja **ifft** () powinna obliczać mnożenie przez macierz odwrotną do $(N+1)*\mathcal{F}_{N+1}$.

Sprawdźmy, jak to działa:

```
x=[1, -3, 4,5];
a=fft(x);
xx=ifft(a);
x-xx
norm(x-xx,2)
```

Powtórzmy to dla większego N:

```
x=rand(100);
a=fft(x);
xx=ifft(a);
norm(x-xx,2)
```

Stwórzmy macierz $(N+1)*\mathcal{F}_{N+1}$ i porównajmy szybkość mnożenia przez tę macierz wykonaną za pomocą standardowego operatora octave'a, czyli operatora $*$, z szybkością działania funkcji **fft**().

Stwórzmy najpierw funkcję tworzącą macierz $(N+1)*\mathcal{F}_{N+1}$:

```
function F=DFTmac(N=3)
 om=exp((-2*pi*i*(0:N))/(N+1));
 F=zeros(N+1,N+1);
 for k=0:N,
  F(k+1,:)=om.^k;
 endfor
endfunction
```

Przetestujmy ją na początek dla $N = 3$:

```
F=DFTmac(3)
F*F'
a=rand(4);
norm(fft(a)-F*a,2)
```

W tym przypadku wyniki się pokrywają.

Zgodnie z teorią, por. np. [8], koszt obliczenia DFT przy zastosowaniu algorytmu FFT to $O(n\log_2(n))$, a standardowe mnożenia przez macierz $\mathcal{F}_n$ kosztuje $O(n^2)$.

A teraz testujemy szybkość:

```
n=4*512
F=DFTmac(n-1);
x=rand(n);
tic; y=F*x; t1=toc
tic; yy=fft(x); t2=toc
t1/t2
```

Na moim komputerze FFT było ponad 63 razy szybsze dla $n = 2048$.

Policzmy jeszcze wykres realnego kosztu względem n:

```
n=t1=t2=zeros(100,1);
n(1)=100;
for k=1:100,
 F=DFTmac(n(k)-1);
 x=rand(n(k));
 tic;y=F*x;t1(k)=toc;
 tic;yy=fft(x); t2(k)=toc;
 n(k+1)=n(k)+10;
endfor
n=n(1:100);
plot(n,t1,";mnozenie przez macierz;",n,t2,";fft;")
```

Na wynik chwilę musimy poczekać, ale potwierdza on teorię, że FFT jest wyraźnie szybszym algorytmem.

5.4 Interpolacja splajnowa

W przypadku interpolacji splajnowej rozpatrujemy dane węzły:

$$a = x_0, \ldots, x_N = b$$

na odcinku $[a, b]$, oraz funkcje splajnowe (splajny), czyli funkcje, które są odpowiedniej klasy gładkości (czyli są w $C^k([a, b])$, oraz obcięte do dowolnego pododcinka $[x_k, x_{k+1}]$ dla $k = 0, \ldots, N-1$, są wielomianami co najwyżej ustalonego stopnia.

W przypadku splajnów kubicznych rozważamy funkcje, które są klasy C^2 na $[a, b]$, oraz są wielomianami kubicznymi na każdym pododcinku.

Zadanie interpolacji splajnowej kubicznej polega na znalezieniu splajnu kubicznego s takiego, że

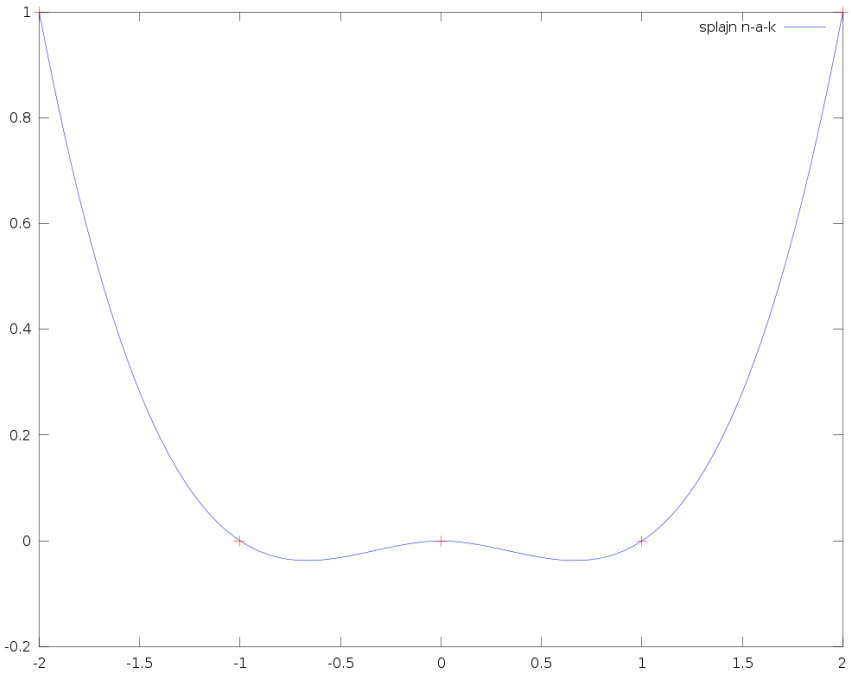
$$s(x_k) = y_k \quad k = 0, \ldots, N$$

dla zadanych y_k, oraz spełniającego odpowiednie warunki brzegowe. Np. w przypadku splajnu hermitowskiego s musi spełnić warunki brzegowe hermitowskie:

$$s'(a) = dy_a \qquad s'(b) = dy_b$$

dla zadanych dodatkowych dwóch wartości dy_a, dy_b. W przypadku splajnu naturalnego warunki brzegowe to

$$s''(a) = s''(b) = 0.$$



Rysunek 5.4: Wykres prostego splajnu typu not-a-knot. Czerwone plusy oznaczają punkty interpolacji

Rozpatruje się również splajny typu *not-a-knot*. W tym przypadku, zamiast warunków brzegowych, dodaje się sztucznie dwa warunki ciągłości trzeciej pochodnej w węzłach x_1 i x_{N-1}.

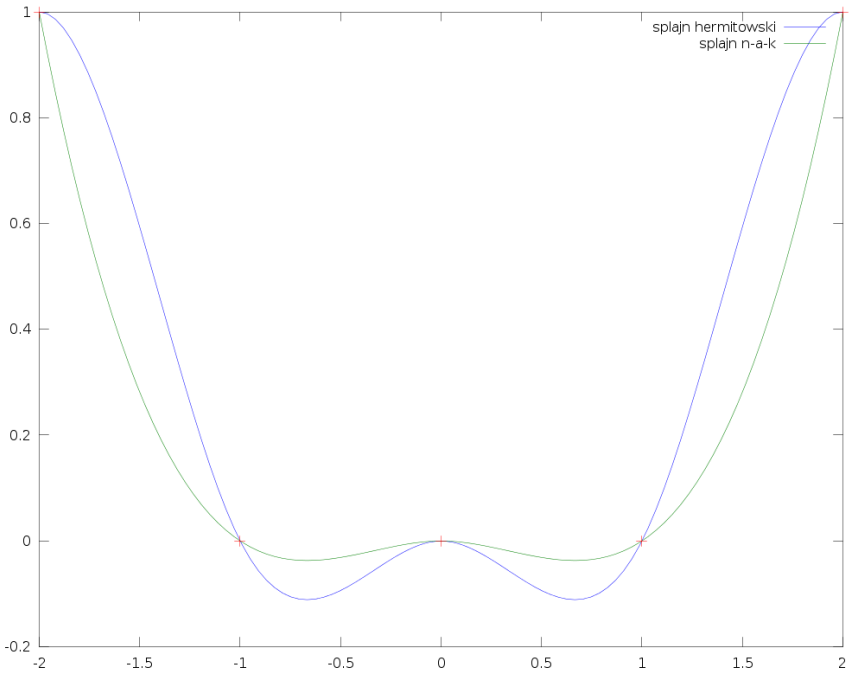
Z kolei splajny okresowe spełniają następujące warunki brzegowe okresowe:

$$s^{(j)}(a) = s^{(j)}(b) \quad j = 0, 1, 2.$$

W każdym z tych czterech przypadków splajn interpolacyjny jest wyznaczony jednoznacznie, por. np. [11].

W octave'ie istnieje kilka funkcji związanych z interpolacją splajnami kubicznymi, czy ogólnie - z funkcjami wielomianowymi na pododcinkach:

- **spline**() - funkcja służąca znalezieniu splajnu interpolacyjnego hermitowskiego, czy typu *not-a-knot*
- ppval() - funkcja służąca obliczeniu wartości splajnu zadanego w formacie octave'a



Rysunek 5.5: Wykresy prostych splajnów typu not-a-knot i hermitowskiego (z zerowymi pochodnymi w końcach) z tymi samymi warunkami interpolacyjnymi. Czerwone plusy oznaczają punkty interpolacji

- mkpp() - tworzy funkcję wielomianową na pod-odcinkach (szczegóły **help** mkpp())
- unmkpp() - ze struktury splajnu w octave'ie zwraca współczynniki wielomianów na pod-odcinkach (szczegóły **help** unmkpp())

Podstawową funkcją służącą znalezieniu splajnu interpolacyjnego hermitowskiego, czy typu *not-a-knot* jest funkcja **spline**(). Jej najprostsze wywołanie to

p=**spline**(x,y)

gdzie x to wektor wymiaru N z węzłami interpolacji splajnowej, a wektor y ma tę samą długość co x i zawiera wartości jakie ma przyjąć splajn w tych węzłach. Druga możliwość to - przy takim samym wektorze węzłów x, podanie wektora y o długości $N+2$. Wtedy pierwsza i ostatnia wartość wektora y to wartości pochodnych splajnu w końcowych węzłach x. Pozostałe

wartości y tzn. y_k dla $k = 2, \ldots, N+1$ zawierają wartości splajnu w węzłach z x, czyli są takie same jak w pierwszym przypadku. Funkcja zwraca strukturę typu *pp*, czyli w odpowiednim formacie octave'a splajnu kubicznego typu *not-a-knot* w pierwszym przypadku, a w drugim - splajnu hermitowskiego. Następnie, korzystając z funkcji ppval() można obliczyć wartość tego splajnu w punkcie, czy tablicy punktów.

Policzmy splajn który w węzłach $-2, 1, 0, 1, 2$ przyjmie wartości $1, 0, 0, 0, 1$ i narysujmy jego wykres, por. rysunek 5.4:

```
x=-2:2;
y=[1,0,0,0,1];
pp=spline(x,y);
z=linspace(-2,2);
plot(z,ppval(pp,z),";splajn n-a-k;",x,y,"r+");
```

Korzystając z tej samej funkcji, stwórzmy splajn hermitowski przyjmując, że jego pochodne w końcach wynoszą zero. Następnie narysujmy wykresy obu splajnów na odcinku $[-2, 2]$, por. rysunek 5.5, oraz blisko lewego końca, por. rysunek 5.6:

```
x=-2:2;
y=[1,0,0,0,1];
pp=spline(x,y);
yh=[0,y,0];
ph=spline(x,yh);
z=linspace(-2,2);
plot(z,ppval(ph,z),";splajn hermitowski;",...
z,ppval(pp,z),";splajn n-a-k;",x,y,"r+");
pause(2);
z=linspace(-2,-2+0.1);
plot(z,ppval(ph,z),";splajn hermitowski;",...
z,ppval(pp,z),";splajn n-a-k;",-2,1.3);
```

Widać, że splajny są różne, oraz że splajn hermitowski ma pochodną równą zero w lewym i prawym końcu.

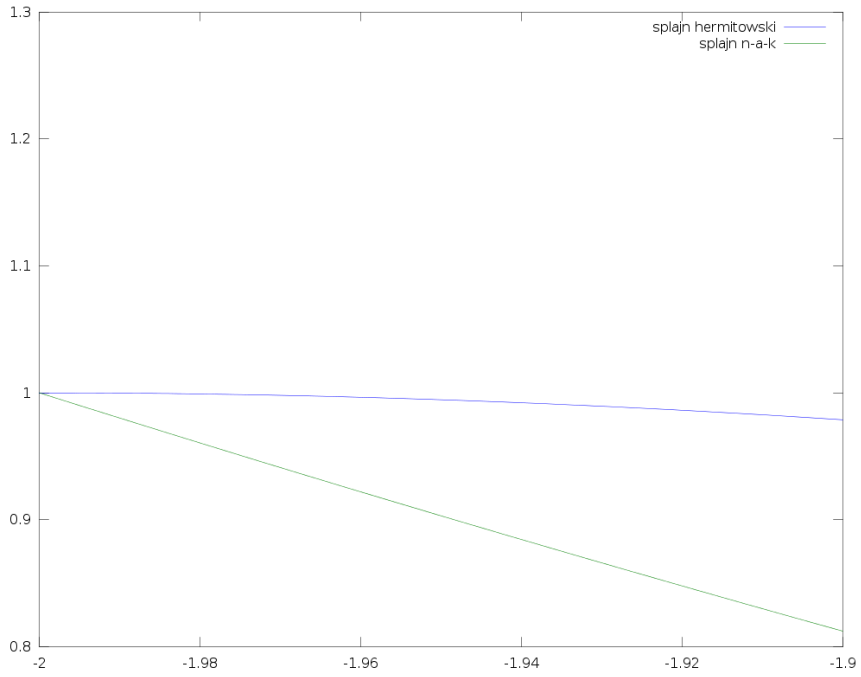
Czy w octave'ie można wyznaczyć w prosty sposób inne splajny, np. naturalny lub okresowy?

Okazuje się, że tak, ale trzeba wykorzystać funkcję z rozszerzenia octave'a, czyli octave-forge'a: csape().

Jej wywołanie to

```
pp=csape(x,y,cond,valz)
```

gdzie x, y to wektory długości N z węzłami i wartościami splajnu w węzłach, a *cond* przyjmuje wartości:



Rysunek 5.6: Wykresy prostych splajnów typu not-a-knot i hermitowskiego (z zerowymi pochodnymi w końcach) blisko lewego końca.

- *'variational'* w przypadku splajnu interpolacyjnego kubicznego naturalnego
- *'complete'* w przypadku splajnu interpolacyjnego kubicznego hermitowskiego, wartości pochodnych w końcach są podane w parametrze valc
- *'not-a-knot'* w przypadku splajnu interpolacyjnego kubicznego typu *not-a-knot*
- *'periodic'* w przypadku splajnu interpolacyjnego kubicznego okresowego
- *'second'* w przypadku splajnu interpolacyjnego kubicznego z ustalonymi wartościami drugiej pochodnej w końcach, zgodnymi z tym, co podano w parametrze valc

Funkcja zwraca strukturę typu *pp*, czyli dane splajnu kubicznego w formacie octave'a.

Porównajmy wyniki tej funkcji z wynikiem funkcji **spline**() dla danych z naszego prostego przykładu i splajnu typu *not-a-knot*. Policzymy dyskretną normę maksimum (na siatce równomiernej 300 punktów na $[-2, 2]$ z wyników obu funkcji:

```
x=-2:2;
y=[1,0,0,0,1];
pp=spline(x,y);
pp1=csape(x,y,'not-a-knot');
z=linspace(-2,2,300);
norm(ppval(pp,z)-ppval(pp1,z),'inf')
```

Otrzymaliśmy zero.

Na jednym wykresie narysujmy wykresy trzech splajnów interpolacyjnych dla $N = 6$ z tymi samymi warunkami interpolacyjnymi, ale z różnymi warunkami brzegowymi: hermitowskim, *not-a-knot* i naturalnym.

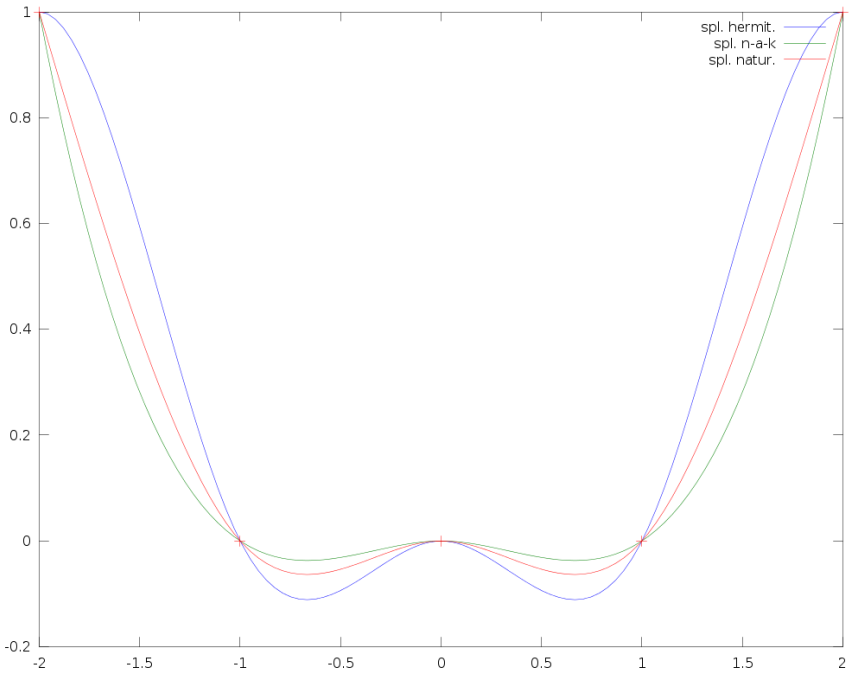
```
N=4;
x=-2:2;
y=[1,0,0,0,1];
pp=spline(x,y);
ppn=csape(x,y,'variational');
yh=[0,y,0];
pph=spline(x,yh);
z=linspace(-2,2);
plot(z,ppval(pph,z),";spl. hermit.;",...
z,ppval(pp,z),";spl. n-a-k;",...
z,ppval(ppn,z),";spl. natur.;",x,y,"r+");
```

Wszystkie trzy splajny są różne w tym przypadku, por. rysunek 5.7.

Oczywiście obie funkcje umożliwiają dowolny wybór węzłów, np. weźmy węzły $[-2, -1, 2, 3, 4]$ z tymi samymi wartościami i stwórzmy splajny obu typów, por. rysunek 5.8:

```
x=[-3,2,3,4];
y=[1,0,0,0,1];
pp=spline(x,y);
yh=[0,y,0];
ph=spline(x,yh);
z=linspace(-3,4);
plot(z,ppval(ph,z),";splajn hermitowski;",...
z,ppval(pp,z),";splajn n-a-k;",x,y,"r+");
```

Popatrzmy na błędy aproksymacji w normie supremum. Ustalmy, że interpolujemy splajnami kubicznymi znaną funkcję gładką $f(x)$ na czterech



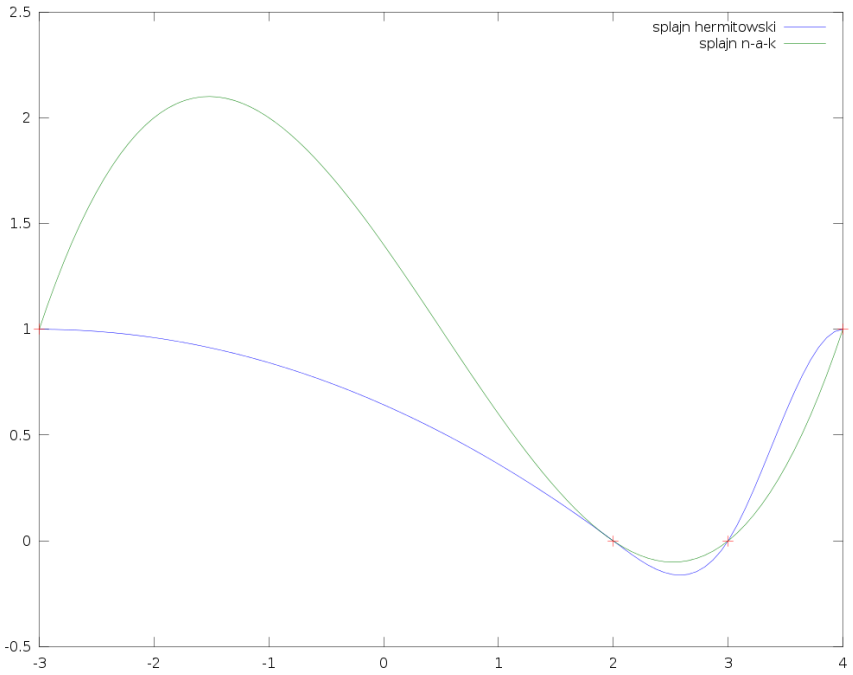
Rysunek 5.7: Wykresy trzech różnych splajnów spełniających te same warunki interpolacyjne. Splajn hermitowski posiada pochodne równe zero w końcach odcinka.

węzłach równoodległych na odcinku $[a, b]$. Jeśli *pp* to struktura typu *pp* opisująca splajn interpolujący f, to dyskretną normę maksimum różnicy między f, a splajnem s, np. na siatce o tysiącu punktach, możemy obliczyć komendą:

```
z=linspace(a,b,1000);
y=f(z);
s=ppval(pp,z);
errmax=norm(s-y,'inf')
```

Założyliśmy, że funkcja f jest zaimplementowana wektorowo, tzn. że wywołanie w octave'ie f(z) dla z wektora zwróci wektor $(f(z(k)))$. W przeciwnym razie należałoby użyć pętli do wyznaczenia wektora y:

```
z=linspace(a,b,1000);
for k=1:length(z),
  y(k)=f(z(k));
endfor
```

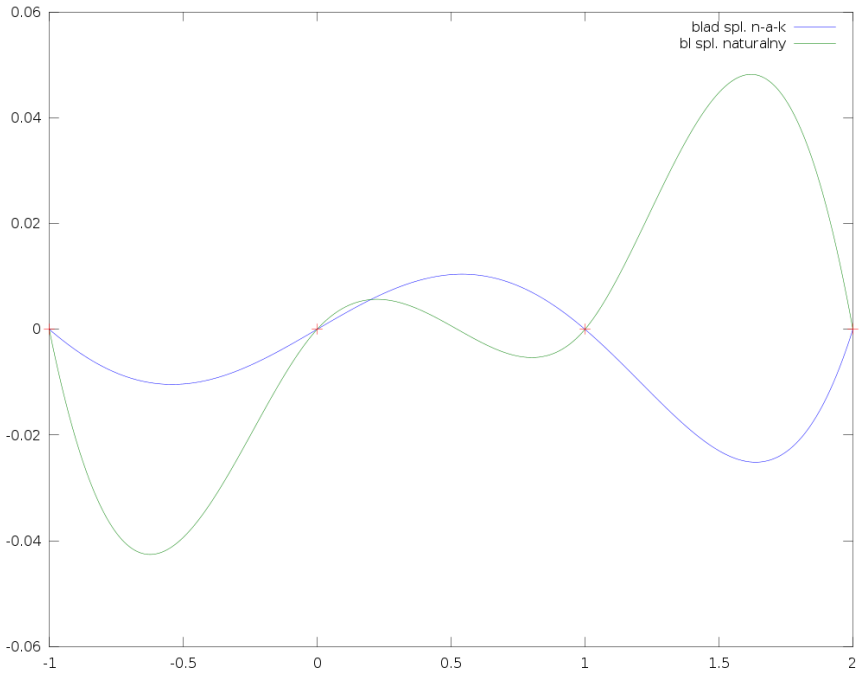


Rysunek 5.8: Wykresy prostych splajnów typu not-a-knot i hermitowskiego z węzłami nierównoodległymi z tymi samymi warunkami interpolacyjnymi. Czerwone plusy oznaczają punkty interpolacji. Splajn hermitowski posiada pochodne równe zero w końcach odcinka.

```
s=ppval(pp,z);
errmax=norm(s-y,'inf')
```

W poniższym kodzie obliczymy przybliżoną normę maksimum na odcinku $[-1, 2]$ pomiędzy $f(x) = \sin(x)$, a jej dwoma splajnami interpolacyjnymi kubicznymi: naturalnym i typu not-a-knot na czterech węzłach równoodległych:

```
a=-1;
b=2;
f=@sin;
N=4;
x=linspace(a,b,4);
y=f(x);
pp=spline(x,y);
ppn=csape(x,y,'variational');
z=linspace(a,b,1000);
```



Rysunek 5.9: Wykresy różnicy pomiędzy funkcją $\sin(x)$, a jej dwoma splajnami interpolacyjnymi - naturalnym i typu not-a-knot. Czerwone plusy oznaczają cztery równoodległe punkty interpolacji na $[-1, 2]$.

```
fz=f(z);
pz=ppval(pp,z);
pn=ppval(ppn,z);
err=norm(fz-pz,'inf')
errn=norm(fz-pn,'inf')
```

Błąd dla splajnu naturalnego był większy: errn $=$ 0.048190, niż dla splajnu typu not-a -knot: err $=$ 0.025120, co widać też na rysunku 5.9.

Jako zadanie pozostawiamy policzenie błędów dla innych funkcji, węzłów i typów splajnów kubicznych interpolacyjnych.