Rozdział 4

Arytmetyka zmiennopozycyjna

Wszystkie obliczenia w octavie są wykonywane w arytmetyce zmiennopozycyjnej (inaczej - arytmetyce fl) podwójnej precyzji (double) - choć w najnowszych wersjach octave'a istnieje możliwość używania zmiennych typu single, czyli zmiennych w pojedynczej precyzji arytmetyki zmiennopozycyjnej.

Dokładność względna arytmetyki podwójnej precyzji wynosi ok. 10^{-16}, a dokładność arytmetyki pojedynczej precyzji wynosi ok. 10^{-7}.

W octavie jest kilka funkcji związanych bezpośrednio z własnościami arytmetyki fl, mianowicie:

- **eps** - zwraca epsilon maszynowy arytmetyki, tzn. najmniejszą liczbę ϵ taką, że $fl(\epsilon + 1) > 1$. Tu $fl()$ oznacza wynik działania w arytmetyce zmiennopozycyjnej. Octave domyślnie działa na liczbach zmiennopozycyjnych w podwójnej precyzji arytmetyki więc oczywiście **eps** zwróci nam epsilon maszynowy dla arytmetyki podwójnej precyzji. Z kolei wywołanie **eps**(a) - dla a liczby zwróci odległość od a do najbliższej większej od a liczby w arytmetyce zmiennopozycyjnej

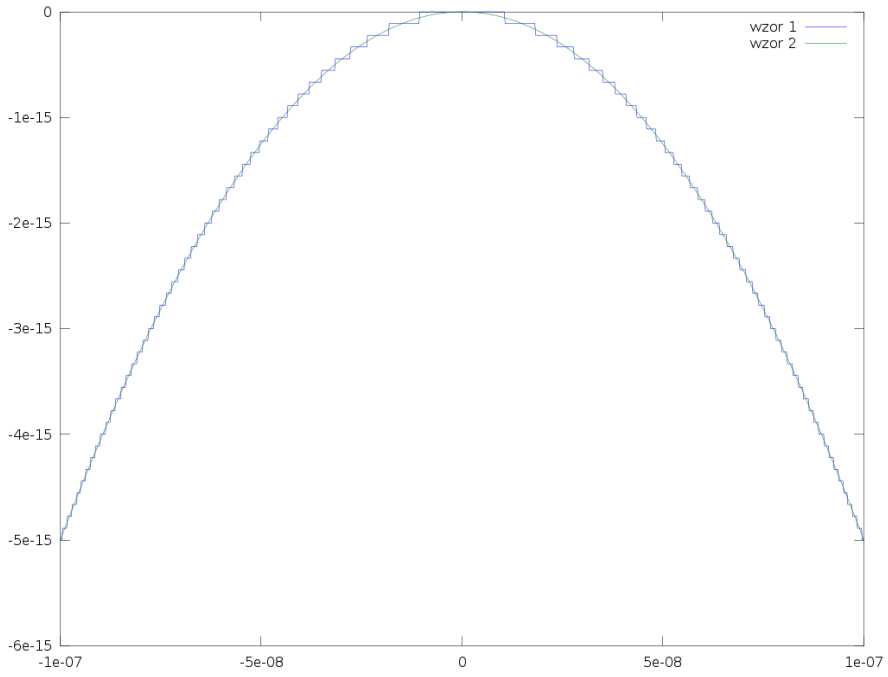
- single (x) - funkcja konwertująca zmienną np. typu double, czyli w podwójnej precyzji arytmetyki, do zmiennej typu single, czyli w pojedynczej precyzji arytmetyki zmiennopozycyjnej

- double(x) - funkcja konwertująca zmienną do typu zmiennej w podwójne precyzji arytmetyki, czyli odwrotna do funkcji single (x). Podwójna precyzja jest w octavie domyślna, więc wydaje się że, głównym zastosowaniem tej funkcji jest konwersja zmiennej typu single z powrotem do typu double

Zastosujmy funkcję **eps** - sprawdźmy, czy rzeczywiście $eps + 1$ w fl jest większe od jeden, a np. $eps/2 + 1$ już nie - tu fragment kodu octave'a:

```
if((eps+1)>1)
 printf("%g+1>1 w fl\n",eps);
else
 printf("%g nie jest epsilonem maszynowym\n",eps);
endif

if((1+eps/2)==1)
 printf(" fl(%g+1)=1\n",eps/2);
endif
```



Rysunek 4.1: Dwa sposoby obliczania $f(x) = 1 - \cos(x)$ w arytmetyce zmiennopozycyjnej.

Przy pomocy **eps** możemy znaleźć najmniejszą liczbę dodatnią w arytmetyce podwójnej precyzji:

```
octave:33> x=eps(0)
x =  4.9407e-324
octave:34>
```

Sprawdźmy:

```
octave:34> x/2>0
ans = 0
octave:35>
```

Sprawdźmy, jak działają funkcje związane ze zmianą precyzji arytmetyki single i double - tu fragment sesji octave'a:

```
b=single(eps)
if((single(eps)+1)==1)
printf("fl(single(eps)+1)==1\n");
endif

if((double(b)+1)>1)
 printf("fl(double(single(eps))+1)>1.\n");
endif
#a teraz epsilon maszynowy precyzja pojedyncza
eps(single(1))
```

Otrzymaliśmy, że epsilon maszynowy w podwójnej precyzji wynosi: $2.2204e-16$, a w pojedynczej precyzji arytmetyki: $1.1921e-07$.

Teraz znajdziemy najmniejszą liczbę dodatnią w arytmetyce pojedynczej precyzji:

```
octave:52> eps(single(0))
ans =  1.4013e-45
octave:53>
```

4.1 Redukcja cyfr przy odejmowaniu

Inną własnością arytmetyki fl, która może powodować problemy, jest tzw. redukcja cyfr przy odejmowaniu liczb tego samego znaku.

4.1.1 Przykład

Przetestujmy to na dwóch równoważnych wzorach na funkcję

$$f(x) = \cos(x) - 1 = 1 - 2 * \sin^2(0.5 * x) - 1 = -2 * \sin^2(0.5 * x).$$

Przetestujmy oba wzory dla x bliskich zeru:

```
f1=@(x) cos(x)-1;
f2=@(x) -2*sin(0.5*x).*sin(0.5*x);
a=1e-7;
```

```
x=linspace(-a,a,1000);
plot(x,f1(x),";wzor 1;",x,f2(x),";wzor 2;");
```

Wykres na rysunku 4.1 pokazuje, że wzór drugi jest lepszy ze względu na arytmetykę fl.

Możemy oczywiście policzyć też błąd pomiędzy wynikami:

```
er=f1(1e-7)-f2(1e-7)
```

Różnica $3.9964e-18$ jest pozornie mała, bo rzędu 10^{-18}. Jednak $f2(10^{-7})$ jest rzędu 10^{-14}, więc dokładność względna jest rzędu 10^{-4}. Jest to bardzo mała dokładność, jak na arytmetykę podwójnej precyzji, w której błąd względny obliczeń jest na poziomie 10^{-16}.

Możemy policzyć względny błąd względem $f2(10^{-7})$:

```
abs((f1(1e-7)-f2(1e-7))/f2(1e-7))
```

i otrzymamy $7.9928e-04$.

4.1.2 Obliczanie funkcji z rozwinięcia w szereg

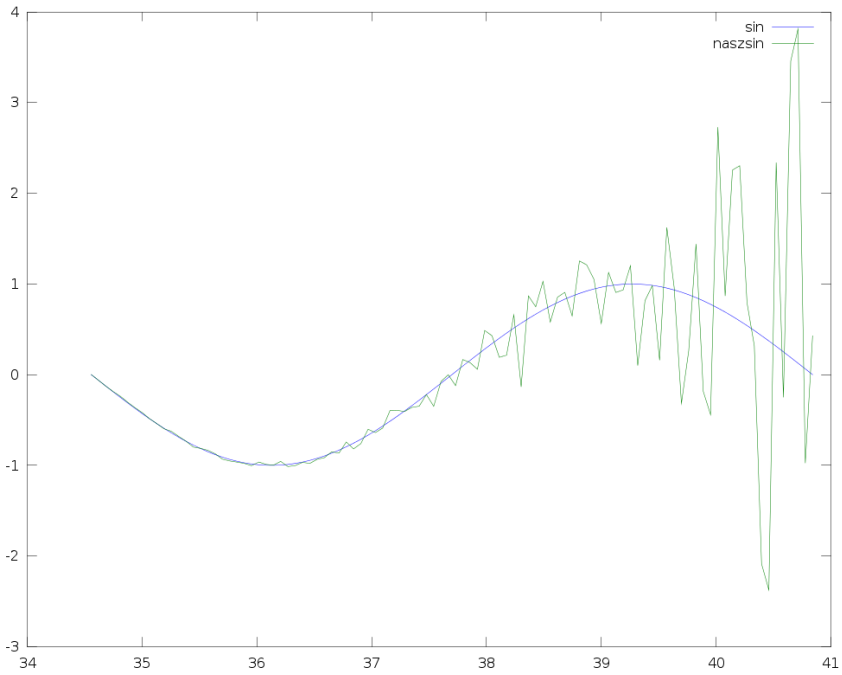
Inny przykład na problemy redukcją cyfr, to obliczanie wartości funkcji korzystające z rozwinięcia w szereg o wyrazach o różnych znakach.

Rozpatrzmy funkcję $\sin(x)$, która ma następujące rozwinięcie w globalnie zbieżny szereg:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots.$$

Zaimplementujmy w octavie funkcję obliczającą przybliżenie funkcji matematycznej $\sin(x)$, korzystające z odpowiedniego obcięcia tego rozwinięcia dla dowolnego x rzeczywistego. Implementacja tej funkcji będzie wektorowa, tzn. wywołanie jej z macierzą X jako argumentem zwróci macierz przybliżeń wartości sin na odpowiednich elementach macierzy X.

```
function y=naszsin(x,N=300)
#obliczamy wartosc sin(x)
#korzystajac z rozwiniecia w szereg :
#x-x^3/3!+x^5/5!-....(-1)^(k+1)*x^{2k+1}/(2k+1)!+...
#N - ilosc wyrazow w szeregu - domyslnie 300
  kx=-x.*x;
  y=x;
  for k=1:N,
    x=x.*kx./((2*k)*(2*k+1));
    y+=x;
  endfor
endfunction
```



Rysunek 4.2: Wykres funkcji sinus obliczanej z rozwinięcia w szereg na $[11\pi, 13\pi]$

Przetestujmy naszą funkcję na $[-\pi, \pi]$.

```
w=linspace(-pi,pi);
plot(w,sin(w),";sin;",w,naszsin(w),";naszsin;")
```

Na wykresie nie widać różnicy.

Przetestujmy naszą funkcję dla większych argumentów: $[11 * \pi, 13 * \pi]$:

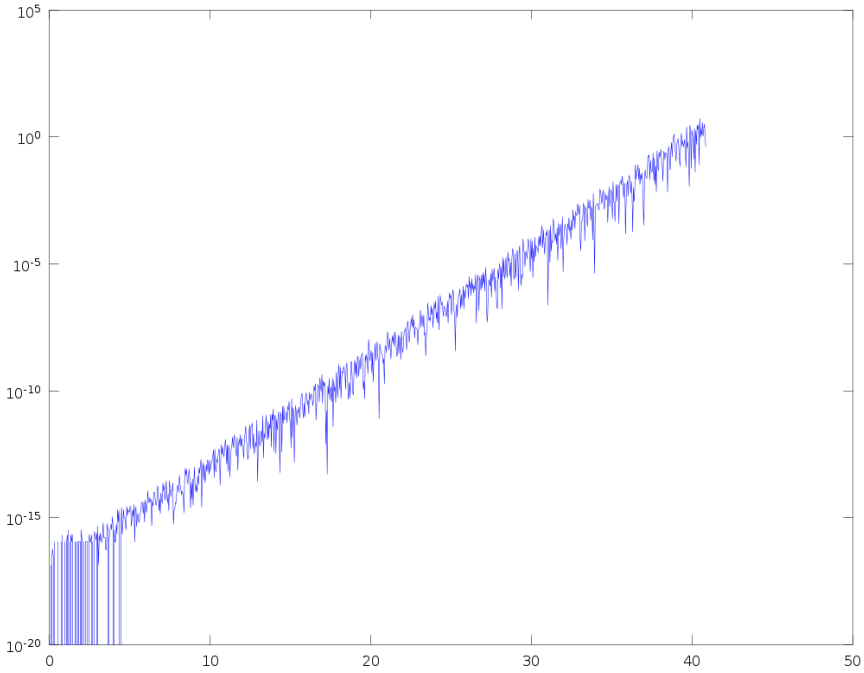
```
w=linspace(11*pi,13*pi);
plot(w,sin(w),";sin;",w,naszsin(w),";naszsin;")
```

Widać z rysunku 4.2, że rozwijanie w szereg nie zawsze ma sens z powodu własności arytmetyki zmiennopozycyjnej.

Można postawić pytanie, czy w rozwinięciu nie było za mało wyrazów szeregu?

```
w=linspace(11*pi,13*pi);
plot(w,sin(w),";sin;",w,naszsin(w,100000),";naszsin;")
```

Proszę sprawdzić, że zwiększenie ilości wyrazów szeregu nic nie zmieniło.



Rysunek 4.3: Wykres błędu w skali półlogarytmicznej obliczania funkcji sinus z rozwinięcia w szereg

Popatrzmy, jak rośnie błąd na skali logarytmicznej, por. rysunek 4.3:

```
w=linspace(eps,13*pi,1000);
semilogy(w,abs(sin(w)-naszsin(w)))
```

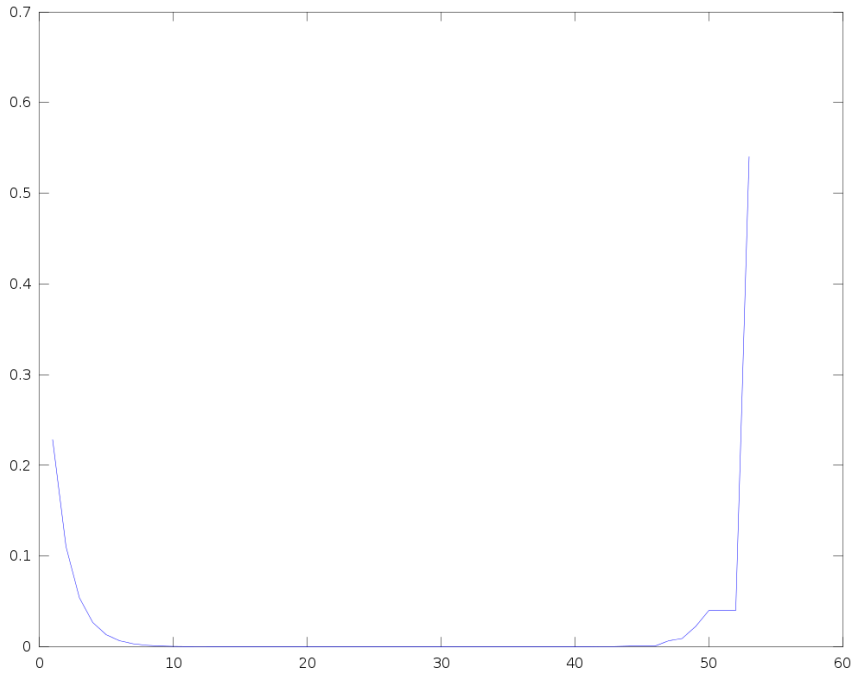
Błąd w skali logarytmicznej rośnie liniowo. To oznacza, że błąd rośnie wykładniczo.

4.1.3 Obliczanie przybliżonej wartości pochodnej

Kolejnym przykładem sytuacji, w której mogą pojawić się problemy, to obliczanie pochodnej za pomocą ilorazów różnicowych, czyli wprost z definicji. Przetestujmy przybliżone obliczanie pochodnej z definicji, czyli ze wzoru:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

Wydawałoby się, że im h jest mniejsze, tym lepsze przybliżenie otrzymamy. Ale np. dla $f(x) = \sin(x)$ z $x = 1$ i $h < eps$, np. dla $h = eps/2$, otrzymamy



Rysunek 4.4: Wykres błędu przy obliczaniu pochodnej ilorazem różnicowym $dsin()$ dla $x = 1$ w zależności od h.

w arytmetyce fl:

$$fl(x + h) = fl(1 + h) = 1.$$

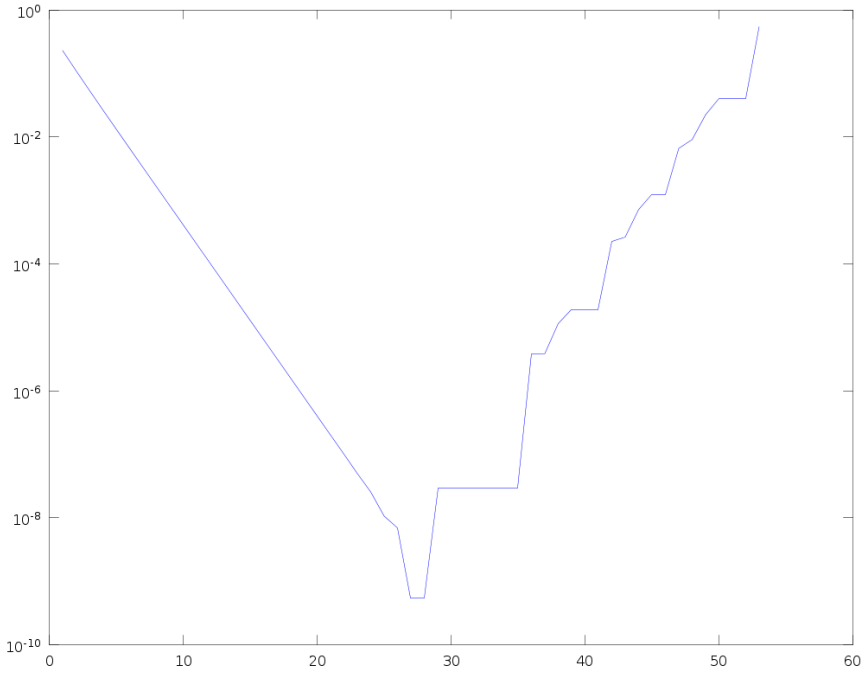
Sprawdźmy:

```
> h=eps/2
h =   1.1102e-16
> sin(1+h)-sin(1)
ans = 0
#czy na pewno
>sin(1+h)==sin(1)
ans=1
```

Czyli rzeczywiście h nie może być dowolnie małe.

Spróbujmy znaleźć dla tego przykładu optymalne h postaci $h = 2^{-n}$ eksperymentalnie.

```
f=@sin;
h=1;x=1;
```



Rysunek 4.5: Wykres błędu przy obliczaniu pochodnej ilorazem różnicowym $dsin()$ dla $x = 1$ w zależności od h w skali półlogarytmicznej

```
fx=f(x); dfx=cos(x);
er=zeros(53,1);
eropt=er(1)=abs((f(x+h)-fx)/h-dfx);
hopt=h;n=nopt=0;
while(h>=eps)
  h/=2;
  n++;
  er(n)=abs((f(x+h)-fx)/h - dfx);
 if(er(n)<eropt)
    eropt=er(n);
    nopt=n;
    hopt=h;
 endif
endwhile
printf("optymalne h=%g=2^(-%d)\n",hopt,nopt);
plot(er);
```

Na wykresie nie widać dokładnie, por. rysunek 4.4, gdzie jest minimum. Na wykresie w skali pół-logarytmicznej minimum jest widoczne, por. rysunek 4.5:

semilogy (e r)

Widzimy, że optymalne h w tym przypadku jest rzędu 2^{-27}, czyli ok 10^{-9}.

Można powtórzyć te same obliczenia dla przybliżania pochodnej za pomocą tzw. ilorazu centralnego:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

W kolejnych rozdziałach będziemy omawiali wpływ arytmetyki zmiennopozycyjnej na wyniki obliczeń dla konkretnych zadań.