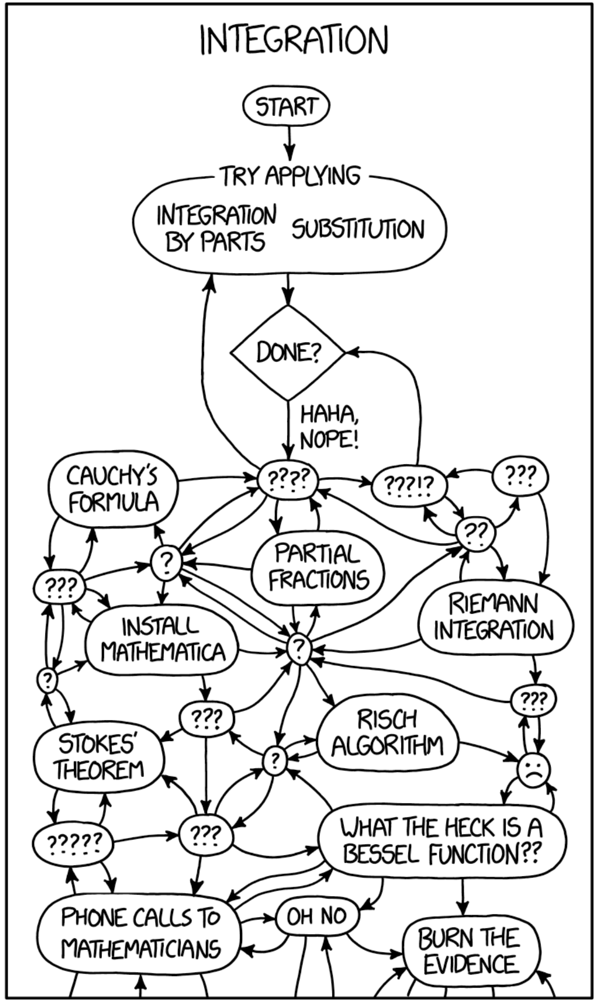
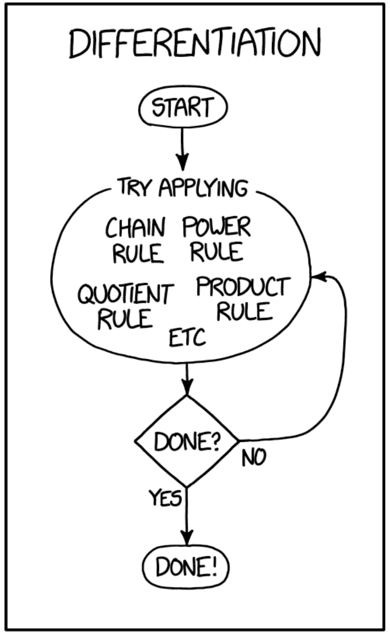
Algorytmy w Inżynierii Danych (2024)

Różniczkowanie: symboliczne, numeryczne i automatyczne

Bartosz Chaber


INTEGRATION
START
TRY APPLYING
INTEGRATION BY PARTS
SUBSTITUTION
DONE?
HAHA, NOPE!
CAUCHY'S FORMULA
????
???!?
???
??
PARTIAL FRACTIONS
?
???
RIEMANN INTEGRATION
INSTALL MATHEMATICA
?
?
???
RISCH ALGORITHM
STOKES' THEOREM
???
?
?????
???
WHAT THE HECK IS A BESSEL FUNCTION??
PHONE CALLS TO MATHEMATICIANS
OH NO
BURN THE EVIDENCE


DIFFERENTIATION
START
TRY APPLYING
CHAIN RULE
POWER RULE
QUOTIENT RULE
PRODUCT RULE
ETC
DONE?
NO
YES
DONE!

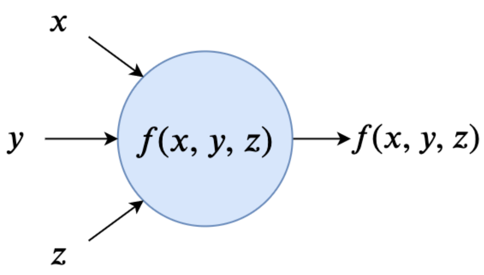
Funkcje dla informatyków i matematyków

Możemy je rozpatrywać w sensie informatycznym jak i matematycznym. Niektóre funkcje w języku programowania mogą być równoważne funkcjom matematycznym.

```python
def distance(x, y):
    return x*x + y*y
```

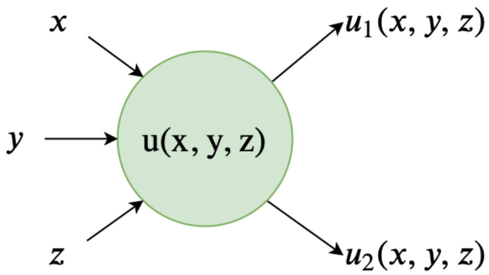
```julia
function distance(x::Real, y::Real)
    x^2 + y^2
end
```

$$f(x, y) = x^2 + y^2, f : \mathbb{R}^2 \to \mathbb{R}$$



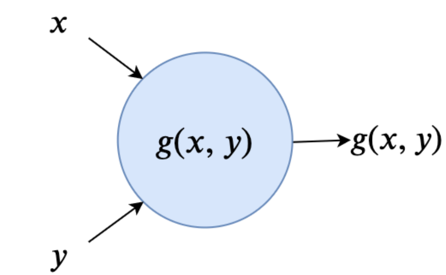
Funkcja skalarna trzech zmiennych

$f(x, y, z) = x^2 + y^2 + z^2$



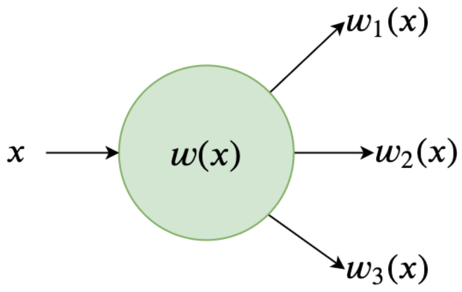
Funkcja wektorowa trzech zmiennych

$u(x, y, z) = [\boxed{-z \cdot y,}\ \boxed{-z \cdot x]}$



Funkcja skalarna dwóch zmiennych

$g(x, y) = \sin(x) \cdot \cos(y)$



Funkcja wektorowa jednej zmiennej

$w(x) = [\boxed{\cos(x),}\ \boxed{\sin(x),}\ \boxed{1}\ ]$

$$\begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} \end{bmatrix}$$

Funkcja skalarna trzech zmiennych

$$f(x, y, z) = x^2 + y^2 + z^2$$

$$\begin{bmatrix} \frac{\partial u_1}{\partial x} & \frac{\partial u_1}{\partial y} & \frac{\partial u_1}{\partial z} \\ \frac{\partial u_2}{\partial x} & \frac{\partial u_2}{\partial y} & \frac{\partial u_2}{\partial z} \end{bmatrix}$$

Funkcja wektorowa trzech zmiennych

$$u(x, y, z) = [\, -z \cdot y, \;\; -z \cdot x \,]$$

$$\begin{bmatrix} \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix}$$

Funkcja skalarna dwóch zmiennych

$$g(x, y) = \sin(x) \cdot \cos(y)$$

$$\begin{bmatrix} \frac{\partial w_1}{\partial x} \\ \frac{\partial w_2}{\partial x} \\ \frac{\partial w_3}{\partial x} \end{bmatrix}$$

Funkcja wektorowa jednej zmiennej

$$w(x) = [\, \cos(x), \;\; \sin(x), \;\; 1 \,]$$

Różniczkowanie: jak program może różniczkować?

Teraz, załóżmy, że mamy funkcję mapującą $f : (\mathbb{R}^2 \to \mathbb{R})$:

```
function f(x::Real, y::Real)
  x^2 + y^2
end
```

Gradient i macierz Jacobiego funkcji skalarnej są tożsame:

$$\text{grad } f = \nabla(f) = \boldsymbol{J}_f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}$$

Macierz Hessego:

$$\boldsymbol{H}_f = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial xy} \\ \frac{\partial^2 f}{\partial yx} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix}$$

Jak **policzyć** jej macierz Jacobiego (lub macierz Hessego) *programistycznie*?

Metody dokładne

Różniczkowanie symboliczne

Pierwszym podejściem jest wykorzystanie programu komputerowego do symbolicznego przetwarzania wzorów matematycznych. W efekcie powstaje "wyrażenie", np. na pochodną funkcji.

```
using SymEngine

function f(x::Number, y::Number)
    x^2 + y^2
end
```

```
julia> x, y = symbols("x y")
julia> dfdx = diff(f(x,y), x)
julia> dfdy = diff(f(x,y), y)
```

```
julia> ∇f = [dfdx, dfdy]
2-element Vector{Basic}:
 2*x
 2*y

julia> subs.(∇f,
             x => 3,
             y => -2)
2-element Vector{Basic}:
  6
 -4
```

Różniczkowanie symboliczne

Niestety, nie wszystko da się w ten sposób przetworzyć (bo symbol nie ma wartości):

```
function g(x, y)
    r = 1.0
    for i=1:y
        r *= x
    end
    return r
end
```

```
julia> x, y = symbols("x y")
julia> dgdx = diff(g(x, y), x)
MethodError: no method matching (::Colon)(::Int64, ::Basic)
Closest candidates are:
  Any(::T, ::Any, !Matched::T) where T<:Real at range.jl:41
  Any(::A, ::Any, !Matched::C) where {A<:Real, C<:Real} at
range.jl:10
  Any(::T, ::Any, !Matched::T) where T at range.jl:40
  ...

Stacktrace:
 [1] g(::Basic, ::Basic) at ./In[1]:5
 [2] top-level scope at In[1]:13
```

Różniczkowanie symboliczne

A nawet, jeżeli się uda, to często wyrażenie jest **bardzo** skomplikowane:

```
function Babylonian(x; N = 10)
    t = (1+x)/2
    for i = 2:N; t=(t + x/t)/2  end
    return t
end

x = symbols("x")
diff( Babylonian(x; N=3), x ) |> expand |> display
```

Pochodna pierwiastka $\sqrt{x}$ wykorzystująca trzy iteracje algorytmu Babilońskiego:

```
1/8 + (-1/2)*x/(1 + 2*x + x^2) + (-1/2)*x/(1/4 + (1/2)*x +
2*x/(1 + x) +
4*x^2/(1 + x)^2 + 2*x^2/(1 + x) + (1/4)*x^2) - 2*x/((1 +
x)*(1/4 + (1/2)*x +
2*x/(1 + x) + 4*x^2/(1 + x)^2 + 2*x^2/(1 + x) + (1/4)*x^2)) +
2*x^2/((1 + 2*x +
x^2)*(1/4 + (1/2)*x + 2*x/(1 + x) + 4*x^2/(1 + x)^2 + 2*x^2/
(1 + x) + (1/4)*x^2)) +
(1/2)*(1 + x)^(-1) + (1/2 + (1/2)*x + 2*x/(1 + x))^(-1)
```

Podstawiając wartość x powinniśmy otrzymać przybliżenie pochodnej pierwiastka: $\frac{1}{2\sqrt{x}}$.

Różniczkowanie “ręczne”: reguła łańcuchowa

Najczęściej jesteśmy zainteresowani *wartością* pochodnej/gradientu, a nie *wyrażeniem*. Dlatego różniczkowanie symboliczne jest nieefektywne (bo daje nam za dużo).

Innym rozwiązaniem jest zastosowanie reguły łańcuchowej (*ang.* chain rule) liczenia pochodnych.

$f(x) = \sin(x^2)$ możemy zapisać jako złożenie (*ang.* composition) dwóch funkcji: $f(x) = \sin(g(x)), g(x) = x^2$.

Taki układ funkcji zapisujemy zwykle matematycznie $f \circ g$, natomiast w Julii:

```
f(y) = sin(y)
g(x) = x^2
(f ∘ g)(5.0)  == sin(5.0^2) # true
5.0 |> g |> f == sin(5.0^2) # true
```

$$f(g) = \sin(g), g(x) = x^2.$$

Zgodnie z regułą łańcuchową:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x},$$

czyli:

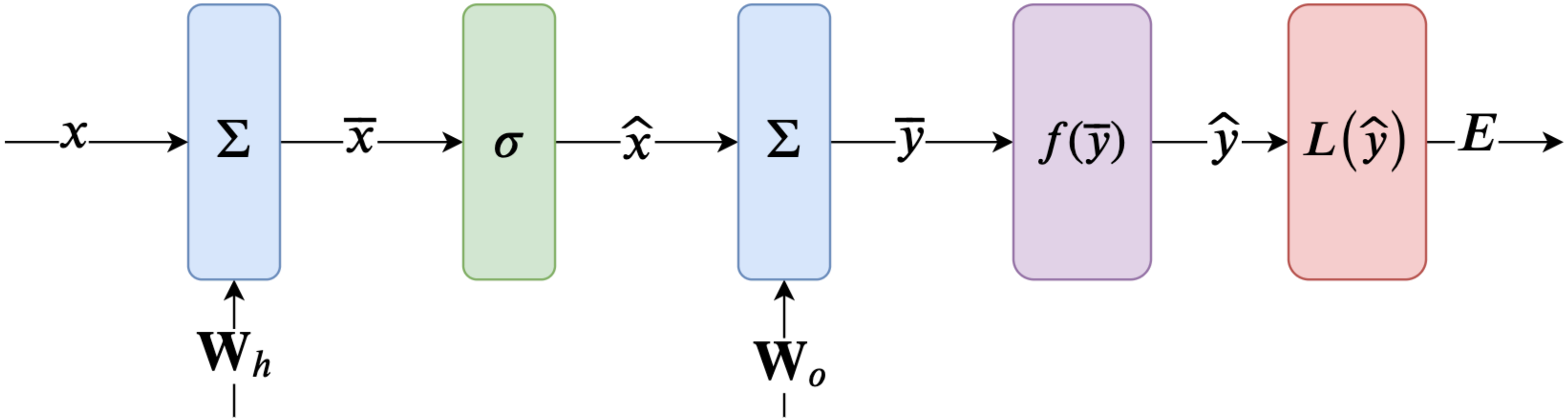
$$\frac{\partial f}{\partial x} = \cos(g(x)) \cdot 2x = \cos\left(x^2\right) \cdot 2x$$

```
f(g) = sin(g)
g(x) = x^2
dfdg(g) = cos(g)
dgdx(x) = 2x
dfdx(x) = dfdg(g(x)) *
          dgdx(x)
dfdx(5.0) ==
     cos(5.0^2)*(2*5.0)
```

Różniczkowanie “ręczne”: reguła łańcuchowa

Zaletą w stosunku do różniczkowania symbolicznego jest to, że otrzymujemy “wartość”, natomiast łańcuchowe stosowanie pochodnych cząstkowych zostawiamy naszemu programowi.

Gdzie spotykamy złożone funkcje? Sieć neuronowa może być postrzegana jako złożenie wielu, funkcji:



Możemy policzyć pochodne każdej z warstw, a potem w sposób łańcuchowy wyznaczyć, np. pochodną błędu E względem wag W_h.

Różniczkowanie symboliczne
i wykorzystujące regułę łańcuchową
nie przybliżają pochodnych.

Obydwie metody pozwalają na obliczenie
wartości gradientów z
dokładnością maszynową.

Metody przybliżone

Różniczkowanie numeryczne: różnica w przód

Jeżeli nie pamiętamy zbyt dobrze wykładów z analizy matematycznej, to możemy przybliżać numerycznie pochodne korzystając z samej definicji pochodnej, tj.: $f'(x_0) = \lim_{\Delta x \to 0} \frac{f(x_0+\Delta x)-f(x_0)}{\Delta x}$ Co możemy wyrazić w kodzie Julii:

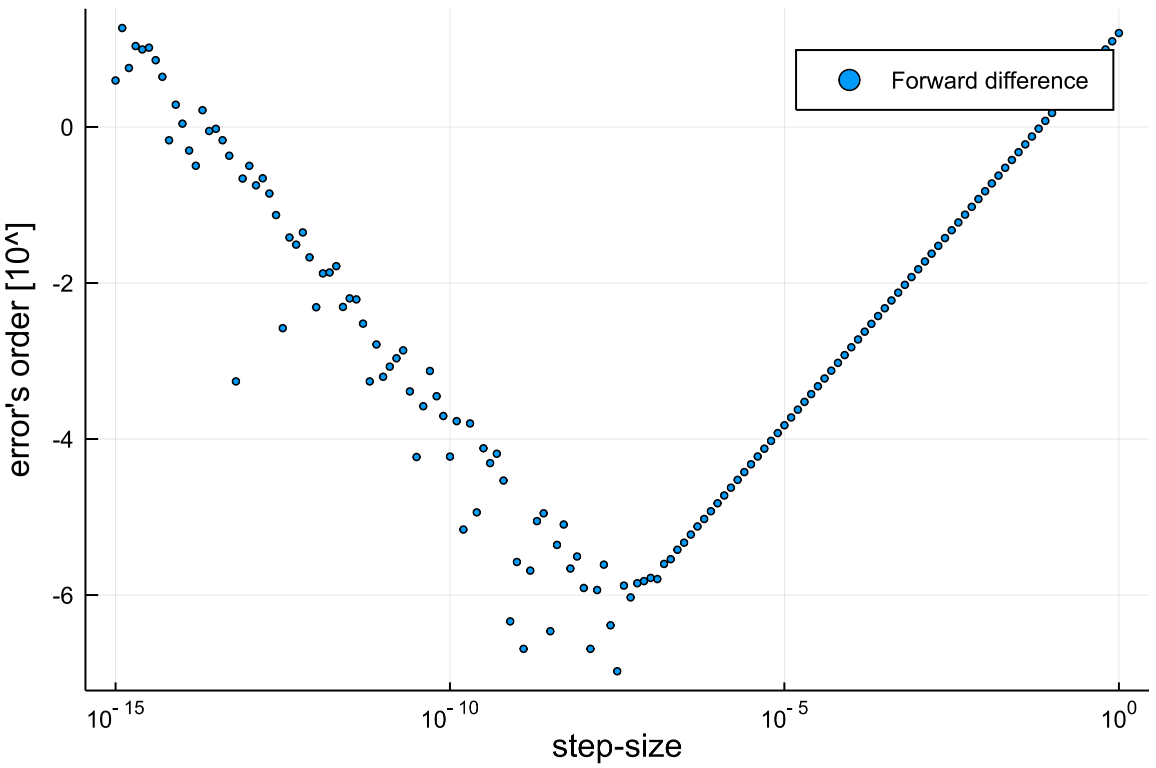
```
forward_diff(f, x_0; Δx=1e-3) = (f(x_0 + Δx) - f(x_0)) / Δx
```

W rzeczywistości `g(x, y)` oblicza wartość x^y, więc jego pochodna dana jest wzorem $y \cdot x^{y-1}$.

```
function g(x, y)
    r = 1.0
    for i=1:y
        r *= x
    end
    return r
end
```

```
julia> x_0, y_0 = 5.0, 3
julia> f(x) = g(x, y_0)
julia> dgdx = forward_diff(f,
                    x_0; Δx=1e-9)
75.0000265281415
julia> dg(x, y) = y * x^(y-1)
julia> dg(x_0, y_0)
75.0
```

Gdy krok jest bardzo mały, to wynik różnicy `f(x_0 + Δx) - f(x_0)` jest bardzo niedokładny:


Forward difference
error's order [10^]
0
-2
-4
-6
10^{-15}
10^{-10}
10^{-5}
10^{0}
step-size

Różniczkowanie numeryczne: różnice centralne

Błąd różnicy w przód jest rzędu $O(h)$ (co oznacza, że dwukrotne zmniejszenie kroku h, dwukrotnie zmniejsza błąd przybliżenia). Wynika to z rozwinięcia funkcji w szereg Taylora. Wartość funkcji $f(b)$ dana jest wzorem względem wartości funkcji (i jej pochodnych) w punkcie (a), czyli:

$$f(b) = f(a) + f'(a)(b-a) + \frac{1}{2!}f''(a)(b-a)^2 + \frac{1}{3!}f'''(a)(b-a)^3 + \cdots$$

Na podstawie rozwinięcia $f(x+h)$ i $f(x-h)$ względem $f(x)$ mamy:

$$f(x+h) = f(x) + hf'(x) + h^2\frac{1}{2!}f''(x) + h^3\frac{1}{3!}f'''(x) + \cdots$$

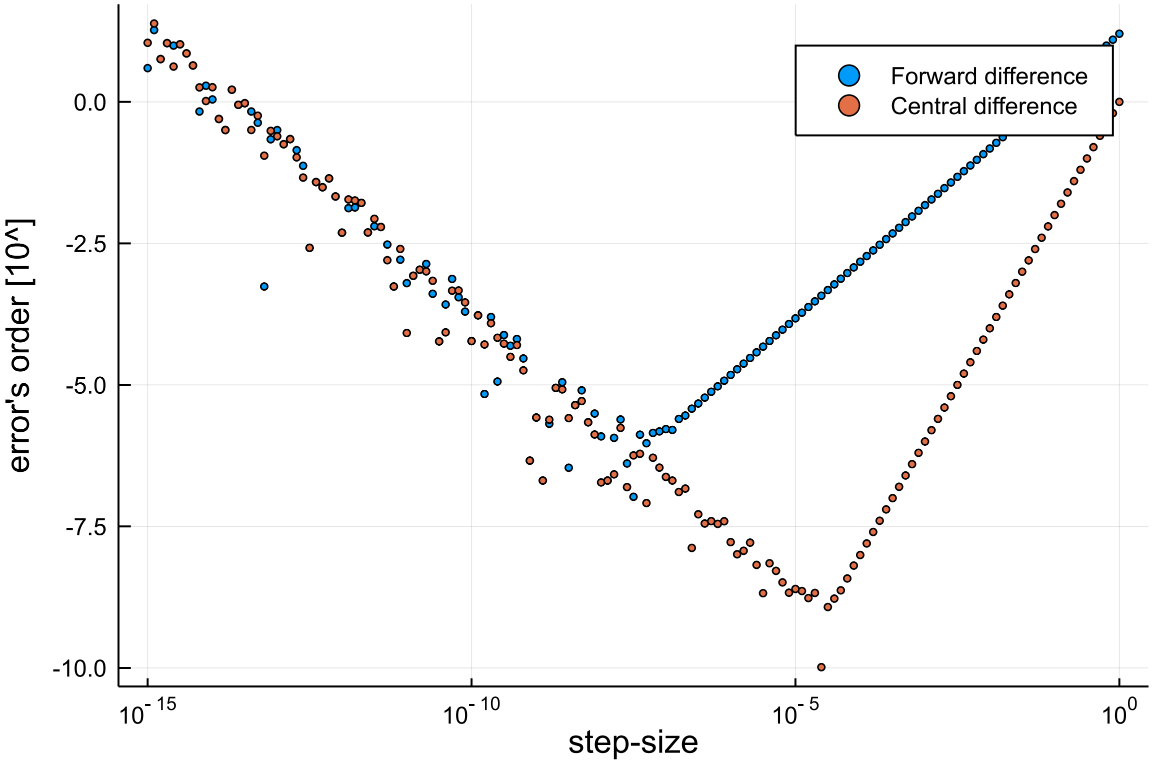
$$f(x-h) = f(x) - hf'(x) + h^2\frac{1}{2!}f''(x) - h^3\frac{1}{3!}f'''(x) + \cdots$$

Widać, że:

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{2}{3!}f'''(x)h^3 + \cdots$$

Przybliżając pierwszą pochodną widzimy, że błąd jest rzędu $O(h^2)$:

$$\frac{f(x+h) - f(x-h)}{2h} = f'(x) + h^2 \frac{1}{6} f'''(x) + \cdots$$


Forward difference
Central difference
0.0
-2.5
-5.0
-7.5
-10.0
error's order [10^]
10^-15
10^-10
10^-5
10^0
step-size

Różniczkowanie numeryczne: zespolona wartość kroku

Ciekawą metodą jest wykorzystanie **zespolonej** długości kroku, wtedy jeżeli $a = x$ oraz $b = x + ih$, to:

$$f(b) = f(a) + f'(a)(b - a) + \frac{1}{2!}f''(a)(b - a)^2 + \frac{1}{3!}f'''(a)(b - a)^3 + \cdots$$

$$f(x + ih) = f(x) + ihf'(x) - h^2\frac{1}{2!}f''(x) - ih^3\frac{1}{3!}f'''(x) + \cdots$$

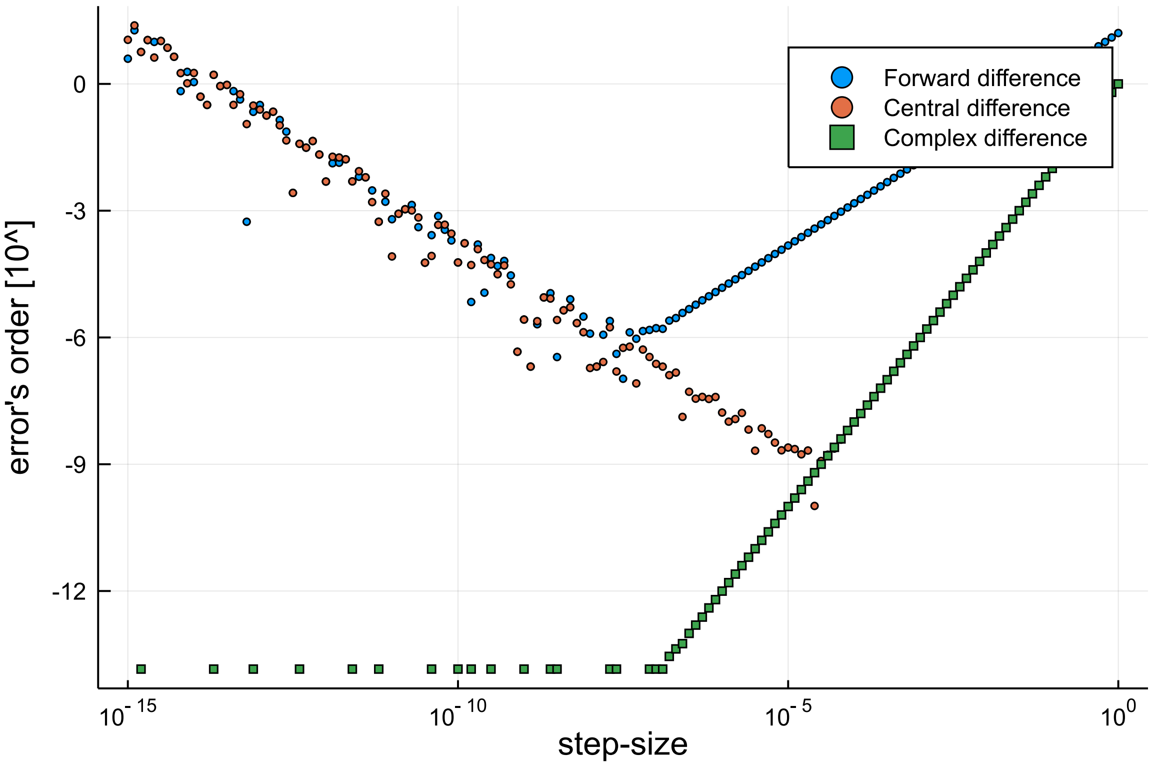
Widać, że część urojona $f(x + ih)$ upraszcza się do:

$$\Im(f(x + ih)) = hf'(x) - h^3 \frac{1}{3!} f'''(x) + \cdots \approx hf'(x)$$

Podobnie co w przypadku różnicy centralnej, przybliżając pierwszą pochodną widzimy, że błąd jest rzędu $O(h^2)$:

$$\frac{\Im(f(x + ih))}{h} = f'(x) - h^2 \frac{1}{6} f'''(x) + \cdots$$

…ale nie mamy już niedokładnego odejmowania:



Metoda różnic skończonych pozwala na liczenie **przybliżonej** wartości pochodnych.

Różnice centralne i te z zespolonym krokiem charakteryzują się zbieżnością **kwadratową**, podczas gdy różnica w przód – **liniową**.

Różnice centralne i w przód wymagają **dwóch** wywołań funkcji, przy bardzo małych wartościach kroku są **niedokładne**.

Różniczkowanie automatyczne

Metoda dokładna

Różniczkowanie automatyczne

A gdyby tak zamiast $i^2 = -1$ wybrać jednostkę taką, że $\varepsilon^2 = 0$?

Wtedy:

$$f(b) = f(a) + f'(a)(b-a) + \frac{1}{2}f''(a)(b-a)^2 + \frac{1}{3!}f'''(a)(b-a)^3 + \cdots$$

Jeżeli $a = x$ oraz $b = x + \varepsilon$, to mamy:

$$f(x+\varepsilon) = f(x) + f'(x)\varepsilon + \frac{1}{2}f''(x)\varepsilon^2 + \frac{1}{3!}f'''(x)\varepsilon^3 + \cdots$$

Zakładając, że $\varepsilon^2 = 0$, wszystkie człony oprócz dwóch pierwszych się **zerują**:

$$f(x + \varepsilon) = f(x) + f'(x)\varepsilon$$

Liczby dualne Wykorzystanie liczb zespolonych do liczenia pochodnych funkcji o wartościach *rzeczywistych* może być postrzegane jako *sztuczka*. Zamiast polegać na zachowaniu jednostki urojonej możemy zdefiniować **nowy** rodzaj liczb, tzw. liczby dualne (*ang.* dual numbers). Gdzie taka liczba jest dana w postaci $v + g \cdot \varepsilon$:

```
struct Dual <:Number
	v::Number
	g::Number
end
```

Nie jest to najbardziej efektywna implementacja[1], ale to dobry prototyp.

[1] `isabstracttype(Number) == true`

Aby takie liczby dualne działały, musimy zdefiniować dla nich podstawowe operacje:

```
import Base: +, -, *, /
-(x::Dual)              = Dual(-x.v,       -x.g)
+(x::Dual, y::Dual) = Dual( x.v + y.v, x.g + y.g)
-(x::Dual, y::Dual) = Dual( x.v - y.v, x.g - y.g)
*(x::Dual, y::Dual) = Dual( x.v * y.v, x.g*y.v + x.v*y.g)
/(x::Dual, y::Dual) = Dual( x.v / y.v,(x.g*y.v - x.v*y.g)/
y.v^2)
```

```
*(x::Dual, y::Dual) = Dual( x.v * y.v, x.g*y.v + x.v*y.g)
```

Powyżej, mnożenie odpowiada:

$$
\begin{aligned}
(v_x + g_x \cdot \varepsilon) \cdot \left(v_y + g_y \cdot \varepsilon\right) = v_x v_y + \\
g_x v_y \cdot \varepsilon + \\
v_x g_y \cdot \varepsilon + \\
g_x g_y \cdot \varepsilon^2 \\
= v_x v_y + \left(g_x v_y + v_x g_y\right) \cdot \varepsilon
\end{aligned}
$$

Zdefiniujemy też wyniki podstawowych funkcji:

```
import Base: abs, sin, cos, tan, exp, sqrt, isless
abs(x::Dual)  = Dual(abs(x.v),sign(x.v)*x.g)
sin(x::Dual)  = Dual(sin(x.v), cos(x.v)*x.g)
cos(x::Dual)  = Dual(cos(x.v),-sin(x.v)*x.g)
tan(x::Dual)  = Dual(tan(x.v), one(x.v)*x.g + tan(x.v)^2*x.g)
exp(x::Dual)  = Dual(exp(x.v), exp(x.v)*x.g)
sqrt(x::Dual) = Dual(sqrt(x.v),.5/sqrt(x.v) * x.g)
isless(x::Dual, y::Dual) = x.v < y.v;
```

…oraz reguły konwersji i promocji typów:

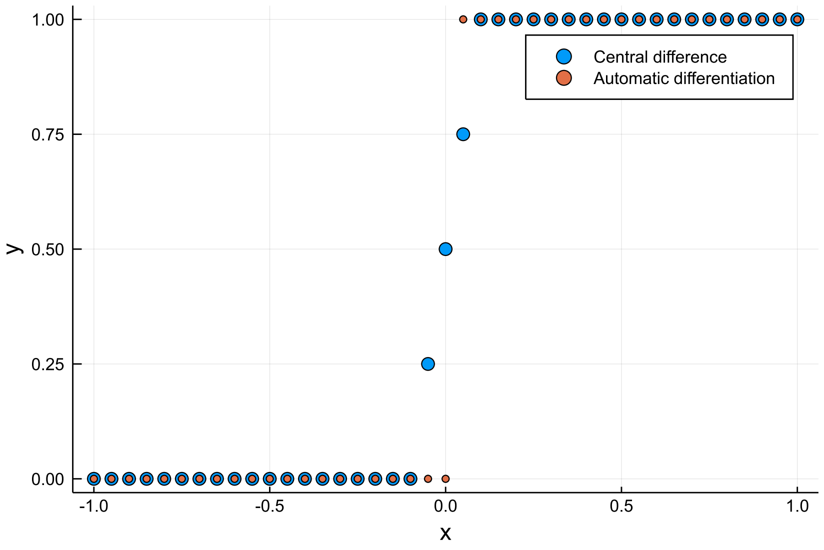
```
# Promocja typów i konwersja
import Base: convert, promote_rule
convert(::Type{Dual}, x::T) where T<:Real = Dual(x, zero(x))
promote_rule(::Type{Dual}, ::Type{T}) where T<:Real = Dual
```

Dzięki temu możemy zrobić: `Dual[2, Dual(1,1)]`, co jest równoważne `[Dual(2, 0), Dual(1, 1)]`.

Policzymy pochodną funkcji $f(x) = \max(0, x)$.

```
f(x) = x > zero(x) ? x : zero(x)
ϵ = Dual(0., 1.) # zarodek/seed

x = -1.0:0.05:+1.0
y = f.(x + ϵ)
```



Podsumowanie

Literatura

- Atılım Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, Jeffrey Mark Siskind, 2018, "Automatic Differentiation in Machine Learning: a Survey", https://arxiv.org/pdf/1502.05767.pdf, dostęp: 20.03.2020
- Mykel J. Kochenderfer, Tim A. Wheeler, 2019, Algorithms for Optimization, The MIT Press
- Alan Edelman, 2018, "Automatic Differentiation in 10 minutes with Julia", url: https://github.com/JuliaAcademy/JuliaAcademyMaterials/blob/master/Courses/-Foundations%20of%20machine%20learning/20.Automatic-Differentiation-in-10-Minutes.jl, dostęp: 20.03.2020

- Joaquim Martins, Peter Sturdza, Juan Alonso, 2003, "The complex-step derivative approximation", ACM Transactions on Mathematical Software, Association for Computing Machinery, 29, pp.245 - 262. 10.1145/838250.838251
- Sören Laue, 2019, "On the Equivalence of Forward Mode Automatic Differentiation and Symbolic Differentiation", url: https://arxiv.org/pdf/1904.02990.pdf, dostęp 23.03.2020
- Mike Innes, 2019, "Differentiation for Hackers: Implementing Forward Mode", url: https://github.com/MikeInnes/diff-zoo/blob/notebooks/forward.ipynb, dostęp: 25.03.2020

- Jarrett Revels, Miles Lubin, Theodore Papamarkou, 2016, “Forward-Mode Automatic Differentiation in Julia”, url: https://arxiv.org/abs/1607.07892, dostęp: 23.03.2020
- Jarrett Revels, 2017, “How ForwardDiff Works”, url: https://github.com/JuliaDiff/ForwardDiff.jl/blob/master/docs/src/dev/how_it_works.md, dostęp: 24.03.2020
- Jeffrey Mark Siskind, Barak A. Pearlmutter, 2005, “Perturbation Confusion and Referential Transparency: Correct Functional Implementation of Forward-Mode AD”, url: http://www.bcl.hamilton.ie/~barak/papers/ifl2005.pdf, dostęp: 25.03.2020

Dziękuję za uwagę