Automatyczne różniczkowanie

(Czerwiec 2021)

K.Wyszyński, A.Sienkiewicz

Strzeszczenie— Niniejszy dokument zawiera wprowadzenie do tematyki automatycznego różniczkowania na podstawie porównania działania dwóch algorytmów – trybu różniczkowania w przód (*ang. Forward mode automatic differentiation)* oraz w tył (*ang. Reverse mode automatic differentiation)*. Powyższe metody przetestowane zostały w tych samych warunkach pod względem dokładności wyznaczonych pochodnych, ilości alokowanej pamięci podczas działania oraz czasu wyznaczenia macierzy Jacobiego dla obliczanych wyrażeń. Praca skupia się na znalezieniu odpowiedzi na pytanie: który z algorytmów jest lepszym rozwiązaniem w określonych okolicznościach.

Słowa kluczowe— automatyczne różniczkowanie, algorytm, pochodna, w przód, w tył

1 WSTĘP

Różniczkowanie za pomocą programów komputerowych powstało z oczywistych powodów – człowiek nie jest w stanie samodzielnie poradzić sobie z wyznaczaniem pochodnych funkcji złożonych. W związku z tym powstają różne podejścia starające się efektywnie rozwiązać problem różniczkowania. Jednym z nich jest różniczkowanie symboliczne – wykorzystanie programu komputerowego do symbolicznego przetwarzania wzorów matematycznych. Metoda ta nie jest jednak skuteczna – może prowadzić do generacji bardzo skomplikowanych wyrażeń. Innym podejściem może być różniczkowanie numeryczne, czyli przybliżanie numeryczne pochodnej na podstawie jej definicji. Należy jednak zauważyć, że metoda ta bazuje na przybliżeniach, które mogą wpływać na dokładność wyznaczonych wartości. Dodatkowo cechuje się brakiem wydajności przy dużej ilości danych.

Odpowiedzią na niedoskonałości powyższych pomysłów jest różniczkowanie automatyczne, którego filarem jest założenie, że wszelkie skomplikowane działania można ostatecznie sprowadzić do prostych operacji. Jako operacje proste należy rozumieć funkcje umożliwiające wyznaczanie prostych, podstawowych pochodnych, z których następnie program korzystać będzie podczas realizacji bardziej złożonych działań. Podejście to pozwala na różniczkowanie z dokładnością maszynową, co daje mu przewagę nad wyznaczaniem pochodnych metodami numerycznymi.

1.1 Różniczkowanie automatyczne w przód

Metoda automatycznego różniczkowania w przód opiera swoje działanie na wprowadzeniu liczby dualnej. Jest to liczba składająca się z dwóch składowych – wartości rzeczywistej oraz pochodnej. Przeciążając operatory dla takiej liczby (definiując ręcznie wyznaczanie prostych pochodnych) można automatycznie różniczkować złożone wyrażenia z dokładnością maszynową[4].

1.2 Różniczkowanie automatyczne w tył

Metoda automatycznego różniczkowania w tył wykorzystuje grafy obliczeniowe. Jej działanie opiera się na obliczeniu wyrażenia przy jednoczesnym zapamiętywaniu operacji cząstkowych podczas realizacji obliczeń. Algorytm tworzy graf obliczeniowy, który reprezentuje zależności między zmiennymi, gdzie liście grafu to zmienne, a węzły to operacje na nich[7]. Następnie, przemieszczając się po stworzonym w ten sposób grafie obliczeniowym w tył, wykorzystuje zapamiętane składowe do wyznaczania gradientu zadanej funkcji.

1.3 Różniczkowanie automatyczne w przód – wydajniejsza wersja

Istnieje lepsze podejście do stosowania automatycznego różniczkowania w przód niż to, które zostanie zaprezentowane w niniejszym artykule. Opiera się ono na stosowaniu wielowymiarowych liczb dualnych, co wpływa bezpośrednio na wydajność algorytmu. Ponieważ podczas automatycznego różniczkowania w przód zarodek musi być dodany do kolejnych argumentów różniczkowanej funkcji, w klasycznym rozwiązaniu z

liczbami dualnymi jednowymiarowymi wymaga to tylu przejść algorytmu, ile argumentów przyjmuje funkcja różniczkowana. Zastosowanie wielowymiarowych liczb dualnych, a więc takich składających się z większej liczby zarodków, pozwala na obejście tego problemu. Wpływa to pozytywnie na wydajność algorytmu w przypadku większej liczby argumentów. Metoda ta nie jest jednak przedmiotem badań w ramach przedstawionej pracy.

2 Opis badania

2.1 Parametry środowiska uruchomieniowego

Algorytmy badane w projekcie zostały zaimplementowane w języku Julia. Eksperymenty przeprowadzono na sprzęcie o następujących parametrach:

Pamięć RAM: 8GB DDR3
CPU: Intel® Core™ i5-2400 CPU @ 3.10GHz
Dysk: GOODRAM SSD 232 GB

W celu zbadania wydajności algorytmów wykorzystano pakiet *BenchmarkTools*, który wyznaczał średnie wartości z 10000 prób dla każdego badanego przypadku.

2.2 Założenia teoretyczne

Algorytm automatycznego różniczkowania w przód przetwarza wszystkie argumenty wejściowe badanych funkcji na liczby dualne[6][3]. Wiąże się to z istotną konsekwencją – złożoność obliczeniowa, a tym samym wydajność algorytmu zależna jest od liczby argumentów wejściowych funkcji. Rosnąca liczba argumentów wejściowych wpłynie negatywnie na czas wykonywania algorytmu[2].

Algorytm automatycznego różniczkowania w tył wywołuje funkcję tyle razy, ile wyników zwraca ta funkcja. Wiąże się to z istotną konsekwencją – złożoność obliczeniowa, a tym samym wydajność algorytmu zależna jest od liczby wyników zwracanych przez funkcję[1]. Rosnąca liczba wyników wpłynie negatywnie na czas wykonywania algorytmu.

Algorytmy automatycznego różniczkowania wyznaczają pochodne z dokładnością maszynową, co daje im przewagę nad metodami numerycznymi bazującymi na przybliżeniach[7].

2.3 Przypadki testowe

Kierując się założeniami z punktu 2.2 dobrym przypadkiem porównawczym będzie zestawienie ze sobą dwóch skrajnych sytuacji – pierwszą z nich będzie funkcja przyjmująca wiele argumentów i zwracająca jedno wyjście. Druga funkcja zwróci wiele wyjść dla pojedynczego argumentu. Dodatkowo zbadany został przypadek pośredni, w którym badana funkcja zwracała tyle wyników, ile otrzymała argumentów wejściowych.

W ramach testów zaimplementowane zostały algorytmy generujące macierz Jacobiego za pomocą automatycznego różniczkowania metodami w przód oraz w tył[5]. Macierz Jacobiego to macierz pochodnych cząstkowych wszystkich funkcji będących argumentami wejściowymi funkcji generującej.

Do testów wykorzystana została następująca funkcja: $f(x) = e^{\frac{\sin x^2}{x^2}}$. Kolejne badania różniły się od siebie liczbą argumentów przyjmowanych przez funkcję oraz liczbą zwracanych przez nią wyników.

Test pierwszy polegał na wyznaczeniu macierzy Jacobiego dla n funkcji f(x), przyjmującej jako argument pojedynczą wartość x = 0.323. Jest to sytuacja, w której zgodnie z założeniami lepiej powinien zachować się algorytm automatycznego różniczkowania w tył.

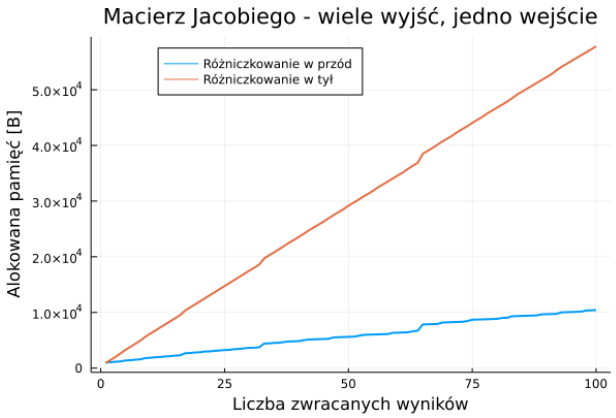
Test drugi to podejście odwrotne do poprzedniego badania. Tym razem wyznaczona została macierz Jacobiego dla pojedynczej funkcji f(x), przyjmującej jako argument n-elementowy wektor liczb pseudolosowych. W tym przypadku, zgodnie z założeniami algorytm automatycznego różniczkowania w przód powinien okazać się bardziej wydajny.

Test trzeci to przypadek pośredni – w tym badaniu macierz Jacobiego została wyznaczona dla n funkcji f(x) przyjmujących jako argument n-elementowy wektor liczb pseudolosowych. Spodziewanym rezultatem jest zbliżona wydajność obu algorytmów, przy większej alokacji pamięci w przypadku algorytmu różniczkowania w tył – wynika to z faktu zapamiętywania przez algorytm operacji cząstkowych.

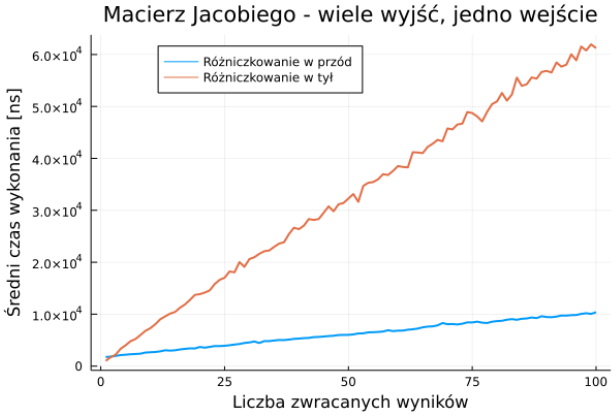
Badania objęły również sprawdzenie dokładności pochodnych wyznaczonych przez algorytmy. W tym celu wykorzystano funkcję ReLU, dla danych z przedziału [-1,1], którą obliczono za pomocą algorytmów automatycznych oraz podejścia numerycznego – metody różnic centralnych, dla której w celach demonstracyjnych zastosowana została wartość kroku wynosząca 0.1. Funkcja ReLU wyraża się następującym wzorem: $f(x) = \max(0, x)$. Różniczkowanie takiej funkcji zwrócić może zatem jedną z dwóch wartości: 0 lub 1. Oczekiwanym wynikiem jest uzyskanie prawidłowych wartości przez algorytmy automatyczne, w przeciwieństwie do metody różnic centralnych, która w okolicy punktu x=0 okaże się niedokładna.

3 Uzyskane rezultaty

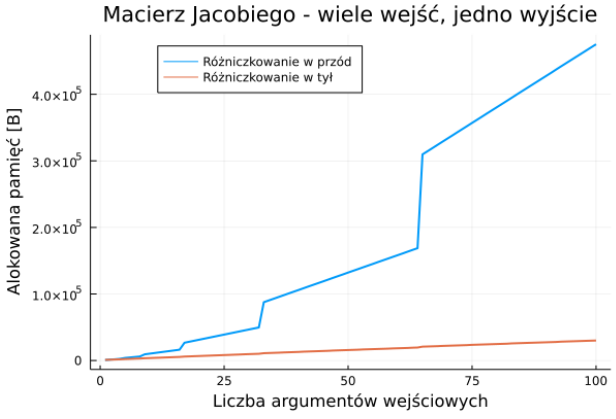
Poniższe wykresy przedstawiają porównanie średniego czasu pracy obu algorytmów oraz ilości alokowanej pamięci dla przypadków testowych zaproponowanych w punkcie 2.3:



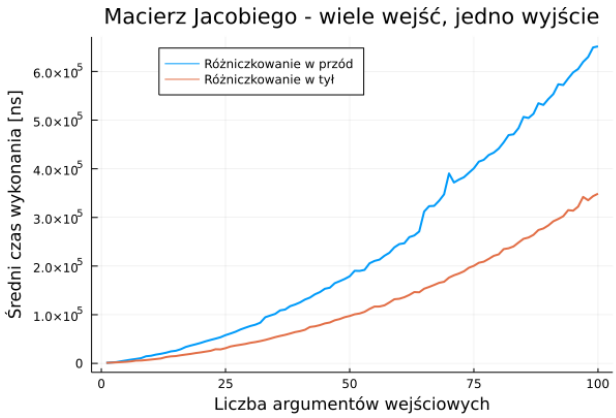
Rys. 1 Wykres przedstawiający zależność alokowanej pamięci od liczby zwracanych wyników funkcji jednoargumentowej



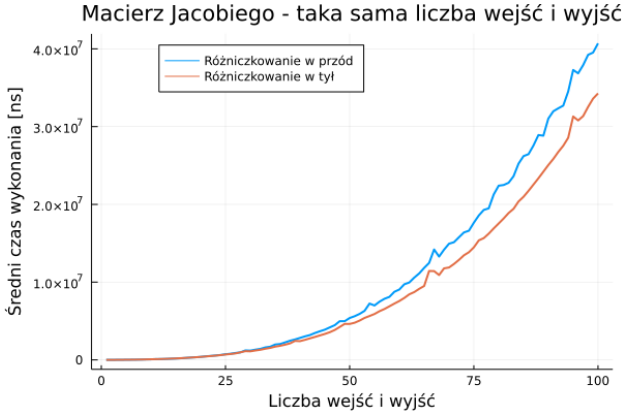
Rys. 2 Wykres przedstawiający zależność średniego czasu wykonywania algorytmu od liczby zwracanych wyników funkcji jednoargumentowej



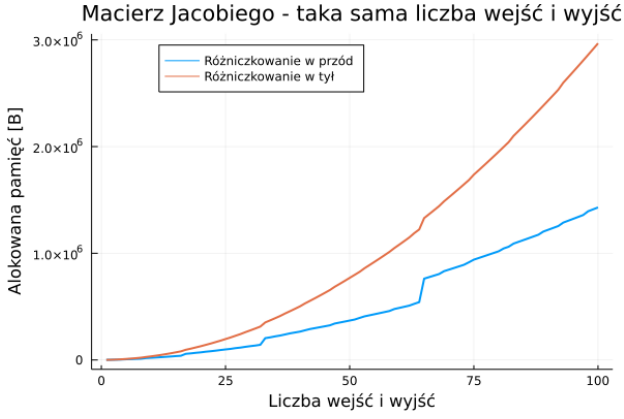
Rys. 3 Wykres przedstawiający zależność alokowanej pamięci od liczby argumentów wejściowych funkcji wieloargumentowej



Rys. 4 Wykres przedstawiający zależność średniego czasu wykonywania algorytmu od liczby argumentów wejściowych funkcji wieloargumentowej



Rys. 5 Wykres przedstawiający zależność średniego czasu wykonywania algorytmu od n dla n funkcji n-argumentowych

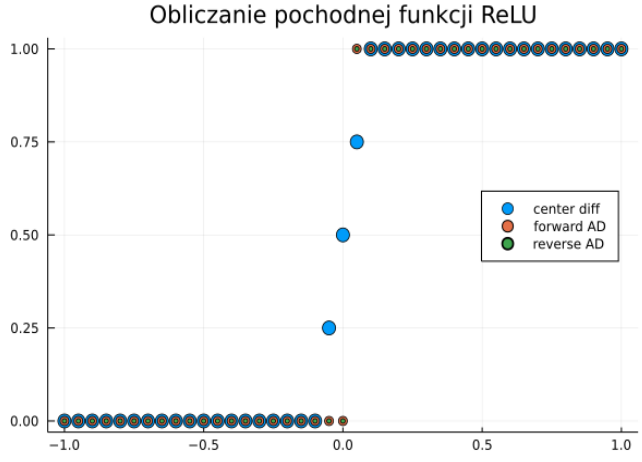


Rys. 6 Wykres przedstawiający zależność alokowanej pamięci od n dla n funkcji n-argumentowych

Tabela 1 Otrzymane wyniki działania algorytmów generacji macierzy Jacobiego dla różnych przypadków testowych

Przypadki testowe	Różniczkowanie w przód		Różniczkowanie w tył	
	średni czas wykonania [ms]	alokowana pamięć [MiB]	średni czas wykonania [ms]	alokowana pamięć [MiB]
Sto wyjść, jedno wejście	0.010816	0.009932	0.070311	0.055117
Sto wejść, jedno wyjście	0.760086	0.453184	0.366784	0.029404
Taka sama liczba wejść i wyjść	43.00800	1.36	39.85200	2.91

Poniższy wykres przedstawia uzyskane wyniki różniczkowania funkcji ReLU dla trzech koncepcji: algorytmu różniczkowania w przód, w tył oraz numerycznej metody różnic centralnych:



Rys. 7 Wykres przedstawiający otrzymane wyniki różniczkowania funkcji ReLU przy zastosowaniu metod automatycznego różniczkowania oraz metody różnic centralnych.

Na podstawie wyników przedstawionych na wykresach oraz w tabeli 1. można wysunąć następujące wnioski:

1. Średni czas wykonania algorytmu różniczkowania w tył był około 7-krotnie większy dla funkcji zwracającej sto wyników niż w przypadku metody w przód. Zaalokowano również około 5-krotnie więcej pamięci.
2. Średni czas wykonania algorytmu różniczkowania w przód był około 2-krotnie większy dla funkcji przyjmującej 100 argumentów niż w przypadku metody w tył. Metoda w przód zaalokowała około 15 razy więcej pamięci.
3. W przypadku funkcji o takiej samej liczbie argumentów i zwracających wyników metoda w tył zaalokowała ok. 2-krotnie więcej pamięci.

5 WNIOSKI

Na podstawie uzyskanych rezultatów można wywnioskować słuszność postawionych wstępnie założeń teoretycznych.

Algorytm automatycznego różniczkowania w przód jest rozwiązaniem optymalnym dla funkcji przyjmujących niewielką liczbę argumentów wejściowych. W przypadku funkcji przyjmującej wiele argumentów lepszym rozwiązaniem jest zastosowanie automatycznego różniczkowania w tył.

Algorytm automatycznego różniczkowania w tył jest optymalny w sytuacji odwrotnej, kiedy to funkcja zwraca niewielką liczbę wyników. Należy jednak zwrócić uwagę na fakt, iż jest to rozwiązanie kosztowne pod względem alokowanej pamięci.

Przeprowadzone testy potwierdzają precyzję maszynową wyznaczanych przez algorytmy pochodnych. Podczas eksperymentów przedstawiona została sytuacja, w której numeryczna metoda różnic centralnych nie uzyskała poprawnego wyniku, w przeciwieństwie do zaimplementowanych rozwiązań.

W przypadku jednakowej liczby wejść i wyjść, oba podejścia prezentują porównywalną wydajność, z niewielką przewagą algorytmu automatycznego różniczkowania w tył. Przewaga czasowa opłacona jest jednak kosztem alokowanej pamięci, która jest znacząco większa w metodzie różniczkowania w tył.

Jak wspomniane zostało w punkcie 1.3., możliwe jest uzyskanie lepszych rezultatów w przypadku automatycznego różniczkowania w przód z wykorzystaniem wielowymiarowych liczb dualnych.

BIBLIOGRAFIA

[1] Roger Luo, 2018, "Implement Your Own Automatic Differentiation with Julia in ONE day",url: http://blog.rogerluo.me/2018/10/23/write-an-ad-in-one-day/, dostęp: 03.05.2021

[2] Mike Innes, 2019, "Differentiation for Hackers: Implementing Forward Mode", url:https://github.com/MikeInnes/diff-zoo/blob/notebooks/forward.ipynb, dostęp: 01.05.2021

[3] Jarrett Revels, Miles Lubin, Theodore Papamarkou, 2016, "Forward-Mode AutomaticDifferentiation in Julia", url: https://arxiv.org/abs/1607.07892, dostęp: 03.05.2021

[4] Jarrett Revels, 2017, "How ForwardDiff Works", url:https://github.com/JuliaDiff/ForwardDiff.jl/blob/master/docs/src/dev/how_it_works.md,dostęp: 03.05.2021

[5] Bartosz Chaber, 2021, Repozytorium z materiałami do przedmiotu 1di2153, url: https://github.com/bchaber/1di2153, dostęp: 03.05.2021

[6] Jarrett Revels, 2015, „JSoC 2015 project: Automatic Differentiation in Julia with ForwardDiff.jl", url: https://julialang.org/blog/2015/10/auto-diff-in-julia/, dostęp: 03.05.2021

[7] Robert Szmurło, Bartosz Chaber, 2021, Materiały do kursu Algorytmy w inżynierii danych