Różniczkowanie algorytmiczne

Jerzy Karczmarczuk
Zakład Informatyki, Uniwersytet w Caen, Francja

1. Wstęp

Rachunek różniczkowy i całkowy stanowią chleb powszedni każdego fizyka od najmłodszych lat i nie ma sensu robić tutaj „reklamy" analizy matematycznej. Nie każdy uczeń umie różniczkować, ale każdy student od pierwszego roku oblicza pochodne. Ten tekst jest przeznaczony dla czytelników, dla których rachunek różniczkowy nie stanowi żadnej zagadki, którzy znają podstawy teoretyczne, oraz mają sensowne doświadczenie praktyczne. Mimo banalności poniższych przykładów, celowe będzie podanie kilku kontekstów, w których student musi, i to dość wcześnie, użyć swojej znajomości rachunku różniczkowego.

1. Rachunek niepewności pomiarowych na pracowni. Pochodne służą do oszacowania niedokładności wartości, które są funkcjami zmiennych niezależnych (czyli wielkości mierzonych bezpośrednio).
2. Obliczenia zależności między natężeniem prądu a napięciem w obwodach zawierających oporniki, cewki i kondensatory.
3. Konstrukcja równań ruchu przez różniczkowanie funkcjonałów Lagrange'a czy Hamiltona. Cała mechanika klasyczna jest na tym oparta.
4. Obliczanie relacji między wielkościami termodynamicznymi. Cała ich klasa oparta jest na formach różniczkowych.
5. Rachunki perturbacyjne we wszystkich znanych działach fizyki.

Jest to oczywiście bardzo niekompletna lista. Zauważmy, że obliczanie pochodnych pełni dwie związane ze sobą, ale w dużej mierze rozłączne funkcje: czasami potrzebujemy jawną, funkcyjną postać pochodnej, aby ją analizować, przekształcać i wprowadzać do innych wyrażeń, nad którymi pracujemy, ale niezmiernie często potrzebujemy jedynie wartości numerycznych tych pochodnych. W rachunku błędów towarzyszącemu opracowaniu zadań laboratoryjnych, pochodne jako funkcje niczemu nie służą, potrzeba nam tylko pojedynczych wartości liczbowych.

Rachunek zaburzeń służy zwykle także do otrzymania konkretnych wyników liczbowych. W technologii pochodne są potrzebne, aby np. obliczyć stabilność pewnych wartości, np. sprawności jakiejś maszyny względem zmian jej parametrów. I tu jawna postać funkcji pochodnej jest mało interesująca, chcemy mieć tylko wykres wartości liczbowych w funkcji parametrów.

Ale czy da się obliczać pochodne bez przejścia przez formy symboliczne określające zależność jakiegoś wyrażenia od zmiennej różniczkowania? Oczywiście można wykonać przybliżone rachunki numeryczne, skorzystać z przybli-

żenia pochodnej przez iloraz różnicowy. Wiemy że pochodna bywa definiowana jako

$$y'(x) = \lim_{\delta \to 0} \frac{y(x+\delta)-y(x)}{\delta} \tag{1.1}$$

lub wzorami zbliżonymi. Przybliżenia wykorzystujące formuły takie jak powyżej ze skończoną wartością δ, nie są zbyt dokładne i są numerycznie niestabilne, gdy δ dąży do zera, błąd jest trudno kontrolować. Wyższe pochodne obarczone są jeszcze większym i źle zachowującym się błędem.

Wydaje się więc, że mamy do wyboru: albo użyć niedokładnych metod numerycznych, albo kosztownych czasowo a czasami niepotrzebnych metod analitycznych, ręcznie, lub z użyciem pakietów do obliczeń symbolicznych, jak Maple, Mathematica, czy Axiom. Okazuje się jednak, że nie jest to prawdą. Poniżej opiszemy jedną z technik (są i inne) tzw. różniczkowania algorytmicznego, albo automatycznego (nazwa historyczna), pozwalającą na różniczkowanie numeryczne, ale bez przybliżeń różnicowych. Pozwoli to w wielu wypadkach uniknąć używania języków do symbolicznych rachunków algebraicznych, które bywają po prostu nadużywane: stosuje się je do obliczania pochodnych, które mają być jedynie użyte w czysto numerycznych programach w Fortranie czy języku „C", i ten dokonuje końcowych obliczeń. Takie automatycznie utworzone wyrażenia są często źle uproszczone, nieefektywne, gdyż wyrażenie czytelne dla człowieka niekoniecznie jest zoptymalizowane pod względem rachunkowym. Tak więc, takie dwuetapowe implementacje często marnują czas, zarówno człowieka jak i maszyny.

2. Różniczkowanie to jest algebra

Różniczkowanie wyrażeń zawierających funkcje elementarne i niektóre specjalne nie wymaga obliczania żadnych granic. Korzystamy z „przepisów kulinarnych", np. $(x^n)' = nx^{n-1}$, $(\cos f(x))' = -\sin f(x) \cdot f'(x)$ itd. Są to lokalne, punktowe (dla jednej wartości) operacje na wyrażeniach. Na tym opiera się automatyzacja obliczeń pochodnych przez pakiety symboliczne. Pakiety te operują *formami*, a nie liczbami.

Ale ogólne zasady dotyczące różniczkowania mogą być ujęte bardziej *abstrakcyjnie*. Czytelnik nie powinien się zaniepokoić, okaże się zaraz, że ta abstrakcja jest czytelna dla studenta pierwszego roku i niezwykle praktyczna, umożliwiająca różniczkowanie numeryczne, ale **szybkie i dokładne** za pomocą standardowych języków programowania. Dokładność obliczeń pochodnych będzie taka sama jak dokładność maszynowa standardowych operacji arytmetycznych i obliczania wartości funkcji.

Nie będziemy operować (wyłącznie) funkcjami, ale *wyrażeniami* zależnymi – dla uproszczenia – od jednej zmiennej niezależnej, którą możemy nazywać x, ale jej nazwa nie ma żadnego znaczenia, będzie to *wartość o charakterze licz-*

bowym (ale, jak się okaże, będzie to obiekt nieco bardziej skomplikowany niż zwykła liczba). Wyrażenia zawierają tę zmienną, szereg stałych, np. 1, czy π, i wszystko jest połączone zwykłymi działaniami arytmetycznymi. Przypadek wielu zmiennych zostanie omówiony później.

Do tych działań dołączymy jedną specjalną operację, lokalną (punktową, określoną dla jednej konkretnej wartości x), oznaczaną przez ∂, którą nazwiemy derywacją (a czasami po prostu różniczkowaniem...). W pewnym uproszczeniu wystarczy zażądać, aby ta operacja była liniowa, tj. dla dowolnych wyrażeń e, f: $\partial(e+f) = \partial e + \partial f$, oraz spełniała warunek Leibniza: $\partial(e \cdot f) = \partial e \cdot f + e \cdot \partial f$. To wszystko, łatwo sprawdzić, że jeśli $e = f/g$, a więc $e \cdot g = f$, czyli $\partial e \cdot g + e \cdot \partial g = \partial f$, dostaniemy $\partial e = (\partial f - e \cdot \partial g)/g$, co można zapisać jako $\partial e = (g \cdot \partial f - f \cdot \partial g)/g^2$, jest to znany wzór na różniczkowanie ilorazu.

Zauważmy, że w ogólności nie wiemy czym są nasze wyrażenia i *co to jest* derywacja, znamy tylko ich ogólne własności. To wystarczy, aby stwierdzić np., że $\partial(e^2) = 2 \cdot e \cdot \partial e$, i wyprowadzić zasady różniczkowania wielomianów i wyrażeń wymiernych bez żadnych operacji obliczania granic. Wszystko sprowadza się do działań algebraicznych.

Dziedzina „wyrażeń" może być dowolna, np. mogą to być liczby całkowite. Wtedy jednak derywacja jest operacją trywialną. Ponieważ $e + 0 = e$, widzimy, że $\partial 0 = 0$. Ale także, ponieważ $e \cdot 1 = e$, równość Leibniza wymaga, aby $\partial e + e \cdot \partial 1 = \partial e$, skąd wynika, że $\partial 1 = 0$. Stąd, poprzez dodawanie, $\partial n = 0$ dla dowolnej liczby całkowitej n, a poprzez różniczkowanie ilorazu widzimy, że derywacja dowolnej liczby wymiernej także się zeruje. Ponieważ na komputerze mamy jedynie liczby wymierne, wygląda, że nasza abstrakcja na nic się nie przyda!

Tak jednak nie jest. Oczywistym przykładem są formy, struktury składniowe, które przedstawiają wyrażenia symboliczne. Programy komputerowe potrafią je różniczkować podobnie jak my to robimy na papierze. Wyrażenia te zawierają stałe liczbowe i referencje do operacji takich jak mnożenie, czy funkcji sinus. Operacje te nie są wykonywane, tylko identyfikowane. Wyrażenia zawierają również referencje do jakichś „zmiennych różniczkowania", które mają sens *nazw*. Nie należy jednak sądzić, że to, o co chodzi, to obecność nazw czy symboli. Ogólnie rzecz biorąc, wyrażenia po prostu winny stanowić algebrę na tyle skomplikowaną, że różniczkowanie (czyli derywacja) staną się nietrywialne.

Dla nas wystarczy, abyśmy operowali parami liczb, oznaczanymi $e = D(e_0, e_1)$. (D jest tu jedynie etykietą odróżniającą te pary od innych). Pary liczb w rachunkach występują wszędzie. Liczby wymierne przedstawia się jako pary złożone z licznika i mianownika. Liczby zespolone są parami zawierającymi część rzeczywistą i urojoną. Rachunki geometryczne na płaszczyźnie operują wektorami dwywymiarowymi. Nasze pary są jeszcze inne z punktu widzenia działań na nich.

Tutaj e_0 i e_1 będą dowolnymi liczbami (zwykłymi), które mogą stanowić elementy wyrażeń w naszych programach komputerowych. Język programowania, którym operujemy winien jednak umożliwić zdefiniowanie operacji arytmetycznych na takich parach, zwykle oznaczanych jako (a,b). Większość współczesnych języków programowania na to pozwala. Naszym językiem wzorcowym będzie Python, dość popularny w środowiskach naukowych i w dydaktyce programowania.

Nasz uogólniony model obliczeń sprawdzi się do kilka prostych przepisów, o następującej intuicji: e_0 to jest „wartość główna" wyrażenia liczbowego, wartość, która byłaby jedyną w klasycznym programie bez żadnego różniczkowania, zaś e_1 jest to wartość jego pochodnej. Pochodnej względem czego? Względem pewnej szczególnej „zmiennej różniczkowania", która jest wyróżnionym obiektem w naszym programie komputerowym, ale nie jest żadnym symbolem! Oto więc nasze nowe „przepisy kuchenne" dla rozszerzonego programu numerycznego, dość oczywiste:

1. Wszystkie jawne stałe c, np. 2, czy π, reprezentowane są przez pary $D(c,0)$. Pochodna stałej się zeruje i to wszystko.
2. Wyróżniona zmienna różniczkowania o wartości x ma postać $D(x,1)$. Powtarzamy: jej wartość liczbowa *x* może być dowolna i jest to po prostu wyrażenie, niemające więcej sensu „symbolicznego", niż jakiekolwiek inne. Ale jej pochodna jest równa zawsze jedności, to oczywiste.
3. Arytmetyka nowych wyrażeń jest zgodna z intuicją. Dla dodawania mamy:
$$D(a,b) + D(c,d) = D(a+c, b+d).$$
4. Mnożenie jest zgodne z regułą Leibniza:
$$D(a,b) \cdot D(c,d) = D(a \cdot c, a \cdot d + b \cdot c).$$

Jak widzieliśmy, to wystarczy, aby zdefiniować dzielenie. *Musi* ono spełniać równość $D(a,b)/D(c,d) = D(a/c, (bc-ad)/c^2)$.

3. Funkcje elementarne

Wyrażenia arytmetyczne w programach zawierają także logarytmy, funkcje trygonometryczne, itp. Ale jak obliczyć np. $\sin D(a,b)$, czy $\sqrt{D(a,b)}$? Teoretycznie wszystko można sprowadzić w przybliżeniu do obliczeń wielomianowych i wymiernych, ale nam zależy na dokładności.

Uzyskamy duże uproszczenie rozumowania zauważając, że nasze pary są równoważne tzw. liczbom dualnym wymyślonym przez Clifforda. Zamiast $D(a,b)$ zapisujemy je jako $z = a + \epsilon b$, gdzie ϵ jest specjalną, algebraicznie niezależną jednostką, analogiczną do jednostki urojonej dla liczb zespolonych, tylko *nilpotentną*: $\epsilon^2 = 0$. (Fizycy czasami nazywają żargonowo składnik b „fermionowym", ze względu na to, że kwadrat fermionowej funkcji falowej w mechanice kwantowej jest równy zeru).

Przy użyciu tej notacji (i tej interpretacji) odtworzenie reguł dodawania i mnożenia dla par różniczkowych jest natychmiastowe. Dla dzielenia wystarczy zauważyć, że $(a+\epsilon b)\cdot(a-\epsilon b)=a^2$, co pozwala uprościć mianownik: $\frac{a+\epsilon b}{c+\epsilon d}=(a+\epsilon b)\cdot\frac{c-\epsilon d}{c^2}=\frac{a}{c}+\epsilon\frac{bc-ad}{c^2}$. Wszyscy znający rozwijanie funkcji w szeregi potęgowe zauważą, że

$$e^{a+\epsilon b}=e^a\cdot e^{\epsilon b}=e^a\cdot(1+\epsilon b), \tag{3.1}$$

bo wyższe potęgi ϵ znikną. Jest to zgodne ze wzorem na różniczkowanie funkcji wykładniczej: $\left(e^f\right)'=e^f\cdot f'$. W naszej notacji będziemy mieli $\exp D(a,b)=D(\exp a,\mathrm{b}\cdot\exp a)$. Wzory na pierwiastek i logarytm uzyskamy przez odwracanie kwadratu i funkcji wykładniczej, a funkcje trygonometryczne przez wzory na sumę i rozwinięcie ich w szereg z dokładnością do członów pierwszego stopnia: $\sin\epsilon b=\epsilon b$, i $\cos\epsilon b=1$. Oto niekompletna lista kilku funkcji elementarnych, rozszerzonych do naszej algebry różniczkowej:

$$\begin{aligned}
&\exp D(a,b)=D(\exp a,b\cdot\exp a),\\
&\ln D(a,b)=D(\ln a,b/a),\\
&\sin D(a,b)=D(\sin a,\mathrm{b}\cdot\cos a),\\
&\cos D(a,b)=D(\cos a,-b\cdot\sin a),\\
&\arctan D(a,b)=D(\arctan a,\tfrac{b}{1+a^2}),\\
&\sqrt{D(a,b)}=D(\sqrt{a},\tfrac{b}{2\sqrt{a}}).
\end{aligned} \tag{3.2}$$

Do tego możemy dopisać reguły obliczania innych funkcji cyklometrycznych, funkcji hiperbolicznych, wyrażeń $D(a,b)^{D(c,d)}$, a jeśli ktoś potrzebuje (i zna) funkcje specjalne potrzebne w fizyce, np. funkcje Bessela, także może je dołączyć do powyższych reguł. Jeśli napiszemy mały pakiet pozwalający na wykonywanie standardowych operacji arytmetycznych na złożonych strukturach danych – parach, nasz program będzie obliczał jednocześnie „wartości główne" oraz pochodne wyrażeń, które dostaniemy „automatycznie", nie musimy ich specjalnie obliczać. Operator derywacji ∂ jest po prostu selektorem drugiego składnika w strukturze $D(a,b)$.

4. Konstrukcja programu do różniczkowania

Określenie „program do różniczkowania" jest więc cokolwiek nieścisłe. Jedyne, co należy zaimplementować, to konstrukcja złożonych struktur danych – par, oraz wyposażenie tych par w operacje arytmetyczne przedstawione w poprzednim rozdziale. Program oblicza równocześnie wartości „główne" wyrażeń arytmetycznych i ich pochodne. Najwygodniej jest użyć w tym celu tzw. języka

obiektowego, pozwalającego na definiowanie dowolnych złożonych danych, o dowolnych własnościach.

Nie możemy zająć się tutaj nauczaniem programowania w Pythonie. Oto w olbrzymim skrócie struktura naszego dydaktycznego pakietu do różniczkowania automatycznego, który jest dostępny przez internet (http):
users.info.unicaen.fr/~karczma/Foton/Progs/autodif.py

Nie ma on nadmiernych ambicji, i nie jest kompletny, ale jest w pełni użyteczny i krótki (poniżej 100 krótkich wierszy), a więc czytelny. W internecie Czytelnik znajdzie i inne programy, np. moduły popularnego pakietu Scientific Python. Polecamy również stronice **http://www.autodiff.org/**.

Nasz program definiuje ***klasę*** obiektów **D(a,b)**, gdzie a, b są dowolne, oraz działania arytmetyczne na nich, a także kilka funkcji elementarnych zgodnie z dyskusją powyżej. Dostęp do składników par jest następujący, jeśli **z=D(x,y)**, **x** otrzymamy jako **z.e**, a **y**, jako **z.d**.

Pakiet dopuszcza arytmetykę „mieszaną", np. **3+D(2.4,3.9)*7** i wprowadza kilka użytecznych skrótów, np. stałe i zmienną: **def cst(x): return D(x,0.0)** oraz **def var(x): return D(x,1.0)**.

Definicje funkcji elementarnych są zgodne z (3.2), ale np. sinus hiperboliczny nie wymaga już żadnych specjalnych elementów, wystarczy napisać **def sinh(x): return (exp(x)-exp(-x))/2**. Ta definicja jest uniwersalna. Jeśli zażądamy wartości **sinh(1)**, otrzymamy ok. 1.1752012, a **sinh(var(1))** dostarczy **D(1.1752012, 1.5430806)**. Python jest językiem o typach dynamicznych, tj. procedury mogą wykonywać różne operacje w zależności od typów argumentów, a fukcję wykładniczą zdefiniowaliśmy tak, aby w razie gdy argument jest zwykłą liczbą, dostarczała także wartości liczbowej. Zwykłe funkcje arytmetyczne, które należą do modułu standardowego **math**, są nadal zawsze dostępne, ale mają nazwy prefiksowane przez literę **m**, np. **m.exp(x)**.

Aby zrobić wykres funkcji i jej pochodnej możemy użyć jednego ze standardowych pakietów graficznych, np. bardzo dobrej biblioteki graficznej Matplotlib. Jeśli **t** zawiera np. listę argumentów **x**, np. od zera do 50, wykres funkcji **x*cos(x)** wraz z wykresem pochodnej, otrzymamy poprzez instrukcje programu poniżej. Jest on kompletny, tylko wymaga instalacji odpowiednich pakietów.

```
from pylab import *
from autodif import *
t=linspace(0,50,600)
y=[var(x)*cos(var(x)) for x in t]
y0=[v.e for v in y]; y1=[v.d for v in y]
plot(t,y0,t,y1); show()
```

5. Wyższe pochodne

Wypada zastanowić się jak rozszerzyć formalizm na wyższe pochodne, również bardzo użyteczne. Do obliczania krzywizn w geometrii potrzebujemy drugich pochodnych, rozwinięcia w szereg Taylora wymagają wyższych pochodnych. Są one też użyteczne w technikach rozwiązywania równań, w optymalizacji itp.

Niestety, nasza algebra wyrażeń arytmetycznych rozszerzona o derywację nie jest „zamknięta", tj. operator ∂ działający na wyrażenie $D(a,b)$ nie dostarcza wyrażenia podobnego typu, tylko zwykłą liczbę. Nie da się tego dalej różniczkować. Tymczasem wyrażenia symboliczne po zróżniczkowaniu są nadal wyrażeniami symbolicznymi i różniczkowanie można wykonywać dowolnie wiele razy.

Jeśli potrzebujemy tylko drugich pochodnych, lub z rzadka trzecich, nasz formalizm (i nasz pakiet w Pythonie) jest wystarczający, choć niezbyt wygodny. Jeśli zdefiniujemy `p=var(var(1.0))`, wartość `p**3` wyniesie `D(D(1.0, 3.0), D(3.0, 6.0))`. Ostatni człon jest drugą pochodną. Pierwsza występuje podwójnie, w członach pośrednich. Funkcje elementarne także „działają" prawidłowo, np. `exp(p)` daje `D(D(2.718282, 2.718282), D(2.718282, 2.718282))`. Jest to możliwe gdyż nasze funkcje są rekursywne, wywołują same siebie dla składników par i korzystają z funkcji standardowych tylko gdy typ argumentu jest już czysto liczbowy. Widać jednak, że obliczanie w ten sposób np. dziesiątej pochodnej jest dość niewygodne, a wyniki są mało czytelne. Na skutek powielania członów pośrednich, wynik może zająć grubo ponad stronę tekstu.

Oczywiście można to nieco zracjonalizować. Czytelnik zechce sam popracować nad formalizmem, który rozszerza algebrę Clifforda, z wyrażeniami typu $z = a + \epsilon b + \eta c$, gdzie $\eta = \epsilon^2$, a $\epsilon^3 = \eta^2 = \epsilon\eta = 0$. Pozwoli to obliczać wszystko z dokładnością do drugiej pochodnej, pracując na wyrażeniach klasy $D(a,b,c)$. Ale ta algebra nadal nie jest zamknięta, otrzymanie wyższych pochodnych cierpi na ten sam problem co poprzednio.

Gdyby było możliwe operowanie listami o nieskończonej długości $D(e_0, e_1, e_2, \dots, e_n, \dots)$, teoretycznie problem byłby rozwiązany. Wyrażenie będące elementem algebry różniczkowej, na którym można dokonywać działań arytmetycznych, obliczać funkcje, a także różniczkować, należałoby do zamkniętej dziedziny. Stałe to wyrażenia $D(x, 0,0, \dots)$, zmienna, to: $D(x, 1,0,0, \dots)$, a derywacja wyrażenia $D(e_0, e_1, e_2, \dots, e_n, \dots)$ dostarcza $D(e_1, e_2, \dots, e_n, \dots)$. To jest również nieskończona lista, a więc wyrażenie tego samego typu co oryginalne, algebra się zamyka.

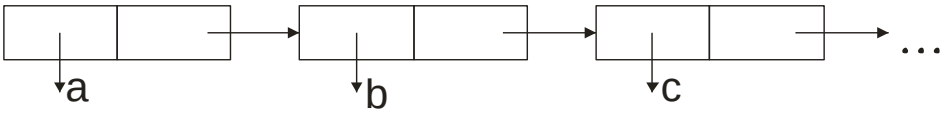
Czytelnik przyzwyczajony do „klasycznego" programowania złapie się tu za głowę... Po pierwsze, w skończonym komputerze nie da się umieścić nieskończonych list, trzeba je jakoś urwać. Po drugie, operacje matematyczne na tych urwanych, ale długich listach chyba będą dość skomplikowane, nieczytelne,

trudne do implementacji, i będą zawierały sporo elementów „administracyjnych" kodu, kontrolujących, aby urywanie wszystkich operacji odbywało się na tym samym poziomie. Wiadomo, że programy do rachunków perturbacyjnych są bardzo nieprzyjemne. Jednym słowem, może już lepiej użyć pakietów symbolicznych, mimo tego, że dziesiąta pochodna w miarę skomplikowanego wyrażenia może „rozsadzić" komputer wzorem, z którego nie ma pożytku, a potrzebna jest jedna liczba?...

Na szczęście nie jest tak. Technika, którą przedstawimy poniżej, nadaje się bardziej do języków tzw. „leniwych", takich jak Haskell czy Clean, ale i w Pythonie można ją zaimplementować, choć jest to na tyle skomplikowane, że omówimy tylko ogólne zasady, które można zawrzeć na jednej–dwóch stronicach tekstu. Najistotniejsze jest to, że w strukturach danych w programie, w parach, listach itp., **elementami nie muszą być liczby, ale także funkcje**: możemy pobrać wartość pewnego elementu danych i *zastosować ją* do jakiejś wartości liczbowej (lub innej). (Nie jest to żadna nowość ani ciekawostka, tej techniki używa się od zarania komputerów, tylko przez wiele lat proste języki programowania ukrywały to przed początkującymi...). Oto sposób na generowanie i operowanie potencjalnie nieskończonymi ciągami danych.

5.1. Programowanie „leniwych list"

Ten rozdział jest poświęcony pewnym elementom programowania list lub ciągów. Jeśli nasz język programowania umożliwia operowanie parami (a,b) złożonymi z dowolnych elementów, wystarczy to do złożenia ciągów o dowolnej długości, ciąg $[a,b,c,d]$ może zostać zrealizowany jako struktura $(a,(b,(c,(d,\mathrm{NIC}))))$, gdzie NIC jest wyróżnionym obiektem – znacznikiem, który służy tylko do sygnalizacji, że lista się kończy. Geometrycznie można to sobie wyobrazić na rysunku poniżej.



Rys. 1. Listy złożone z par

Takie listy możemy tworzyć w prawie wszystkich używanych aktualnie językach programowania. Wyobraźmy sobie, że w naszym programie dysponujemy parami $L(a,p)$, gdzie a jest liczbą, ale p jest „demonem" – obiektem-procedurą, której wywołanie dostarczy dopiero jakiejś wartości. Jakiej? Sztuka polega na tym, że to wywołanie dostarczy innej pary, $L(b,q)$, gdzie b jest następnym elementem ciągu, a q – następnym demonem, którego wywołanie da nam kolejną wartość w ciągu, itd. Możemy to powtarzać dowolnie wiele razy, dziesięć, albo milion. To stanowi naszą „nieskończoną" listę danych. Nie cho-

dzi o to, że jest ona fizycznie nieskończona, jest to oczywiście nonsensem, ale o to, że w ogóle *nie musimy myśleć o tym, gdzie ona się kończy*. Jeśli potrzebujemy 3 elementy, generujemy 3, jeśli milion – milion i ani jednego więcej. Program optymalizuje wywołania, jeśli odwołaliśmy się do elementu p w pierwszej parze, zostanie on fizycznie zastąpiony przez parę $L(b,q)$ i następnym razem już dostaniemy b bez potrzeby wywołania procedury p. Czytelnik zechce zrozumieć poniższą definicję „stałej" w programie:

```
def lcst(c):
    def zr():return L(0.0,zr)
    return L(c,zr)
```

Ta funkcja, analogiczna do poprzednio wprowadzonej **`cst(c)`**, która dostarczała $D(c,0)$, zawiera wewnętrzną funkcję **`zr()`** (bezparametrową), której wywołanie tworzy parę $L(0,\mathrm{zr})$. Odniesienie się do drugiego elementu tej pary, tj. do „ogona" listy, zamienia odnośnik do tej funkcji na następną parę o identycznej wartości. W ten sposób mamy listę o potencjalnie nieskończonej liczbie zer, które są generowane w miarę potrzeby.

Aby otrzymać „zmienną różniczkowania" x, tworzymy po prostu obiekt **`L(x,lcst(1.0))`**, co generuje listę $[x,1,0,0,\ldots]$.

5.2. „Leniwa", rekurencyjna arytmetyka

Nasze przepisy dotyczące operacji arytmetycznych nie są o wiele dłuższe od poprzednich, zawierają jednak pewien element programowania, który typowo nie jest uczony na studiach fizyki, mianowicie *odroczoną rekursję*. Zauważmy, że w definicji funkcji **`zr`** odnosi się ona do samej siebie, ale się nie wywołuje. Wywołanie nastąpi dopiero na wyraźne życzenie, w dowolnym późniejszym momencie. W podobny sposób konstruujemy operatory arytmetyczne. Na przykład, aby dodać do siebie dwie leniwe pary **`p`** i **`q`**, o składnikach **`p.e`** i **`p.d`** itp., możemy zaprogramować

```
def add(p,q):
    def der():
        return add(p.d(),q.d())
    return L(p.e+q.e,der)
```

i to praktycznie wszystko. Nasz program, dostępny w tym samym folderze, co poprzedni i noszący nazwę **`lautodif.py`**, jest ustrukturowany nieco inaczej, operator dodawania nie nazywa się **`add`**, i jest tzw. metodą w klasie par reprezentujących ciągi. Wywołanie **`p.d()`** nie tylko oblicza odroczoną wartość „ogona" ciągu, ale zastępuje w składnik d wewnątrz p przez obliczoną wartość. Uwzględniliśmy możliwość arytmetyki mieszanej, kombinującej liczby (traktowane jako stałe) i pary. Są to jednak szczegóły techniczne, ważne tylko gdyby ktoś chciał zrozumieć i np. rozszerzyć nasz pakiet.

Mnożenie kryje w sobie nowy element rekurencyjny. W odróżnieniu od poprzedniego modelu, teraz pochodna to jest znowu para, której pierwszy element dostarcza liczby, o którą ostatecznie chodzi. Procedura mnożąca będzie miała zgrubsza strukturę poniższą:

```
def mul(p,q):
    e=p.e*q.e
    def fl():
        return mul(p,q.d()) + mul(q,p.d())
    return L(e,fl)
```

(Zastąpiliśmy **add** przez operator dodawania. W naszym pakiecie także mnożenie jest identyfikowane przez operator * wyglądający nieco inaczej niż powyższa procedura **mul**). Pozostawiamy Czytelnikowi skonstruowanie zbliżonej (prostszej) procedury, która leniwie mnoży ciąg przez stałą liczbową. Pomińmy odejmowanie i dzielenie, które definuje się w sposób analogiczny i zakończmy rozdział przedstawieniem kilku funkcji elementarnych na ciągach.

Logarytm jest bardzo prosty, jeśli $p = L(e, d)$, otrzymamy algorytm $\ln p = L(\ln e, \lambda: d(\,)/p)$, gdzie forma ($\lambda$: wyrażenie) jest bezparametrową funkcją obliczającą odroczone wyrażenie. W Pythonie można to zaprogramować jako anonimową funkcję-formę **lambda**, albo przy pomocy wewnętrznych funkcji **fl**, jak w przykładach powyżej.

Aby obliczyć funkcję wykładniczą, winniśmy przejść do następnej klasy w naszej szkółce programowania leniwego. Przypominamy, że $(\exp e)' = e' \cdot \exp e$. Można to zaprogramować bezpośrednio, tylko po co wielokrotnie w ciągu pochodnych wielokrotnie obliczać $\exp e$? Podobnie jest z pierwiastkiem: $\left(\sqrt{e}\right)' = e'/2\sqrt{e}$. Ale normalne (tj. nie leniwe, takie jak Haskell czy Clean) języki programowania nie dopuszczają konstrukcji typu równań **p = f(p)**, gdzie **p** po obu stronach oznacza *to samo*. W Pythonie nasz program będzie miał następującą (z grubsza) strukturę:

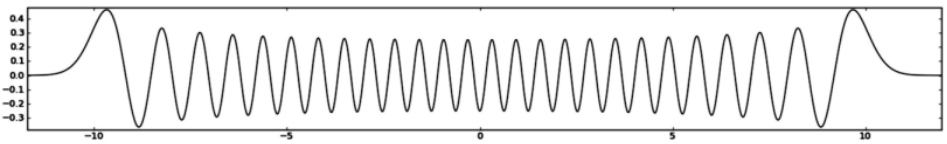
```
def exp(p):
    r=LL(exp(p.e),None) # None jest dowolnym wypełniaczem.
    def fl( ):return p.d( )*r
    r.d=fl      # Wymiana wypełniacza na właściwą wartość
    return r
```

Czytelnik jest w ten sposób przygotowany do skonstruowania samemu procedur na pierwiastek, arctan, czy funkcje trygonometryczne, a także na opracowanie własnych, specjalnych procedur różniczkowania funkcji charakteryzujących się specyficznymi własnościami. Jest to dość ważne dla optymalizacji wyższych pochodnych.

5.3. Uwaga metodyczna

Naszym celem nie jest zaoferowanie Czytelnikowi profesjonalnego, dobrze zoptymalizowanego pakietu do różniczkowania algorytmicznego, tylko przedstawienie ogólnych zasad. Nasz pakiet (w aktualnej wersji) zawiera nieoptymalności, które zresztą dotyczą również programów do różniczkowania symbolicznego: wykładniczy wzrost – „wybuch" – długości wyrażeń pośrednich przy obliczaniu wyższych pochodnych. Weźmy przykładowe wyrażenie `sin(x)*exp(-x*x)` i obliczmy ręcznie np. dziesiątą pochodną. Jest to dość okropne... Programy symboliczne generują bardzo długie formy strukturalne, ich upraszczanie trwa więc dość długo. U nas nie ma symboli, ale pamięć jest zajmowana przez pary zawierające „odroczone" obiekty funkcyjne, generowane dynamicznie podczas wykonywania programu. Ich przetwarzanie także trwa długo i może także zatkać dostępną pamięć, ale tak, czy inaczej, nasz program może być ekonomiczniejszy i pozwolić na obliczanie niektórych wyrażeń, z którymi np. Maple ma kłopoty. (Bywa jednak odwrotnie, o ile Maple jest w stanie dokonać daleko idących uproszczeń wyrażeń pośrednich). Nie jest to oczywiście krytyka pakietów symbolicznych, są one o wiele potężniejsze i służą innym celom. Zagadnienie optymalizacji takich rachunków jest ciągle przedmiotem prac naukowych i nie możemy mu poświęcić tutaj więcej miejsca.

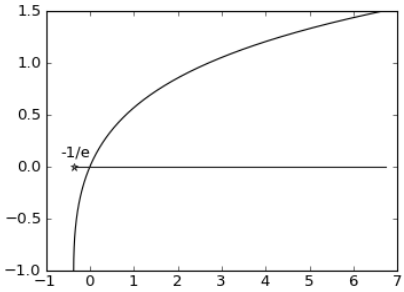
Czytelnik może jednak spróbować sam dokonać pewnych optymalizacji, np. nie ma sensu generowanie ciągu zer, skoro wiadomo, że tak się jawnie kończą pewne ciągi; można to zastąpić przez jedno zero liczbowe i odpowiednio zmodyfikować program. Można też prosto zoptymalizować różniczkowanie wielomianów, traktowanie potęgi x^n inaczej niż iterowane mnożenie pozwoli (czasami) uniknąć „wybuchu" reguły Leibniza przy obliczaniu wyższych pochodnych. To jest naprawdę ważny problem! Przypuśćmy, że chcemy obliczyć w pętli dwudziestą pochodną funkcji $e^{-x^2/2}$: może to posłużyć do obliczenia funkcji Hermite'a, niezwykle ważnych w mechanice kwantowej. Dziesiąta pochodna wymaga ułamka sekundy. Dwunasta – około 0.16 (na moim komputerze). Piętnasta: 1.7 sekundy, a dwudziesta ponad 80 sekund! Tzw. eksponencjalna złożoność obliczeniowa może być zabójcza. Jednak nie należy mylić trudności algorytmicznych z niestarannym kodowaniem! Ten wykres, dla funkcji Hermite'a pięćdziesiątego rzędu:



Rys. 1. Funkcja Hermite'a pięćdziesiątego rzędu

został wygenerowany w czasie pół sekundy, tymczasem trywialne skorzystanie z naszych procedur różniczkujących zajęło by czas przekraczający czas życia Wszechświata (szacujemy, że ok. 10^{21} sekund...) W przypadku procedur rekursywnych opłaca się czasami zapamiętywanie i powtórne wykorzystanie już raz obliczonych wartości. W naszych procedurach różniczkujących nie zawsze jest sensowne odraczanie wszystkiego, często jest rozsądniejsze obliczenie od razu czego się da, byle by nie rozwijać niepotrzebnie wyrażeń, których *aktualnie* nie potrzebujemy. Leniwe programowanie jest pewną sztuką!

Zakończymy ten rozdział zabawnym rozwinięciem w szereg pewnej funkcji użytecznej w wyrafinowanej fizyce teoretycznej, ale rzadko spotykanej na co dzień, funkcji Lamberta $W(x)$, która nie ma prostego wzoru analitycznego, natomiast znana jest funkcja do niej odwrotna: $x(W) = We^W$. Jak rozwinąć funkcję $W(x)$ w szereg potęgowy w zerze?



Rys. 3. Funkcja Lamberta

Pochodna funkcji odwrotnej wynosi $dx/dW = e^W(1 + W)$, czyli pochodna funkcji Lamberta spełnia równość $dW/dx = \exp(-W)/(1 + W)$. Ciąg pochodnych w zerze (znany analitycznie, n-ty człon jest równy $(-n)^{n-1}$) otrzymamy następującym programem:

```
def lambert( ):
    w=L(0.0,None)
    def fl():return exp(-w)/(1+w)
    w.d=fl
    return w
```

Program błyskawicznie wygeneruje ciąg liczb {0.0, 1.0, –2.0, 9.0, –64.0, 625.0, –7776.0, 117649.0, ...}

6. Przypadek wielu zmiennych (różniczkowanie w wielu wymiarach)

Uogólnienie przedstawionego formalizmu na wiele zmiennych jest łatwe do wyobrażenia. W dalszym ciągu opiszemy jedynie problem obliczania pierwszych pochodnych (gradientów), aczkolwiek fizykowi przydaje się bardzo znajomość Hesjanów – macierzy drugich pochodnych: $\frac{\partial^2 f}{\partial x_i \partial x_j}$, czy Laplasjanów – sum: $\frac{\partial^2 f}{\partial x_1^2} + \cdots + \frac{\partial^2 f}{\partial x_n^2}$. Musimy dysponować wieloma „zmiennymi różniczkowania", struktura $D(x, 1)$ nie wystarczy. Zagadnienie staje się wielowymiarowe, wektorowe.

Ten rozdział będzie schematyczny, podamy tylko główne zasady postępowania, bez szczegółów, które niewiele wnoszą do samego formalizmu. Rozszerzenie programu **`autodif.py`** na wektory nosi nazwę **`vautodif.py`** i mieści się w tym samym folderze co reszta. Ten program nie jest autonomiczny, wymaga popularnej biblioteki **`numpy`**, ułatwiającej operacje na tablicach i która zastąpi również standardowy moduł **`math`**. Praktycznie każdy fizyk, czy inżynier programujący w Pythonie instaluje sobie **`numpy`**, a jeśli Python jest instalowany centralnie w jakiejś placówce dydaktycznej, administratorzy systemu winni to zrobić od razu.

Klasa struktur danych opisujących wyrażenia nazywa się V. W tej klasie definiujemy pewną stałą n, która opisuje wymiar przestrzeni, tj. liczbę zmiennych niezależnych w sensie różniczkowania, np. 3. Tak, jak poprzednio tworzymy pary $V(a,b)$, z tym, że b jest wektorem – tablicą, n-elementową. Stała ma postać $V(c,[0,0,0\ldots])$, a k-ta zmienna: $V(x,[0,\ldots,0,1,0,\ldots])$, z jedynką w pozycji k-tej, dla k od zera do $n-1$.

Tutaj forma $[a,b,c]$ jest skrótem, kodowanie w Pythonie ma postać **`array([a,b,c])`**. Tablice w **`numpy`** są niezwykle wygodne w operowaniu. Jeśli A i B są tablicami, wystarczy napisać $A*B$, lub $A+B$, aby je pomnożyć lub dodać, element po elemencie. Wyrażenie $c*A$ daje wynik mnożenia tablicy przez stałą liczbową, a np. $\sin(A)$ dostarcza tablicy sinusów elementów.

Tak więc, o dziwo, dodawanie, odejmowanie, mnożenie i dzielenie form $V(a,b)$ ma *dokładnie tę samą postać* co w przypadku skalarnym! Funkcje arytmetyczne również zachowują się podobnie, np. $\sin V(e,[a,b,c]) = V(\sin e\,, \cos e\cdot[a,b,c])$. Tak więc, dostaliśmy pakiet wielowymiarowy praktycznie za darmo. Rozszerzenie go na wyższe pochodne także nie jest trudne, ale żmudne, zostawimy to już ambitniejszym Czytelnikom.

7. Wnioski końcowe

Przedstawione modele różniczkowania automatycznego nie są jedyne. Nie omówiliśmy bardzo frapującej techniki tzw. różniczkowania odwrotnego, gdzie fragmenty (zmodyfikowanego) programu numerycznego wykonuje się „do tyłu w czasie", od końcowych wyników, do zmiennych niezależnych. Nie poświęciliśmy uwagi technikom „półsymbolicznym", w których specjalny preprocesor modyfikuje tekst źródłowy oryginalnego programu numerycznego, tak, aby program liczył też dodatkowo pochodne; jest to w pewnym sensie równoważne użyciu pakietu symbolicznego, tylko podczas kompilacji programów w Fortranie itp.

Sytuacja, w jakiej znajduje się dziedzina różniczkowania algorytmicznego, jest zastanawiająca. Dysponujemy techniką użyteczną, efektywną i łatwą, opanowaną i rozwijaną od wielu lat, a tymczasem fizyków jej się albo w ogóle nie uczy, albo bardzo rzadko i powierzchownie. Są dziedziny, w których różnicz-

kowanie automatyczne zdobyło sobie już trwałe miejsce, np. w analizie efektywności i stabilności skomplikowanych obiektów przemysłowych (np. reaktorów jądrowych), a także w meteorologii i gdzieniegdzie w astronomii obliczeniowej, ale w typowej fizyce – wedle danych, którymi dysponujmy – jeszcze na ogół nie.

Jest to jedynie kwestia tradycji. Tzw. „skomputeryzowani fizycy" są raczej konserwatystami, programy w fizyce są dość proste, usiłuje się je pisać tak, aby maksymalnie wykorzystać istniejące biblioteki oprogramowania, a te, np. w Fortranie, nie są dostosowane do specyficznych, nietypowych struktur danych algebraicznych. Istnieją oczywiście odpowiednie programy w Fortranie (ADIFOR, TAPENADE, czy OpenAD/F) do różniczkowania automatycznego, ale wymagają one dobrego ich opanowania i nadają się bardziej do skomplikowanych obliczeń profesjonalnych, niż do dydaktyki.

Ponadto wśród fizyków pęd do obliczeń jak najszybszych jest wyraźny i dość zrozumiały, a użycie wyrafinowanych metod strukturalnych na ogół spowalnia obliczenia. Pewnym paradoksem w świecie fizyki i innych środowisk prowadzących intensywne obliczenia jest poważniejsze skoncentrowanie uwagi na szybkości samych programów, niż na stracie ludzkiego czasu potrzebnego do kodowania, optymalizacji, parametryzacji i zwłaszcza poprawiania programów. Miejmy nadzieję, że to się jednak zmienia, gdyż zubaża dydaktykę w tej dziedzinie...

Programy, które oferujemy Czytelnikom nie mają charakteru pakietów użytkowych, profesjonalnych (choć zostały one nieco zoptymalizowane w porównaniu do dyskusji w tekście). Mogą jednak posłużyć do dalszych eksperymentów w tej dziedzinie.
Zainteresowanych prosimy o kontakt (jerzy.karczmarczuk@unicaen.fr).