Jest to zapis odczytu wygłoszonego na seminarium *Popularyzacja Matematyki* na Uniwersytecie Warszawskim w roku akademickim 2004/2005.

Błądzić rzeczą nie tylko ludzką... czyli arytmetyka z komputera

Łukasz BODZON, Warszawa

Konstruktor Trurl zbudował raz ośmiopiętrową maszynę rozumną (...) złożył wszystkie narzędzia, rzucił fartuch na ziemię, wytarł twarz i ręce, i już, ot, tak, dla świętego spokoju, spytał:
– Ileż to jest: dwa a dwa?
– SIEDEM! – odparła maszyna.
Stanisław Lem, „Maszyna Trurla" ze zbioru „Cyberiada"

1. Udomowiony, ale narowisty

Żyjemy otoczeni komputerami. Większość naszego czasu upływa przy wtórze cichego szmeru, z jakim prąd przepływa przez układy elektroniczne. Właściwie maszyny liczące do tego stopnia wtopiły się w naszą codzienność, że prawie przestaliśmy już je zauważać – komputery, mniejsze lub większe, znajdują się w zegarkach, telefonach komórkowych, a nawet w niektórych pralkach i lodówkach. Żeby nie szukać zbyt daleko – nawet ten tekst powstawał z wykorzystaniem starego, poczciwego domowego „peceta".

Odnoszę wrażenie, że ludzkość pogodziła się już z tą dyskretną obecnością, a nawet trochę od niej uzależniła. Rzeczywiście, wykonanie pewnych czynności, które jeszcze dekadę wcześniej odbywało się w pełni „analogowo", dziś trudno sobie wyobrazić bez użycia nowoczesnych technologii (mówią: „Nie osiągniesz nigdy celu, gdy się nie znasz na Excelu!"). Wielu z nas stało się, z zamiłowania bądź z konieczności, ekspertami techniki komputerowej: wiemy, co to „megabajt", „piksel", „dysk twardy"; potrafimy zainstalować nową myszkę, wydrukować dokument oraz bezbłędnie odróżnić port szeregowy od równoległego. Ci, którzy mają duszę odkrywców odważyli się nawet zajrzeć do wnętrza piekielnej maszyny. Dzięki takim śmiałkom dowiedzieliśmy się, że w środku znajduje się plątanina kabli, jakieś układy drukowane, ogromna liczba małych i dużych mikroprocesorów oraz mnóstwo kurzu. To wszystko oczywiście w warstwie fizycznej, natomiast można by zapytać, co dzieje się na poziomie „mentalnym" (jeśliby uznać komputer za elektroniczny mózg, to takie określenie wydaje się uprawnione). Innymi słowy, chcielibyśmy wiedzieć, jakim cudem to wszystko działa.

Osoby bardziej wtajemniczone w arkana informatyki zdają sobie pewnie sprawę, że świat komputerów składa się z liczb – to liczby reprezentują znaki, które wyświetlane są na ekranie po naciśnięciu odpowiedniego klawisza klawiatury; również liczby modelują postać, którą sterujemy w grze komputerowej. Liczby. A gdzie są liczby, tam musi pojawić się arytmetyka, chcemy przecież dodawać, mnożyć, dzielić... Niestety, tu ujawnia się drobny dysonans w zgodnej, zdawałoby się, współpracy między człowiekiem a maszyną. Okazuje się mianowicie, że komputerowe rozumienie działania na liczbach, a w zasadzie samego pojęcia liczby, jest znacząco różne od ludzkiego. Mówiąc precyzyjniej, arytmetyka komputerowa jest niedoskonała, a jej mankamenty mogą prowadzić do nieoczekiwanych, często groźnych, rezultatów.

Aby zdać sobie sprawę z rangi problemu rozważmy pewien przykład. Można by go zatytułować...

... Sprzeczność w matematyce?!
Wyobraźmy sobie szereg harmoniczny

$$\sum_{n=1}^{\infty} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \dots .$$

Istnieje dowód, że suma odwrotności kolejnych liczb naturalnych (poczynając od jedynki) jest nieskończona – matematycy powiedzieliby, że szereg harmoniczny jest rozbieżny do plus nieskończoności.

Moglibyśmy wpaść na pomysł, aby potwierdzić ten fakt przy użyciu komputera. Nic prostszego, użyjemy w tym celu krótkiego programu:

$x_0 := 0$;
od $k = 1$ **do** N **wykonuj**:
$x_k := x_{k-1} + \frac{1}{k}$;

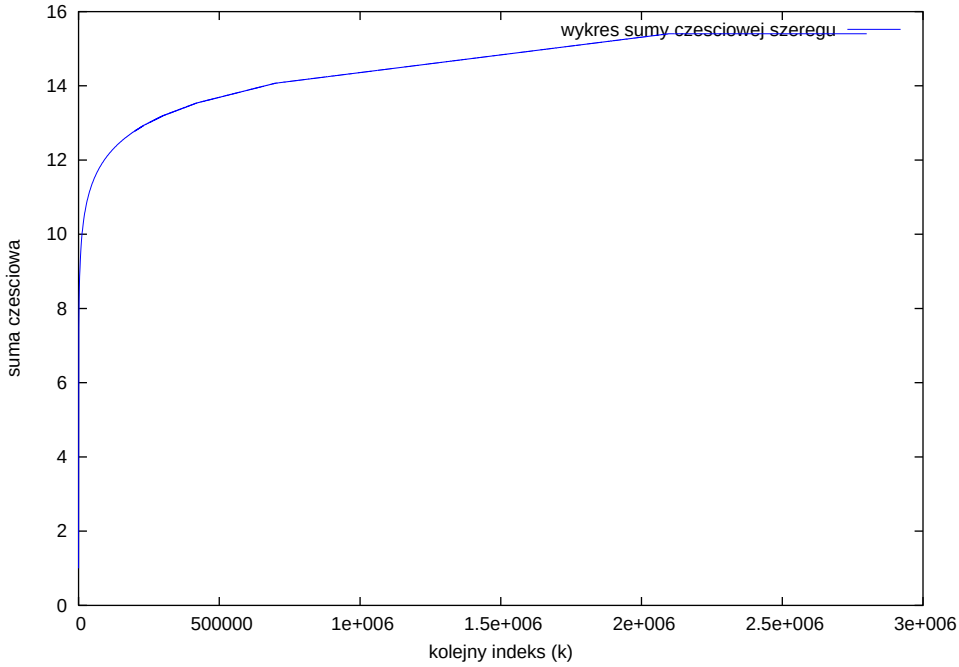
Po uruchomieniu powyższej procedury zmienna x_k będzie równa k-temu wyrazowi ciągu sum częściowych szeregu harmonicznego, czyli

$$x_k := \sum_{n=1}^{k} \frac{1}{n} = 1 + \frac{1}{2} + \ldots + \frac{1}{k}.$$

Zgodnie z tezą o rozbieżności tego szeregu, ciąg $\{x_k\}_{k=1}^{\infty}$ powinien być nieograniczony. Intuicja mówi nam, że jest to wręcz ciąg ściśle rosnący do nieskończoności, tzn.

$$x_k < x_{k+1} \qquad \text{dla każdej liczby naturalnej } k.$$

Jakież będzie zatem nasze zdziwienie, gdy odkryjemy, że wyliczone przy pomocy komputera liczby x_k od pewnego miejsca są sobie równe. Istotnie, dla wszystkich k, ℓ większych niż 2 200 000 okazuje się, że $x_k = x_\ell$, co można zobaczyć na poniższym rysunku.



Wypłaszczająca się linia wykresu wskazuje, że wygenerowany przez komputer ciąg sum częściowych szeregu harmonicznego jest od pewnego miejsca stały. Oznacza to, że szereg $\sum_{n=1}^{k} \frac{1}{n}$ jest zbieżny w arytmetyce komputerowej!

My jednak niezachwianie wierzymy, że w matematyce nie ma sprzeczności. Przedstawiony powyżej przykład powinien wzbudzić w nas więc pewną nieufność. I słusznie – na wynikach generowanych przez urządzenia liczące nie zawsze można bezpiecznie polegać. Postaramy się zbadać nieco głębiej przyczynę tego stanu rzeczy. W tym celu przyjrzymy się ograniczeniom arytmetyki komputerowej, spróbujemy określić związane z nimi zagrożenia, przekonamy się wreszcie, jak poważne mogą być konsekwencje błędów komputera.

2. Dogonić nieskończoność

Właściwie cały problem polega na tym, że w cyfrowej maszynie liczącej, która jest przecież skrzynką o skończonych rozmiarach, nijak nie da się zmieścić całej prostej rzeczywistej. Innymi słowy – komputery bardzo nie lubią nieskończoności, dlatego też pracują ze skończonymi reprezentacjami liczb. „Skończoność”, o której mowa powyżej, ma dwojaki sens: po pierwsze, wszystkich wielkości arytmetycznych możliwych do przechowywania w komputerze jest tylko skończenie wiele; po drugie, w arytmetyce komputerowej dokładnie reprezentowane są jedynie liczby o skończonym, i to nie przesadnie długim, rozwinięciu.

Skoro już jesteśmy przy temacie rozwinięcia, to w komputerach na ogół używa się bazy dwójkowej. Jest to dość naturalne rozwiązanie z punktu widzenia

urządzenia elektronicznego:

0 reprezentuje stan, gdy prąd nie płynie,
1 – gdy płynie.

W zamierzchłych czasach prowadzono wprawdzie eksperymenty z urządzeniami pracującymi w oparciu o system trójkowy, jednak ta koncepcja nie przyjęła się.

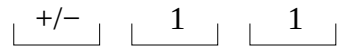
Dość już jednak tych dygresji, zobaczmy wreszcie jak wygląda reprezentacja liczb w arytmetyce komputerowej. W przypadku wielkości całkowitoliczbowych sytuacja jest stosunkowo nieskomplikowana – wystarczy umówić się co do tego, ile bitów zechcemy wykorzystać do przechowywania liczby. Gdybyśmy na przykład mogli wygospodarować trzy bity,

Bit będziemy rozumieć jako pewne miejsce, komórkę w pamięci, w które możemy wpisać 0 albo 1.

to jeden z nich przeznaczylibyśmy na znak, a dwa pozostałe – na reprezentację modułu liczby.

Znak reprezentujemy następująco
$0 \rightsquigarrow (-1)^0$ dając znak „+”,
$1 \rightsquigarrow (-1)^1$, co oznacza znak „−”.

Oczywiście szybko można się zorientować, że we wspomniane trzy bity żadną miarą nie wciśniemy wszystkich wartości całkowitych – najlepsze, co możemy osiągnąć to



Oznacza to, że zakres reprezentowalnych w ten sposób liczb zamyka się w zbiorze

$$\{-3, -2, -1, -0, +0, +1, +2, +3\}.$$

Pojawia się zatem problem **nadmiaru**: jeżeli liczba, którą zamierzamy wpisać do komputera, wyposażonego w naszą trzybitową reprezentację, będzie większa niż 3 lub mniejsza niż -3, to urządzenie zasygnalizuje błąd przekroczenia zakresu.

W rzeczywistych zastosowaniach do przechowywania liczb całkowitych używa się 16 lub 32 bitów, co oznacza, że dopuszczalny zakres to

$$-(2^{31} - 1), \ldots, 2^{31} - 1 \text{ dla liczb 32-bitowych.}$$

Tak, czy inaczej, problem nadmiaru wciąż pozostaje aktualny, choć naturalnie nie jesteśmy w jego obliczu całkiem bezradni. Istnieje kilka pomysłów, jak unikać przekroczenia zakresu.
Spróbujmy zacząć od zastanowienia się, czy, w konkretnym zastosowaniu, potrzebujemy wielkości różnych znaków. Gdyby wystarczyły nam tylko liczby dodatnie (lub tylko ujemne), to możemy zrezygnować z przechowywania znaku. Wówczas zakres wzrośnie dwukrotnie.
Jeżeli to nie przynosi rezultatu, przyjrzyjmy się algorytmowi realizowanemu przez nasz program – czasem da się tak go zmodyfikować, by problem nadmiaru nie występował. Na przykład, zamiast obliczać średnią arytmetyczną jako

$$\frac{n_1 + n_2 + \ldots + n_m}{m}; \qquad n_i \in \mathbb{Z}; \quad i = 1, \ldots, m,$$

obliczmy

$$\frac{n_1}{m} + \frac{n_2}{m} + \ldots + \frac{n_m}{m}.$$

Taka procedura będzie działać wolniej, bo zamiast jednego wykonujemy m dzieleń, ale dzięki temu unikamy niebezpieczeństwa przekroczenia zakresu w wyniku sumowania, potencjalnie dużych, liczb $n_1, \ldots, n_m$.
Osoby bardziej zdeterminowane, a przy tym biegłe w programowaniu, mogą zaimplementować własny system obsługi dużych liczb całkowitych.
Wreszcie – racjonalnym wyjściem jest, by przechowywać wartości całkowitoliczbowe jako liczby rzeczywiste, których zakres jest na ogół większy... ale jak reprezentować liczby rzeczywiste?...

W 1985 roku organizacja IEEE (*Institute of Electrical and Electronic Engineers*) opracowała standard określający sposób zapisu liczb rzeczywistych w pamięci komputera. Oparta na tych wytycznych arytmetyka nosi nazwę zmiennoprzecinkowej lub, krócej, arytmetyki *fl* (od angielskiego *floating point*

arithmetic). Zgodnie ze wspomnianym standardem, liczby rzeczywiste pojedynczej precyzji przechowywane są przy użyciu 32 bitów. Oznacza to, że maszyny cyfrowe odróżniają co najwyżej 2^{32} liczb rzeczywistych – zauważmy, że choćby na odcinku $[0,1]$ jest „trochę" więcej, bo continuum, punktów!

Wróćmy jednak do reprezentacji. Jest ona podobna do znanej nam notacji naukowej, ale w bazie dwójkowej, a nie dziesiętnej. Wartość rzeczywista jest więc tutaj postaci

$$(-1)^s \cdot m \cdot 2^{e-127},$$

przy czym

- s – decyduje o znaku (0 odpowiada znakowi „+", 1 – znakowi „-". Jak widać, w tym wypadku wystarczy jeden bit. A więc dobra nasza – przynajmniej znak jest reprezentowany dokładnie!);
- e – wykładnik przechowywany w 8 bitach, a zatem maksymalnie wynoszący $2^8 - 1 = 255$. Odejmujemy 127 po to, aby móc otrzymać wykładniki ujemne;
- m – 23-bitowa mantysa znormalizowana tak, by zawierała się między $\frac{1}{2}$ a 1. Da się to osiągnąć manipulując wykładnikiem: istotnie
  - gdyby $m = \frac{1}{3}$, to mnożąc przez 2 dostajemy $\frac{2}{3} \in [\frac{1}{2}, 1]$; wówczas wykładnik maleje o 1,
  - jeżeli $m = 1\frac{1}{2} = \frac{3}{2}$, to dzielenie przez 2 daje nam $\frac{3}{4} \in [\frac{1}{2}, 1]$; wtedy wykładnik rośnie o 1.

  Sprytnie zauważono, że, ponieważ mantysa jest odpowiednio znormalizowana, to ma postać

  $$m = \frac{1}{2} + \ldots,$$

  toteż w jej zapisie dwójkowym na pierwszym miejscu po przecinku zawsze stoi 1. Skoro jednak wiadomo, że rzeczona jedynka (zwana „ukrytym bitem") zawsze tam jest, to nie ma potrzeby jej przechowywać. Dzięki temu, że „ukryty bit" nie jest uwzględniany w reprezentacji, udało się dodatkowo zwiększyć dokładność.

Oto, jak wygląda przykład liczby rzeczywistej zapisanej w formacie arytmetyki *fl*.

Przykład: $x := 0,085$:

zakres bitów:	31	30-23	22-0
dwójkowo:	0	01111011	01011100001010001111011
dziesiętnie:	0	123	3019899
oznaczenia:	s	e	m

Zobaczmy, że liczba przechowywana w tej postaci nie jest idealnie równa $0,085$:

$$fl(x) := 2^{e-127}\left(1 + \frac{m}{2^{23}}\right) = 2^{-4}\left(1 + \frac{3019899}{8388608}\right) = \frac{11408507}{134217728} =$$
$$= 0,085000000984069671630859375.$$

Przekonujemy się naocznie, że komputer „widzi" odrobinę co innego niż to, co chcemy mu przekazać. Rzeczywiście, nie sposób ukryć, że model arytmetyki zmiennoprzecinkowej jest niedoskonały. Wpływa na to kilka czynników. Przede wszystkim, podobnie, jak w przypadku liczb całkowitych, mamy ograniczony zakres możliwych do przechowywania wartości – wykładnik, e, może być równy co najwyżej 128, a więc liczby zmiennoprzecinkowe nie wychodzą poza przedział $[-2^{128}, 2^{128}]$. Wiąże się to ze zjawiskiem nadmiaru, gdy wielkość, którą chcemy komputerowo przetwarzać, nie mieści się w zbiorze liczb reprezentowalnych. Jednocześnie wykładnik jest ograniczony również z dołu – najmniejszej co do modułu spośród niezerowych liczb, które da się uzyskać odpowiada $e = -127$, tzn. wszystkie wartości między 0 a 2^{-128} są w arytmetyce *fl* sklejane w jedną – mianowicie w zero! Prowadzi to do problemu **niedomiaru**, kiedy liczba, którą chcemy wczytać do komputera, jest zbyt mała, by mieć niezerową reprezentację. Może to mieć wiele groźnych następstw.

Sygnalizowaniu wystąpienia nadmiaru służą dwie dodatkowe liczby: $+\infty$ oraz $-\infty$, które uzyskuje się ustawiając wszystkie bity wykładnika na 1, zaś wszystkie bity mantysy na 0. Znak tych wartości ustala się w zależności od znaku liczby, dla której zakres został przekroczony.
Specyfikacja arytmetyki *fl* obejmuje jeszcze jedną specjalną liczbę, mianowicie NaN (od angielskiego *Not a Number*). Komputer używa tego wyrażenia w przypadku napotkania wartości nieoznaczonej, czyli np. $\frac{0}{0}$, $\frac{\infty}{\infty}$, $0 \cdot \infty$. Warto w tym miejscu wspomnieć, że w arytmetyce zmiennoprzecinkowej występuje zero ze znakiem. Całe szczęście $+0 = -0$, choć, co ciekawe, $\frac{1}{+0} = +\infty$, natomiast $\frac{1}{-0} = -\infty$. Jest to interesujący przykład wskazujący, że w świecie obliczeń numerycznych

$x = y \not\Rightarrow$
$\not\Rightarrow (f(x) = f(y)$ dla każdej funkcji $f)$.

Przykład: Wyobraźmy sobie dwie małe wartości x_1, x_2 takie, że każda z nich jest niezerowa w arytmetyce *fl*, ale już $x_1 \cdot x_2$ ma reprezentację równą zero. Gdybyśmy dla tych liczb zechcieli obliczyć $\frac{1}{x_1 \cdot x_2}$, to komputer zaprotestuje, twierdząc, że każemy mu dzielić przez zero.

Kolejną wadą rozpatrywanej przez nas arytmetyki jest fakt, że mantysa zawiera skończoną liczbę cyfr rozwinięcia. Nie możemy więc w sposób dokładny przechowywać w komputerze liczb, których zapis w systemie dwójkowym jest dłuższy niż 23 pozycje. Oczywiście tym bardziej nie możemy bezstratnie przetwarzać wartości o nieskończonym rozwinięciu binarnym.

Przykład: Pewnym zaskoczeniem może być to, że $\frac{1}{10}$, która w systemie dziesiętnym ma tak porządne rozwinięcie, nie da się w maszynie cyfrowej reprezentować dokładnie, bo jej zapis w bazie dwójkowej jest nieskończony

$$\frac{1}{10} = \frac{1}{16} + \frac{1}{32} + \frac{1}{256} + \frac{1}{512} + \frac{1}{4096} + \frac{1}{8192} + \ldots =_2 0,0001100110011001100110011\ldots$$

Jest to zagadnienie godne głębszej refleksji, zatem zatrzymajmy się na chwilę przy temacie dokładności reprezentacji liczb, które mieszczą się w zakresie, ale mają nieskończenie wiele cyfr w notacji dwójkowej. Przypuśćmy, że x jest jedną z takich wartości. Przedstawmy go w postaci

$$x = s \cdot 2^{e-127} \cdot m,$$

przy czym

$$m = \sum_{j=1}^{\infty} c_j 2^{-j}; \qquad c_j \in \{0,1\}; \quad c_1 = 1.$$

Jeżeli przechowujemy tylko t pierwszych cyfr mantysy, to zamiast m pamiętamy jej zaokrąglenie:

$$m_t := \sum_{j=1}^{t} c_j 2^{-j}.$$

W tej sytuacji x będzie w arytmetyce zmiennoprzecinkowej utożsamiane z liczbą

$$fl(x) := s \cdot 2^{e-127} \cdot m_t.$$

Względny błąd reprezentacji wynosi zatem

$$\nu_t := \frac{|x - fl(x)|}{|x|} = \frac{|m - m_t|}{|m|} \leqslant \frac{2^{-t}}{2^{-1}} = 2^{-(t-1)}.$$

Liczba ν_t charakteryzuje siłę arytmetyki – jeżeli mantysę przechowujemy w 23 bitach, to dokładność wynoszącą 2^{-22} można uznać za stosunkowo wysoką. Ścisłość nakazuje dodać, że w powyższych wyliczeniach nie uwzględniono faktu pomijania „ukrytego bitu" przy reprezentacji liczby. Gdybyśmy zechcieli wziąć to pod uwagę, wówczas stwierdzilibyśmy, że dokładność wzrasta do 2^{-23}.

3. Diabeł tkwi w szczegółach

Można by pomyśleć, że arytmetyka *fl*, mimo paru drobnych niedociągnięć, w gruncie rzeczy jest całkiem porządna – wprawdzie zakres reprezentowalnych liczb jest ograniczony, ale dość szeroki; także dokładność na poziomie 2^{-23} wydaje się na ogół satysfakcjonująca. To wszystko prawda, jednak okazuje się, że niewielkie błędy zaokrągleń mogą kumulować się podczas obliczeń, czasem rosnąc do całkiem znaczących rozmiarów.

Przykład: Przypuśćmy, że inwestujemy 1000 PLN na lokacie bankowej o oprocentowaniu 5% w skali roku, przy czym odsetki są naliczane codziennie. Ponieważ rok ma w przybliżeniu 360 dni, to stosując wzór

$$1000 \cdot \left(1 + \frac{0,05}{360}\right)^{360}$$

przekonujemy się, że po upływie dwunastu miesięcy należy nam się 1051,27 PLN.
Załóżmy jednak, że bank przechowuje wartość naszego kapitału w postaci liczby całkowitej wyrażonej w groszach. Każdego dnia stan naszego konta jest mnożony przez $(1 + \frac{0,05}{360})$ i sprowadzany do najbliższego grosza. Jeżeli zaokrąglenie następuje w górę, to po roku otrzymujemy 1053,42 PLN, co być może bardzo nas nie martwi, bo zyskujemy 2,15. Gorzej, gdy wartość lokaty zaokrąglana jest w dół – wówczas po upływie 360 dni stan konta wynosi 1049,78 PLN, co oznacza, że straciliśmy 1,49!

Powyższy przykład powinien uświadomić nam zagrożenie, jakie jest połączone z błędami nawarstwiającymi się w czasie wykonywania wielu operacji arytmetycznych na niedokładnych danych. Bez wątpienia zależałoby nam na możliwości sprawowania kontroli nad szybkością i sposobem rozprzestrzeniania się niedokładności. Inaczej mówiąc, chcielibyśmy mieć pewność, że początkowy, najczęściej mały, błąd danych wejściowych nie urośnie szaleńczo w czasie wykonywania programu. Tę świadomość zapewniają **zadania dobrze uwarunkowane**. Nazwiemy tak każde zagadnienie o tej własności, że niewielkie zaburzenie jego parametrów powoduje małe zaburzenie wyniku. Inaczej mówiąc, rozwiązania dobrze uwarunkowanego zadania dla bliskich sobie danych również są zbliżone. Można na to patrzeć, jak na coś w rodzaju ciągłości: niech R oznacza operator rozwiązania, D – przestrzeń danych, W – klasę wyników

$$R: \ D \to W$$
$$\forall\, \varepsilon > 0 \ \ \exists\, \delta > 0 \ \ \forall\, d_1, d_2 \in D \ \ \|d_1 - d_2\|_D < \delta \ \Rightarrow \ \|R(d_1) - R(d_2)\|_W < \varepsilon.$$

Niestety, nie wszystkie zagadnienia mają tak dobre własności. Na przykład odejmowanie jest w arytmetyce komputerowej zadaniem źle uwarunkowanym. Jeżeli α i β mają równe wszystkie cyfry rozwinięcia, poza kilkoma mniej znaczącymi, to ich różnica, $(\alpha - \beta)$, może mieć zaledwie kilka cyfr dokładności. Innymi słowy, odejmowanie bliskich sobie liczb jest obarczone dużym błędem.

Przykład: Rozważmy liczby (w programie komputerowym powinniśmy użyć liczb podwójnej precyzji)

$$x_1 = 10 + 4 \cdot 10^{-15} \qquad y_1 = 10 + 4 \cdot 10^{-14}$$
$$x_2 = 10 \qquad y_2 = 10.$$

Policzymy za pomocą komputera

$$z := \frac{y_1 - y_2}{x_1 - x_2}.$$

Jak łatwo stwierdzić, $z = 10$, natomiast komputer uważa, że... $z = 11,5$.

Okazuje się, że zadanie dobrze uwarunkowane to nie wszystko – aby otrzymać wiarygodny wynik należy rozwiązywać je algorytmem **numerycznie stabilnym**. Jest to pojęcie blisko spokrewnione z poprzednio omówionym terminem, tyle tylko, że dotyczy algorytmów. Chodzi mianowicie o takie procedury komputerowe, które dla bliskich sobie danych generują zbliżone wyniki. A zatem numeryczna stabilność również określa, jak kumuluje się błąd podczas wykonywania algorytmu.
Duże niedokładności pojawiają się, gdy używamy stabilnego algorytmu do problemu źle uwarunkowanego, albo algorytmu numerycznie niestabilnego rozwiązując zadanie dobrze uwarunkowane. Jeżeli zagadnienie jest źle uwarunkowane, a stosowana do niego procedura numerycznie niestabilna, to zdarzyć może się niemal wszystko. Z tego punktu widzenia dział matematyki zwany analizą numeryczną to sztuka znajdowania stabilnych algorytmów dla zadań dobrze uwarunkowanych.

Przykład: Wiadomo, że szereg $\sum_{k=0}^{\infty} \frac{x^k}{k!}$ jest jednostajnie zbieżny do funkcji e^x. Stąd mógłby narodzić się pomysł, by wartość funkcji wykładniczej przybliżać sumą częściową tego szeregu:

$$e^x \approx \sum_{k=0}^{3} \frac{x^k}{k!}.$$

Jest to całkiem dobre rozwiązanie, ale tylko, gdy $x \geqslant 0$, ponieważ dla ujemnych wartości argumentu x nasz algorytm wymaga odejmowania bliskich sobie liczb, o którym to zadaniu wiemy, że jest źle uwarunkowane. Rzeczywiście, wykonawszy tę procedurę dla $x = -2$ otrzymujemy wynik ujemny, co powinno odrobinę nas niepokoić.
Wadę powyższego algorytmu da się na szczęście łatwo naprawić: wystarczy dla $x < 0$ policzyć $w := e^{-x}$, co można bez obaw zrobić korzystając z sumy częściowej, a następnie jako wynik, e^x, wziąć $\frac{1}{w}$.

Nie powinniśmy jednak być tendencyjni. Obiektywizm nakazuje powiedzieć, że błędy zaokrągleń, choć rzadko, czasem bywają pomocne.

Przykład: Metoda potęgowa służy do przybliżania unormowanego wektora własnego odpowiadającego największej wartości własnej macierzy A, nieosobliwej i dodatnio określonej. Idea tej metody jest bardzo prosta:

$$\text{dla wektora startowego } x_0 \qquad x_n := Ax_{n-1} = A^n x_0.$$

Okazuje się, że podczas sukcesywnego wymnażania przez macierz A najszybciej rośnie składowa w kierunku wektora odpowiadającego największej wartości własnej. Co więcej, jeżeli x_n zostanie w każdym kroku znormalizowane, to składowe w pozostałych kierunkach będą wygaszane. Jest jednak jeden warunek: nasz wektor początkowy, x_0, musi mieć niezerowy rzut na prostą wyznaczaną przez poszukiwany wektor własny. W przeciwnym wypadku (teoretycznie) metoda nie jest zbieżna. Okazuje się jednak, że w praktyce obliczeniowej nawet, jeśli x_0 ma zerową składową w kierunku wektora odpowiadającego największej wartości własnej, to po kilku iteracjach ta składowa robi się niezerowa dzięki błędom zaokrągleń. Powoduje to, że metoda potęgowa jest (praktycznie) zbieżna dla każdego x_0.

4. Zetknięcie dwóch rzeczywistości

Nie pozbawiony odrobiny słuszności byłby zarzut, że powyższe przykłady są raczej akademickie – kogo interesuje wartość e^{-2} albo różnica jakichś prawie równych liczb, szczególnie jeśli jest ona tak mała, że można ją bez zażenowania zastąpić zerem? Czy rzeczywiście trzeba dbać o to, czy zadanie jest dobrze uwarunkowane, a algorytm numerycznie poprawny? Najlepszą odpowiedź na te pytania daje praktyka obliczeniowa – pora więc na przykłady z życia.

Przenieśmy się odrobinę w czasie i przestrzeni, do położonej w Arabii Saudyjskiej miejscowości Dahran. Podczas Wojny w Zatoce Perskiej stacjonujące tam odziały Stanów Zjednoczonych wykorzystywały rakiety patriot do namierzania i zestrzeliwania wrogich pocisków, zanim zniszczą one ważne obiekty naziemne. 25 lutego 1991 systemy obronne zawiodły – iracka rakieta typu scud trafiła w baraki armii amerykańskiej zabijając 28 żołnierzy i raniąc dalsze 100 osób. Okazało się, że winę za tę katastrofę ponoszą błędy arytmetyki komputerowej. Otóż po uruchomieniu wewnętrzny zegar baterii rakiet patriot zaczynał odliczać czas mierzony w dziesiątych częściach sekundy, ale przechowywany jako liczba całkowita (np. 32, 33 dziesiętne sekundy od uruchomienia). Jednak do obliczenia przyspieszenia oraz położenia śledzonej rakiety potrzebny był czas wyrażony wielkością rzeczywistą. Zatem liczba całkowita podawana przez zegar systemowy była mnożona przez $\frac{1}{10}$ i przechowywana w 24-bitowym rejestrze już jako wartość rzeczywista. Pamiętamy jednak, że rozwinięcie dwójkowe $\frac{1}{10}$ jest nieskończone. Każdorazowe obcięcie go do 24 bitów (24 miejsc po przecinku w rozwinięciu dwójkowym) powodowało błąd równy ok. $0,000\,000\,095$ w zapisie dziesiętnym. Nie jest to dużo, jednak ten niewielki błąd nawarstwiając się przez np. 100 godzin działania urządzenia robił się już znaczący:

$$0,000\,000\,095 \cdot 100 \cdot 60 \cdot 60 \cdot 10 = 0,342.$$

Rakieta scud poruszająca się z prędkością 1676 m/s może pokonać w tym czasie ponad pół kilometra, co na ogół wystarczy do tego, by znalazła się poza zasięgiem baterii patriotów.

• • •

Zdjęcie uzyskane dzięki uprzejmości agencji ESA/CNES, do której należą prawa autorskie.

4 czerwca 1996 roku bezzałogowa (na szczęście!) rakieta Ariane 501, owoc 7 miliardowej inwestycji oraz dziesięcioletnich badań Europejskiej Agencji Kosmicznej, eksplodowała 40 sekund po starcie. Moment wybuchu został zarejestrowany na zamieszczonej obok fotografii. W tym wypadku komputer pokładowy poddał się po tym, jak nie powiodła się konwersja przyspieszenia poziomego rakiety z 64-bitowej liczby rzeczywistej na 16-bitową liczbę całkowitą ze znakiem. Stało się tak dlatego, że konwertowana wartość była większa niż 32767, czyli największa liczba możliwa do przechowywania w 16-bitowym rejestrze. Co ciekawe, błąd ten nie dotyczył poprzedniczki pechowej rakiety, Ariane 4, która po prostu osiągała mniejsze przyspieszenia poziome, a więc problem nadmiaru nie występował.

• • •

Stavanger, Norwegia, 23 sierpnia 1991 rok. Podczas operacji balastowania tonie w Morzu Północnym warta 700 milionów dolarów platforma wiertnicza firmy Statoil. Zatonięcie 57-tysiącotonowej instalacji wywołuje wstrząs o sile 3 stopni

w skali Richtera. Tym razem wypadek również wiązał się z komputerami, ale jego podłoże było bardziej skomplikowane, niż tylko brak precyzji arytmetyki. Otóż rozkład ciśnień działających na fragmenty konstrukcji modeluje się za pomocą równań różniczkowych cząstkowych. W tym przypadku zawiódł pakiet numeryczny przybliżający rozwiązania RRCz za pomocą tzw. „metody elementu skończonego". Nacisk na podwodne części platformy został niedoszacowany o ok. 47%, co spowodowało, że przy pewnej głębokości konstrukcja po prostu rozpadła się pod wpływem ciśnienia.

• • •

Więcej przykładów niepożądanych skutków błędów związanych z obliczeniami numerycznymi można znaleźć na stronie
http://www5.informatik.tu-muenchen.de/~huckle/bugse.html.

5. Zasada ograniczonego zaufania

Mam nadzieję, że, zapoznawszy się z faktami dotyczącymi arytmetyki zmiennoprzecinkowej, zaczniemy patrzeć na nasze, wydawałoby się całkiem oswojone, komputery odrobinę bardziej podejrzliwie. Ale, mówiąc poważnie – maszyny cyfrowe to bezsprzecznie bardzo wygodne i użyteczne narzędzia, powinniśmy jednak pamiętać, że, jak wszystkie narzędzia, mają one swoje ograniczenia i nie da się ich stosować równie skutecznie do każdego rodzaju problemu. Aby korzystanie z nich było bezpieczne i efektywne należy pamiętać o pewnych zasadach. Przede wszystkim, trzeba zastanowić się, czy zadanie, jakie mamy do rozwiązania jest dobrze postawione, tzn. czy istnieje jego rozwiązanie. Po drugie, w razie potrzeby powinniśmy przeformułować problem tak, by stał się dobrze uwarunkowany, tzn. by małe zaburzenie danych powodowało małą zmianę wyniku. Po trzecie, należy zadbać o to, żeby algorytm, który stosujemy, był numerycznie poprawny – będziemy wówczas w stanie kontrolować przyrost błędu. W szczególności unikajmy w naszej procedurze odejmowania liczb bliskich sobie. Powinniśmy również zdawać sobie sprawę z ograniczeń arytmetyki. Mówiąc bardziej konkretnie, musimy się upewnić, czy typ, którego używamy do przechowywania danych w pisanym przez nas programie ma wystarczająco duży zakres, by nie wystąpił nadmiar albo niedomiar. Miejmy, w końcu, wyobraźnię, aby móc przewidzieć szczególne przypadki i wyjątkowe sytuacje, które mogą się zdarzyć podczas realizowania algorytmu. A tak w ogóle, to, jeśli to tylko możliwe, stosujmy gotowe, sprawdzone biblioteki numeryczne – to na ogół rozwiązanie szybsze i bezpieczniejsze niż używanie własnoręcznie napisanych procedur.

Pamiętajmy wreszcie, że nie tylko niedokładność arytmetyki jest źródłem błędów. Jeżeli dane dla naszej procedury są na przykład wynikami pomiarów, to musimy mieć świadomość, że nie wszystko da się zmierzyć dokładnie. Często już na starcie mamy parametry obarczone błędami, a wówczas szczególnie ważna jest numeryczna stabilność algorytmu. Poza tym istotne bywają również błędy dyskretyzacji – komputer, jako stworzenie nie lubiące nieskończoności, źle radzi sobie z problemami ciągłymi. Aby w ogóle myśleć o przybliżeniu rozwiązania tego rodzaju zadania, trzeba je zastąpić zagadnieniem dyskretnym. Na przykład całkę $\int_0^1 f(x)dx$ przybliżamy kwadraturą $\sum_{i=1}^{n} a_i f(t_i)$ używającą skończenie wielu wartości funkcji f. Oczywiście mogą istnieć funkcje, które w punktach używanych przez algorytm mają te same wartości, ale np. w normie supremum znacząco się różnią. Tu tkwi potencjalne źródło niedokładności.

Na zakończenie pozostaje mi tylko życzyć owocnych i ekscytujących obliczeń.

Literatura

[1] *ANSI/IEEE Standard 754-1985. Standard for binary floating point arithmetic.* Technical report, Institute of Electrical and Electronics Engineers, 1985.

[2] David Goldberg, *What every compute scientist should know about floating-point arithmetic*, ACM Computing Surveys, 23(1) (March 1991), 5-48.

[3] http://www.cs.princeton.edu/introcs/91float/

[4] http://www.resonancepub.com/oops.htm