Metody numeryczne I

Janusz Szwabiński

Redakcja techniczna, opracowanie tekstu i skład: Janusz Szwabiński
Złożono za pomocą systemu LaTeX 2_ε i edytora Kile.

Spis treści

Przedmowa

Rozdział 1

Warsztat pracy

> Unix jest przyjazny dla użytkownika - jest tylko wybiórczy jeżeli chodzi o dobór przyjaciół.
>
> Anonim (za [1])

Obliczenia numeryczne, jak wszystkie inne formy działalności człowieka, wymagają odpowiedniego warsztatu pracy. Można wprawdzie uprawiać numerykę przy pomocy kartki i ołówka, jednak nie rozwiąże się w ten sposób wielu ciekawych problemów. Ale już komputer wyposażony w kilka odpowiednich narzędzi pozwoli wygodnie i wydajnie pracować.

1.1 System operacyjny

Na ostatniej, czerwcowej liście Top500 [2] pięciuset najszybszych superkomputerów świata 367 z nich to maszyny pod kontrolą różnych odmian Linuksa (patrz tabela 1.1). Na tej samej liście znajdziemy tylko dwa superkomputery pracujące pod kontrolą systemów z rodziny Windows, jeden na 130, drugi na 470 miejscu. Ta statystyka powinna wystarczyć jako kryterium wyboru systemu operacyjnego każdemu, kto planuje na poważnie zająć się obliczeniami numerycznymi o dużej wydajności. Linux nie tylko jest dużo tańszy (bo darmowy!) od swego konkurenta. Bije go na głowę pod względem stabilności, czasu nieprzerwanej pracy, wyboru specjalistycznego oprogramowania i dostępności technologii wspierających obliczenia równoległe [3]. Właśnie z tego powodu niniejszy skrypt oparty jest na Linuksie.

Nie oznacza to jednak, że każdy słuchacz kursu z metod numerycznych powinien od razu porzucić swój ulubiony system operacyjny i przejść na Linuksa. W ramach kursu stabilność i czas nieprzerwanej pracy nie będą odgrywać aż tak dużej roli, ponieważ projekty do wykonania dadzą się policzyć dość szybko nawet na bardziej wysłużonym sprzęcie. Ponadto jednym z kryteriów doboru oprogramowania była jego dostępność na wszystkich popularnych systemach operacyjnych, nie tylko pod Linuksem.

Rodzina systemów	Liczba
Linux	367 (73,4%)
Windows	2 (0,40%)
Unix	98 (19,60%)
BSD	4 (0,80%)
Mieszane	24 (4,80%)
Mac OS	5 (1,00%)

Tabela 1.1: Systemy operacyjne na liście Top500 z czerwca 2006 [2].

1.2 Języki oprogramowania

Najprostszym kryterium doboru języka programowania jest stopień jego opanowania - wybieramy po prostu ten, który najlepiej znamy, o ile oczywiście nadaje się do prowadzenia obliczeń numerycznych. Dla większości uczestników tego kursu będzie to najprawdopodobniej C++, bo właśnie tego języka uczyli się najdłużej w trakcie dotychczasowych zajęć w ramach studiów. W pierwszych latach istnienia C++ panowało uzasadnione przekonanie, że nie nadaje się on do wydajnych obliczeń. Jednak od tego czasu sporo się zmieniło, głównie dzięki lepszym kompilatorom i STL, i kod napisany w nowoczesnym C++ potrafi być równie wydajny, a czasami nawet szybszy od programów stworzonych w konkurencyjnych językach [4].

Jeżeli zdecydujemy się na C++, do dyspozycji będziemy mieli sporo darmowych narzędzi. Pod Linuksem kompilator tego języka wchodzi w skład zestawu kompilatorów gcc [5]. Wsparcie dla C++ oferuje większość popularnych edytorów, zarówno tych pracujących w trybie tekstowym, jak i graficznym. Możemy też skorzystać ze zintegrowanych środowisk graficznych - Kdevelopa [6] i Anjuty [7]. Pod Windowsem istnieje darmowe środowisko programistyczne Dev-C++ [8], które również korzysta z kompilatorów gcc.

Mówiąc o obliczeniach numerycznych prowadzonych na komputerze nie sposób nie wspomnieć o Fortranie, czyli języku, w którym dokonano najwięcej do tej pory obliczeń tego typu. Język ten powstał w latach 50-tych ubiegłego stulecia z myślą o zastosowaniach naukowych i inżynieryjnych. Między innymi dlatego już od pierwszych wersji oferował pełne wsparcie dla liczb zespolonych. Ze względu na szybkość obliczeń, porównywalną z asemblerem, Fortran był najchętniej używanym językiem wśród naukowców. Obecnie traci swoją pozycję na rzecz C++. Bo chociaż ciągle jeszcze wygrywa w większości zadań numerycznych (choć różnice nie są już tak duże, jak kilkanaście lat temu), to ustępuje mu znacznie pola pod względem uniwersalności, a co za tym idzie - nienaukowych zastosowań.

W zestawie kompilatorów gcc znajdziemy bardzo dobry kompilator Fortrana 77 i coraz lepszy Fortrana 90 [5]. Dodatkowo Intel (`http://www.intel.com`) oferuje linuksową wersję swojego kompilatora Fortrana 90 za darmo do celów niekomercyjnych. Pod Windowsem możemy korzystać z kompilatorów należących do gcc. Mamy również do dyspozycji darmowy kompilator Fortrana 77 udostępniony przez OpenWatcom [9].

Obecnie w projektach programistycznych można zaobserwować tendencję odchodzenia (przynajmniej częściowego) od tradycyjnych języków kompilowanych, takich jak C++ czy Fortran, na rzecz języków skryptowych (m.in. Perl,

Tcl, Python). Badania wskazują bowiem, że programiści stosujący te ostatnie są prawie dwukrotnie wydajniejsi od swoich kolegów kodujących w tradycyjnych językach [10]. Przyczyna jest stosunkowo prosta. Pojedynczym poleceniom w wysokopoziomowych językach skryptowych odpowiada często wiele linii kodu w kompilowanych językach programowania. A średnia liczba błędów, jaką popełnia programista w określonej liczbie linii kodu, praktycznie nie zależy od wyboru języka. Innymi słowy, pisząc krótszy kod popełniamy z reguły mniej błędów.

Oczywiście gdzieś za ten wzrost wydajności programisty trzeba zapłacić - program napisany w języku skryptowym wykonuje się najczęściej znacznie wolniej niż jego skompilowany odpowiednik. W większości zastosowań nienaukowych to spowolnienie nie ma większego znaczenia - statystycznie program i tak większość czasu spędza na oczekiwaniu na operacje wejścia/wyjścia i tym podobne rzeczy. Jednak na pierwszy rzut oka dyskwalifikuje ono języki skryptowe do zastosowania w obliczeniach naukowych. Rzeczywiście w wielu dużych projektach szybkość wykonania jest ciągle jeszcze najważniejszym kryterium wyboru języka. Ale również wśród naukowców daje się zaobserwować wzrost zainteresowania językami skryptowymi. Po pierwsze, nie każde zadanie obliczeniowe to od razu program wykonujący się miesiącami na wielkich superkomputerach. Istnieje wiele nietrywialnych zadań (np. aproksymacja danych doświadczalnych, poszukiwanie miejsc zerowych funkcji, różniczkowanie numeryczne, wyznaczniki macierzy), w których czas napisania programu jest dużo ważniejszy od czasu wykonania programu. Po drugie, języki skryptowe łatwo się integrują z językami kompilowanymi. Dlatego nawet w dużych projektach znajdują one zastosowanie, np. do tworzenia interfejsów użytkownika, podczas gdy cały silnik obliczeń działa w C++ bądź w Fortranie.

Języków skryptowych jest bardzo dużo. Jedne zostały stworzone do konkretnych zadań, inne stanowią pełnoprawne języki programowania o uniwersalnym przeznaczeniu. Wśród tych ostatnich jednym z bardziej interesujących jest Python [11], który może przydać się w codziennej pracy (nie tylko początkującym) numerykom.

W skrypcie tym do każdego rozdziału dołączyłem przykładowe programy w C++, Fortranie i Pythonie, a także GNU Octave, o którym mowa będzie w paragrafie 1.4. Wybór C++ powinien być jasny - to „naturalny" język naszego instytutu. Mamy wieloletnie doświadczenia i sukcesy w nauczaniu go i każdy student fizyki komputerowej zaczynając kurs z metod numerycznych powinien się nim w miarę sprawnie posługiwać.

Programiści C++ mogą potraktować Fortrana jako ciekawostkę. Gdyby jednak komuś „nie leżało" C++, bo akurat nie planuje kariery programisty i własne programy tworzy raczej sporadycznie, być może zainteresuje go właśnie ten język. Z moich doświadczeń wynika, że składnia Fortrana nie przeszkadza aż tak mocno niewprawionym programistom w rozwiązaniu problemu jak składnia C++ i łatwiej wraca się do tego języka po dłuższych przerwach w programowaniu. Równie ważne jest, że początkujący programista Fortrana jest w stanie bez większego wysiłku napisać wydajny program, podczas gdy z C++ bywa pod tym względem różnie.

Python powinien przydać się każdemu. To doskonałe narzędzie, aby szybko przeliczyć jakieś zadanie, które z jednej strony jest zbyt trudne, aby zrobić to samemu na kartce, a z drugiej - na tyle proste, aby wynik obliczeń na komputerze otrzymać prawie natychmiast (znalezienie wyznacznika macierzy 10×10 jest tu dobrym przykładem). Inny obszar zastosowań, w którym Python sprawdza

się bardzo dobrze, jednak jego omówienie wykracza poza ramy tego kursu, to łączenie ze sobą różnych aplikacji lub opakowywanie istniejących programów przyjaznymi interfejsami użytkownika.

1.3 Biblioteki numeryczne

Osoba skonfrontowana z praktycznym problemem do rozwiązania, o ile nie zajmuje się zawodowo analizą numeryczną, raczej nie będzie miała czasu i środków, aby samodzielnie rozwinąć i zaimplementować odpowiedni algorytm numeryczny. I z reguły nie będzie takiej potrzeby, bo z biegiem lat powstało wiele bibliotek numerycznych, które pozwalają korzystać z gotowych implementacji wielu algorytmów obliczeniowych.

Programistów C/C++ zainteresować powinna przede wszystkim darmowa biblioteka GSL [12], która oferuje ponad 1000 funkcji bibliotecznych z różnych dziedzin metod numerycznych (patrz dodatek C). Znajdziemy w niej m.in. implementacje prawie wszystkich algorytmów omówionych w tym skrypcie.

Pisząc programy w Fortranie, będziemy mogli skorzystać z LAPACKa [13] - bardzo dobrej biblioteki do algebry liniowej[1]. LAPACK stworzony został z myślą o procesorach wektorowych i równoległych, ale można stosować go z powodzeniem również na zwykłych komputerach osobistych. Gdyby interesowały nas inne darmowe biblioteki, możemy przeszukać zasoby Fortran Library [14]. Duży wybór procedur na każdą okazję znajdziemy również w „Numerical Recipes" [15].

Decydując się na Pythona, możemy skorzystać z bardzo dobrych modułów NumPy/SciPy [16] oraz ScientificPython [17]. Znajdziemy w nich implementacje do większości algorytmów omówionych w tekście. Warto również zainteresować się projektem PyGSL [18], który pozwala korzystać z biblioteki GSL z poziomu Pythona.

W tym miejscu można zadać pytanie o sens nauczania metod numerycznych ludzi, którzy zajmują się praktycznymi obliczeniami. Skoro większość algorytmów i tak jest już zaimplementowana, czy nie wystarczyłoby przedstawić listę odpowiednich funkcji bibliotecznych w wybranym języku? Otóż nie. Stosowanie bibliotek numerycznych nie zwalnia użytkownika od uważnego przeanalizowania sposobów rozwiązania problemu. Znajomość mocnych i słabych stron metody, po implementację której sięgamy, pozwoli uniknąć wielu niespodzianek i lepiej zrozumieć uzyskany wynik.

1.4 Środowiska numeryczne

Biblioteki numeryczne znacznie upraszczają rozwiązywanie praktycznych problemów, jednak ciągle jeszcze wymagają pewnego wysiłku programistycznego od użytkownika. Aby wysiłek ten zredukować do niezbędnego minimum, stworzono kilka środowisk dedykowanych obliczeniom numerycznym. Wyposażone są one zwykle w narzędzia do analizy, manipulacji i wizualizacji danych oraz rozbudowane biblioteki numeryczne o uproszczonym interfejsie. Pozwalają na pracę

[1] Istnieje również CLAPACK, czyli wersja biblioteki przetłumaczona na język C za pomocą narzędzia `f2c`.

w trybie interaktywnym, interpretując polecenia wprowadzane przez użytkownika, oraz w trybie wsadowym dzięki skryptom tworzonym we wbudowanym i łatwym do opanowania języku programowania. Ten pierwszy tryb jest idealny do testowania nowych pomysłów i eksperymentów numerycznych, drugi przydaje się w większych projektach, gdy przebieg obliczeń nie wymaga interakcji z użytkownikiem i można go zautomatyzować.

Pierwsze środowiska do obliczeń numerycznych powstały dość dawno. Jednak dopiero w ostatniej dekadzie, dzięki dostępności naprawdę szybkich komputerów za stosunkowo niewielką cenę, stały się powszechnie stosowanym narzędziem przez naukowców i inżynierów. Niepisanym standardem wśród programów tego typu jest Matlab firmy MathWorks (`http://www.mathworks.com/`). Istnieje jednak kilka darmowych programów niewiele ustępujących mu możliwościami. Do najciekawszych należą GNU Octave [19], Scilab [20] i Rlab [21]. GNU Octave to chyba najlepszy darmowy klon Matlaba. Przy zachowaniu kilku zasad [22] użytkownik może przenosić większość swoich skryptów między tymi programami. Składnia Scilaba również wzorowana jest na Matlabie, jednak pełna kompatybilność nie była głównym priorytetem jego autorów. Program wyróżnia się za to przyjaznym, graficznym interfejsem użytkownika. Najmniej kompatybilny z Matlabem jest Rlab, jednak według autora celem nie było sklonowanie Matlaba, a jedynie przejęcie jego najlepszych cech i obudowanie ich językiem o poprawionej semantyce.

Z omawianym wcześniej Pythonem rozszerzonym np. o moduł SciPy możemy pracować podobnie, jak ze środowiskami do obliczeń numerycznych, tzn. interaktywnie lub tworząc programy automatyzujące obliczenia. Różnica między tymi środowiskami a Pythonem jest mniej więcej taka, jak między Fortranem a C++ - łatwiej tworzy się w nich programy czysto numeryczne, natomiast trudniej zastosować je do innych celów.

1.5 Programy do wizualizacji danych

Ludzki mózg tylko w ograniczonym zakresie przyswaja informacje zakodowane w ciągach liczb, natomiast całkiem nieźle radzi sobie z obrazami. Dlatego, o ile to tylko możliwe, powinniśmy prezentować uzyskane w trakcie obliczeń dane w formie graficznej.

Jednym z popularnych wśród naukowców programów do wizualizacji danych jest Gnuplot [23]. Pozwala tworzyć wykresy dwu- i trójwymiarowe, manipulować danymi, zapisywać wyniki do wielu formatów graficznych (patrz dodatek D). Dostępny jest przy tym na większość współczesnych platform

Gnuplot działa w trybie tekstowym. Chociaż moim zdaniem to jedna z jego wielu zalet, zdaję sobie sprawę, że nie wszyscy mogą tak to odbierać. Interesującym programem do wizualizacji danych pracującym w trybie graficznym jest Grace [24]. Jego interfejs jest może na początku mało intuicyjny, ale możliwości imponujące. Podobnie jak Gnuplot, oprócz samej wizualizacji Grace oferuje funkcje do analizy danych i manipulowania nimi. Program działa we wszystkich systemach uniksowych. Istnieją również jego porty na systemy z rodziny Windows, VMS i OS/2.

Innym ciekawym programem graficznym jest SciGraphica [25], która pretenduje do miana klona komercyjnego (i drogiego!) pakietu Origin (`http://www.originlab.com/`). Wprawdzie nie wiem, w jakim stopniu udało się autorom

zrealizować ten cel, ale stworzyli intuicyjny program o dużych możliwościach, pozwalający tworzyć profesjonalne wykresy.

1.6 Inne programy wspierające obliczenia

Inną kategorię programów ułatwiających życie naukowcom i inżynierom stanowią pakiety do obliczeń symbolicznych (ang. *Computer Algebra System*, CAS). Programy tego typu operują na wyrażeniach zbudowanych z wielomianów, funkcji elementarnych, macierzy, całek i pochodnych funkcji, a do typowych operacji należą: upraszczanie wyrażeń, podstawianie wyrażeń symbolicznych za zmienne, redukcja wyrazów podobnych, rozwijanie iloczynów, rozkład wyrażeń na czynniki, różniczkowanie i całkowanie symboliczne, rozwiązywanie niektórych typów równań i ich układów, rozwiązywanie równań różniczkowych określonych typów, obliczanie granic funkcji i ciągów, obliczanie sum szeregów, rozwijanie funkcji w szereg, operacje na macierzach, obliczenia związane z teorią grup itp. Dodatkowo wiele pakietów potrafi przeprowadzić obliczenia numeryczne i oferuje narzędzia do wizualizacji danych.

Możliwości symbolicznego manipulowania nawet skomplikowanymi wyrażeniami przydadzą się również w obliczeniach numerycznych. Już w następnym rozdziale dowiemy się, że nie wszystkie algorytmy matematyczne są numerycznie równoważne. Dlatego na potrzeby obliczeń numerycznych czasami trzeba przekształcić wyjściowy problem. Ponadto przy pomocy takich programów można dużo wydajniej, niż na kartce, rozwijać i testować nowe algorytmy.

Z programów CAS największe możliwości oferują pakiety Mathematica (`http://www.wolfram.com/`) i Mapple (`http://www.maplesoft.com/`), jednak ich cena w Polsce potrafi odstraszyć większość studentów. Na szczęście wśród darmowych programów do algebry symbolicznej znajdziemy kilka o dość dużych możliwościach. Ustępują one wprawdzie pod pewnymi względami swoim komercyjnym odpowiednikom, jednak można z nimi wydajnie pracować. Najbardziej kompletnym (o największych możliwościach) darmowym programem do obliczeń symbolicznych jest dostępna na wiele platform Maxima [26], która wywodzi się ze sławnej Macsymy - jednego z pierwszych tego typu pakietów na świecie. Oprócz Maximy możemy skorzystać z Axioma [27] bądź Yacasa [28]. Za darmo można również używać okrojonej wersji MuPADa (`http://www.sciface.com/`).

Rozdział 2

Dokładność w obliczeniach numerycznych

Rozważmy następujący przykład:

```
octave:1> x=2^30;
octave:2> x+2^-22==x, x+2^-23==x
ans=0
ans=1
```

Choć nieznacznie, liczba $x + 2^{-22}$ różni się od x, dlatego ich porównanie zwraca wartość 0 (czyli fałsz). Takiego samego wyniku oczekujemy również w drugim przypadku, kiedy x porównywany jest z $x+2^{-23}$. Tutaj jednak komputer zwraca wartość 1 (prawda), co oznacza, że traktuje obie liczby jako identyczne.

Ze względu na ograniczoną pamięć komputery nie mogą przechowywać liczb z dowolną dokładnością. Konieczne jest wprowadzenie pewnych przybliżeń (tzw. reprezentacji), w ramach których liczba daje się przedstawić za pomocą skończonej (ustalonej) ilości bitów. Powyższy przykład to właśnie efekt skończonej dokładności maszyny liczącej. Program, w którym dokonano obliczeń (w tym wypadku GNU Octave), używa zbyt małej ilości bitów, aby przedstawić sumę $x + 2^{-23}$ dokładnie.

Efekty związane ze skończoną dokładnością nie zawsze muszą stanowić poważny problem w obliczeniach. Licząc na kartkach większość z nas i tak pewnie zaniedbałaby 2^{-23} w porównaniu z 2^{30}. Należy być jednak świadomym tego, że nie zawsze wynik obliczeń na komputerze będzie taki jak tych dokładnych (gdyby te ostanie były możliwe do przeprowadzenia).

Zazwyczaj programista ma do wyboru kilka reprezentacji (typów danych), które różnią się długością słowa, czyli ilością bitów użytych do przedstawienia liczb. Ponadto inaczej na ogół reprezentowane są liczby całkowite, a inaczej - rzeczywiste.

2.1 Liczby całkowite

Liczby całkowite reprezentowane są na komputerze w sposób dokładny, choć oczywiście jest ich tylko skończona ilość. W zależności od tego, czy korzystamy

Liczba	Reprezentacja
0	00000000
2	00000010
51	00110011
127	01111111
255	11111111

Tabela 2.1: Liczba całkowita i jej 8-bitowa reprezentacja prosta. Dla $N = 8$ można przedstawić liczby z zakresu $0 \leqslant z \leqslant 2^8 - 1 = 255$.

z liczb całkowitych bez znaku, czy ze znakiem, mówimy o reprezentacji prostej lub uzupełnieniowej.

2.1.1 Reprezentacja prosta (bez znaku)

N-bitowa liczba całkowita z bez znaku reprezentowana jest w następująco:

$$z = (a_{N-1}, a_{N-2}, \ldots, a_0), \quad a_i = \{0, 1\}, \tag{2.1}$$

przy czym

$$z = \sum_{i=0}^{N-1} a_i * 2^i. \tag{2.2}$$

W reprezentacji tej można przedstawić liczby z zakresu $0 \leqslant z \leqslant 2^N - 1$. Przykłady 8-bitowej reprezentacji prostej liczb całkowitych znajdują się w tabeli 2.1.

2.1.2 Reprezentacja uzupełnieniowa do 2 (ze znakiem)

W przypadku liczb całkowitych ze znakiem zapis (2.1) interpretowany jest w nieco odmienny sposób, a mianowicie:

$$z = -a_{N-1} * 2^{N-1} + \sum_{i=0}^{N-2} a_i * 2^i. \tag{2.3}$$

Określona w ten sposób reprezentacja to najczęściej chyba spotykany w obliczeniach komputerowych zapis liczb całkowitych, tzw. reprezentacja uzupełnieniowa do 2 (w skrócie U2). Za jej pomocą można przedstawić liczby z zakresu $-2^{N-1} \leqslant z \leqslant 2^{N-1} - 1$. Przy tym, liczby dodatnie reprezentowane są przez zwykłe liczby binarne. Natomiast liczby ujemne tworzymy, zamieniając w liczbie dodatniej o tej samej wartości bezwzględnej bit po bicie na przeciwny, a następnie dodając 1 do wyniku (to właśnie jest uzupełnienie do dwójki):

$$\begin{aligned} +2_{10} &= 00000010_{U2} \\ &\Downarrow \quad \text{zamieniamy bity na przeciwne} \\ -3_{10} &= 11111101_{U2} \\ &\Downarrow \quad \text{dodajemy 1} \\ -2_{10} &= 11111110_{U2} \end{aligned} \tag{2.4}$$

Liczba	Reprezentacja
127	01111111
51	00110011
2	00000010
-2	11111110
-51	11001101
-128	10000000

Tabela 2.2: Liczba całkowita i jej 8-bitowa reprezentacja z uzupełnieniem do 2. Dla $N = 8$ można przedstawić liczby z zakresu $-128 \leqslant z \leqslant 127$.

Kilka liczb całkowitych w reprezentacji uzupełnieniowej przedstawionych jest w tabeli 2.2.

Główną zaletą reprezentacji uzupełnieniowej jest to, że nie trzeba rozróżniać liczb na dodatnie i ujemne przy dodawaniu. Po prostu dodajemy je do siebie bit po bicie:

+2	0	0	0	0	0	0	1	0
-1	1	1	1	1	1	1	1	1
+1	0	0	0	0	0	0	0	1

Warto zauważyć, że mnożenie przez 2 w tej reprezentacji odpowiada przesunięciu bitów o jedną pozycją w lewo, a dzielenie przez 2 - o jedną pozycję w prawo:

$$
\begin{array}{c}
4 \\
\Downarrow \\
00000100 \\
\Downarrow \\
00001000 \quad \text{przesuwamy o jeden w lewo} \\
\Downarrow \\
8 = 4 * 2
\end{array}
$$

2.1.3 Typowe problemy w obliczeniach

Liczby całkowite reprezentowane są na komputerach dokładnie, więc również wyniki działań na nich powinny być dokładne. Z problemami możemy mieć do czynienia jedynie wtedy, gdy wynik działania przekroczy dozwolony w danym typie zakres. Nie ma standardów określających zachowanie systemu w takiej sytuacji, jednak w praktyce komputerowe typy całkowite mają najczęściej strukturę pierścienia, w której zwiększenie liczby największej o jeden da w wyniku liczbę najmniejszą. I odwrotnie - zmniejszenie liczby najmniejszej o jeden da liczbę największą.

Poniżej przedstawiony jest prosty program w C++ wyliczający silnię podanej liczby całkowitej:

```
#include <iostream>

int silnia (int n)
{
   if(n<=1)
```

```
    return 1;
  else
    return n*silnia(n-1);
}

int main()
{
  int n;
  std::cout << "Podaj liczbę: ";
  std::cin >> n;
  std::cout << n << "!=" << silnia(n) << "\n";
}
```

Efekt kilku wywołań tego programu jest następujący:

```
Podaj liczbę: 12
12!=479001600
Podaj liczbę: 16
16!=2004189184
Podaj liczbę: 17
17!=-288522240
```

Chociaż zgodnie z definicją matematyczną silnia liczby naturalnej jest większa od zera, w powyższym przykładzie pojawił się wynik ujemny. To właśnie efekt pierścieniowej struktury typu `int` i wyjścia poza dozwolony zakres przy obliczaniu 17!.

2.1.4 Implementacje

C++

W C++ mamy do dyspozycji kilka typów całkowitych, zarówno ze znakiem, jak i bez (patrz tabela 2.3). Różnią się one między sobą ilością bitów używanych do zapisania liczb, a co za tym idzie, zakresem liczb możliwych do przedstawienia na komputerze oraz szybkością obliczeń wykonywanych na nich. Warto wiedzieć, że na różnych platformach te same typy mogą się różnić między sobą długością. Dlatego minimalne i maksymalne wartości możliwe do przedstawienia w każdym typie danych na konkretnej platformie zapisane są w pliku nagłówkowym `climits`.

Gdybyśmy potrzebowali liczb całkowitych większych od maksymalnej wartości reprezentowanej w typie `long int`, możemy skorzystać z jednej z wielu bibliotek, np. C++ BIL [29] lub GMP [30], pozwalających na operowanie na liczbach, których wielkość ograniczona jest jedynie pamięcią komputera.

Fortran

W tabeli 2.4 zestawione zostały dostępne w Fortranie typy całkowite. Odpowiadają one typom ze znakiem w C++ z tą różnicą, że Fortran oferuje typ o długości 64 bitów również na platformach 32-bitowych.

Python

W Pythonie mamy do dyspozycji dwa typy całkowite. Typ `int` odpowiada 32-bitowemu typowi `int` z C++ i pozwala na przedstawienie liczb z zakresu

Typ	Ilość bitów	Zakres
`signed char`	8	$-128 \leqslant z \leqslant 127$
`unsigned char`	8	$0 \leqslant z \leqslant 255$
`short`	16	$-32768 \leqslant z \leqslant 32767$
`unsigned short`	16	$0 \leqslant z \leqslant 65535$
`int`	32	$-2147483648 \leqslant z \leqslant 2147483647$
`unsigned int`	32	$0 \leqslant z \leqslant 4294967295$
`long int`	32	$-2147483648 \leqslant z \leqslant 2147483647$
`unsigned long int`	32	$0 \leqslant z \leqslant 4294967295$

Tabela 2.3: Typy całkowite w C++ dostępne na 32-bitowej platformie Intela (kompilator GCC v.4.0.3). Dla porównania, typ `long int` w tym samym kompilatorze na platformie AMD64 ma długość 64 bitów, co pozwala na zapisanie liczb z zakresu $-9223372036854775808 \leqslant z \leqslant 9223372036854775807$.

Typ	Ilość bitów	Zakres
`INTEGER(KIND=1)`	8	$-128 \leqslant z \leqslant 127$
`INTEGER(KIND=2)`	16	$-32768 \leqslant z \leqslant 32767$
`INTEGER(KIND=4)`	32	$-2147483648 \leqslant z \leqslant 2147483647$
`INTEGER(KIND=8)`	64	$-9223372036854775808 \leqslant z \leqslant 9223372036854775807$

Tabela 2.4: Typy całkowite w Fortranie dostępne na 32-bitowej platformie Intela (kompilator IFC v. 8.0).

$-2147483648 \leqslant z \leqslant 2147483647$ (maksymalna wartość możliwa do przedstawienia przechowywana jest w zmiennej `maxint` w module `sys`). Mocną stroną Pythona jest wbudowany typ `long int`, który pozwala na przedstawienie liczb o rozmiarze ograniczonym jedynie pamięcią komputera. Przy tym, jeżeli w obliczeniach zdarzy się wyjście poza zakres typu `int`, liczby zostaną automatycznie przekonwertowane na typ `long int`. Aby zilustrować ostatnią własność, rozważmy ponownie funkcję obliczającą wartość silni, tym razem w wersji „pythonowej”:

```
>>> def silnia(x):
...     if x<=1:
...             return 1
...     return x*silnia(x-1)
...
>>> silnia(4)
24
>>> silnia(17)
355687428096000L
>>> silnia(100)
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000L
```

Mimo, że 17! wykracza znacznie poza zakres liczb, które można przedstawić za pomocą 32-bitów, zdefiniowana przez nas funkcja zwróciła poprawną wartość. Jest to możliwe dzięki temu, że Python automatycznie dokonał konwersji między swoim podstawowym a rozszerzonym typem całkowitym. O tym, że mamy do czynienia z typem `long int`, informuje nas dołączona do liczby litera `L`

GNU Octave

GNU Octave nie ma specjalnego typu danych dla liczb całkowitych. Wszystkie liczby przechowywane są jako 64-bitowe liczby zmiennopozycyjne (patrz następny podrozdział). Maksymalna wartość możliwa do przedstawienia przechowywana jest w zmiennej wbudowanej `realmax`, a wyjście poza zakres traktowane jest jako nieskończoność:

```
octave:1> realmax
realmax =  1.7977e+308
octave:2> 2*realmax
ans = Inf
```

2.2 Liczby rzeczywiste

Najczęściej obecnie stosowanym sposobem zapisu liczb rzeczywistych na komputerach jest reprezentacja zmiennoprzecinkowa [31][1]. W reprezentacji tej liczba przedstawiona jest za pomocą **bazy** B oraz **precyzji** (ilości cyfr) p. Jeśli np. $B = 10$ i $p = 3$, wówczas liczba 0.1 będzie miała postać 1.00×10^{-1}.

Ze względu na skończoną dokładność (ilość bitów) nie wszystkie liczby rzeczywiste dadzą się przedstawić dokładnie. Dla $B = 2$ i $p = 24$ liczba 0.1 będzie miała np. tylko reprezentację przybliżoną

$$1.10011001100110011001100 \times 10^{-4}.$$

Ogólnie, liczba zmiennopozycyjna w bazie B i precyzji p ma postać:

$$\pm d.dd\ldots d \times B^e, \tag{2.5}$$

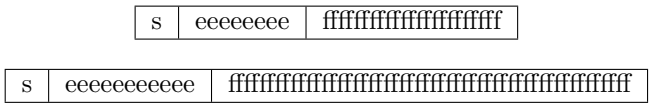
gdzie **mantysa** $d.dd\ldots d$ ma p cyfr. Bardziej dokładnie, powyższy zapis oznacza:

$$\pm \left(d_0 + d_1 B^{-1} + \cdots + d_{p-1} B^{-(p-1)} \right) \times B^e,$$

gdzie $0 \leqslant d_i \leqslant B$. Oczywiście, reprezentacja zmiennopozycyjna liczby nie musi być jednoznaczna. Na przykład, obie liczby 0.01×10^1 i 1.00×10^{-1} przedstawiają liczbę 0.1 w bazie $B = 10$ i precyzji $p = 3$. Dlatego wprowadza się pojęcie **reprezentacji znormalizowanej**, w której $d_0 \neq 0$. I tak 1.00×10^{-1} jest znormalizowaną reprezentacją liczby 0.1, natomiast 0.01×10^1 nią nie jest.

Wymóg, aby reprezentacja była znormalizowana, czyni ją jednoznaczną, wprowadza jednak pewien problem. Otóż niemożliwym staje się przedstawienie zera w naturalny sposób. W takim wypadku przyjmuje się najczęściej, że 0 reprezentowane jest przez $1.0 \times B^{e_{min}-1}$.

[1] O innych sposobach można przeczytać np. w [32, 33]



Rysunek 2.1: Ułożenie bitów w standardzie IEEE 754 w pojedynczej (góra) i podwójnej (dół) precyzji.

2.2.1 Standard IEEE 754

Nowoczesne maszyny cyfrowe używają najczęściej reprezentacji zmiennoprzecinkowej zgodnej (w mniejszym lub większym) stopniu ze standardem IEEE 754 [31]. Standard ten wymaga $B = 2$ i $p = 24$ dla liczb w pojedynczej oraz $p = 53$ w podwójnej precyzji[2]. W pojedynczej precyzji liczba maszynowa g będzie miała postać:

$$g = (-1)^s * 1.f \times 2^{e-127} \tag{2.6}$$

Bit s określa znak liczby. 23 bity f reprezentują część ułamkową mantysy, natomiast cecha (wykładnik) jest reprezentowana przez 8 bitów. Przesunięcie wykładnika o 127 (w podwójnej precyzji o 1023) powoduje, że do jego przedstawienia wystarczą liczby dodatnie. Na rysunku 2.1 widzimy, jak poszczególne bity umieszczone są w słowie.

W pojedynczej precyzji można przedstawić liczby rzeczywiste z zakresu

$$-3,4 \times 10^{38} \leqslant g \leqslant 3,4 \times 10^{38},$$

natomiast w podwójnej precyzji z zakresu

$$-1,8 \times 10^{308} \leqslant g \leqslant 1,8 \times 10^{308}.$$

Standard IEEE 574 definiuje dodatkowo reprezentację pewnych specjalnych wartości:

0 00000000 00000000000000000000000	0
1 00000000 00000000000000000000000	-0
0 11111111 00000000000000000000000	$+\infty$
1 11111111 00000000000000000000000	$-\infty$
0 11111111 00000100000000000000000	NaN
1 11111111 00100010001001010101010	NaN

NaN, czyli „nieliczba” (ang. *Not a Number*) to wynik następujących działań:

$\infty + (-\infty)$
$0 \times \infty$
$0/0, \infty/\infty$
$x \ mod \ 0, \ \infty \ mod \ y$
$\sqrt{x}, \ x < 0$

Jeżeli w pośrednim kroku obliczeń pojawi się NaN, wynik końcowy też będzie „nieliczbą”.

[2]Oprócz tego zdefiniowane są jeszcze dwie precyzje: pojedyncza rozszerzona ($p = 32$) i podwójna rozszerzona ($p = 64$). Nie będziemy się jednak bliżej nimi zajmować.

Nieskończoności pozwalają na kontynuowanie obliczeń w przypadku wyjścia poza możliwy do przedstawienia zakres liczb. Jeżeli uzyskamy ∞ w pośrednim kroku obliczeń, wynik końcowy może być dobrze zdefiniowaną liczbą zmiennoprzecinkową, ponieważ obowiązuje $1/\infty = 0$.

Dziwić może trochę zero ze znakiem. Jednak dzięki temu zawsze spełniona będzie równość $1/(1/x) = x$ dla $x = \pm\infty$. Gdyby zero pozbawione było znaku, spełnienie tej nierówności nie byłoby możliwe, ponieważ informacja o znaku byłaby tracona w pośrednich krokach obliczeń. Ponadto zero ze znakiem jest użyteczne w przypadku funkcji, które mają nieciągłości w zerze [31].

W celu zwiększenia dokładności obliczeń standard IEEE wprowadza również wartości zdenormalizowane, których mantysa nie posiada domyślnej części całkowitej 1. Liczby te mają wszystkie bity cechy równe 0 i przynajmniej jeden bit mantysy różny od zera. Bez wartości zdenormalizowanych najmniejsza co do modułu liczba różna od zera możliwa do przedstawienia w pojedynczej precyzji to $\pm 1,1754 \times 10^{-38}$, a w podwójnej precyzji - $\pm 2,225 \times 10^{-308}$. Liczby mniejsze od wartości granicznej w każdej precyzji traktowane byłyby jako 0. Dzięki wartościom zdenormalizowanym najmniejsze liczby to odpowiednio $\pm 1,40 \times 10^{-45}$ i $\pm 4,94 \times 10^{-324}$.

Przykład Aby oswoić się z formatem IEEE, znajdziemy 32-bitową reprezentację zmiennoprzecinkową liczby 5,375. W pierwszym kroku przekształcamy liczbę dziesiętną do postaci dwójkowej:

$$5,375 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} + 1 * 2^{-3} \ \rightarrow \ 101.011.$$

Otrzymaną liczbę normalizujemy:

$$101.011 \times 2^0 = 1.01011 \times 2^2,$$

a następnie pomijamy wiodącą jedynkę w mantysie

$$01011$$

i obliczamy wykładnik

$$2 + 127 = 129 \ \rightarrow \ 10000001$$

Pozostaje nam tylko określenie bitu znaku, który w przypadku 5,375 wynosi 0 (liczba dodatnia). Reprezentacja naszej liczby ma postać:

znak	wykładnik	mantysa
0	10000001	01011000000000000000000

2.2.2 Implementacje

C++

Programista C++ ma do dyspozycji trzy typy rzeczywiste, `float`, `double` i `long double` (patrz tabela 2.5). Odpowiadają one pojedynczej, podwójnej i rozszerzonej podwójnej precyzji w standardzie IEEE. Warto przy tym wiedzieć, że chociaż zmienna typu `long double` zajmuje 96 bitów, na komputerach 32-bitowych wykorzystywane jest tylko 80 z nich. Najmniejsze i największe wartości

Typ	Bity	Największa liczba
`float`	32	3.40282×10^{38}
`double`	64	1.79769×10^{308}
`long double`	96	1.18974×10^{4932}

Tabela 2.5: Typy zmiennopozycyjne w C++.

możliwe do przedstawienia w każdym z nich zdefiniowane są w pliku nagłówkowym `cfloat`.

W dobie szybkich i tanich komputerów wyposażonych w dużo pamięci RAM typ `float` stracił na znaczeniu. Obecnie wykorzystuje się go przede wszystkim w sytuacjach, kiedy należy zapamiętać naprawdę ogromne ilości danych i każdy bajt staje się na wagę złota. Podstawowym typem w obliczeniach stał się `double`. Natomiast `long double` używa się najczęściej do przechowania pośrednich wyników obliczeń.

Reprezentację IEEE liczb można podglądnąć przy pomocy funkcji

```
void gsl_ieee_printf_float (const float * x)
```

oraz

```
void gsl_ieee_printf_double (const double * x)
```

z biblioteki GSL [12]. Ich działanie ilustruje poniższy program:

```
#include <iostream>
#include <gsl/gsl_ieee_utils.h>

int main ()
{

    float f = 1.0/3.0;
    std::cout << "Reprezentacja liczby " << f << ":\n" ;
    gsl_ieee_printf_float(&f);
    std::cout << "\n";
}
```

Wynik jego działania jest następujący:

```
Reprezentacja liczby 0.333333:
 1.01010101010101010101011*2^-2
```

Fortran

Typy zmiennopozycyjne w Fortranie odpowiadają tym w C++. Zestawione są one w tabeli 2.6. Podobnie jak w C++, rozszerzony typ na procesorach 32-bitowych wykorzystuje jedynie 80 bitów.

Największą liczbę możliwą do przedstawienia w danym typie danych można odczytać za pomocą wbudowanej funkcji `huge()`, której argumentem jest zmienna interesującego nas typu.

Typ	Bity	Największa liczba
`real(kind=4)`	32	3.40282×10^{38}
`real(kind=8)`	64	1.79769×10^{308}
`real(kind=16)`	128	1.18974×10^{4932}

Tabela 2.6: Typy zmiennopozycyjne w Fortranie.

Python

Python ma jeden 64-bitowy wbudowany typ zmiennopozycyjny, który odpowiada typowi `double`. Wyjście poza dozwolony zakres traktowane jest jako nieskończoność:

```
>>> x=1.7*10**308
>>> 2*x
inf
```

GNU Octave

GNU Octave posiada jeden, 64-bitowy typ zmiennopozycyjny, który odpowiada typowi `double` w C++. Największa liczba możliwa do przedstawienia oraz najmniejsza liczba znormalizowana przechowywane są w zmiennych `realmax` i `realmin`:

```
octave:1> realmax
realmax =  1.7977e+308
octave:2> realmin
realmin =  2.2251e-308
```

Należy jednak zwrócić uwagę, że `realmin` nie jest najmniejszą liczbą zmiennopozycyjną różną od zera:

```
octave:3> realmin/2
ans =  1.1125e-308
```

Innymi słowy, GNU Octave używa również zdenormalizowanych liczb zmiennopozycyjnych.

2.2.3 Typowe problemy w obliczeniach

W większości przypadków wynikom obliczeń zmiennopozycyjnych można ufać. Na ich korzyść przemawia m.in. ogromny postęp, jaki dokonał się w wielu dziedzinach nauki właśnie dzięki maszynom cyfrowym i obliczeniom przeprowadzonym za ich pomocą. Mimo to czasami wyniki uzyskane na komputerach potrafią zaskoczyć ludzi, również tych biegłych w matematyce. Dlatego poniżej opisanych zostanie kilka takich „dziwnych" rezultatów. Więcej szczegółów na ten temat można znaleźć w artykułach D. Goldberga [31] i B. Busha [34].

Niedokładności

Wiele dziwnych na pierwszy rzut oka wyników bierze się stąd, że obliczenia zmiennopozycyjne wykonywane są w bazie $B = 2$, natomiast na zewnątrz liczby

reprezentowane są najczęściej w bazie $B = 10$. Dlatego pisząc w kodzie programu np. `x=0.01` intuicyjnie oczekujemy, że liczba 0,01 będzie reprezentowana dokładnie, a tak niestety nie jest. Aby się o tym przekonać, rozważmy następujący program:

```
#include <iostream>

int main()
{
   float x=0.01;
   if(100.0*x==1.0)
     std::cout << "Równe :)\n";
   else
     std::cout << "Nierówne :(\n";
}
```

Teoretycznie równość powinna być spełniona. Jednak w większości systemów po wywołaniu programu zobaczymy na ekranie

```
Nierówne :(
```

Ponieważ liczba $0,01$ nie ma dokładnej reprezentacji w standardzie IEEE, przybliżana jest najbliższą liczbą maszynową (czyli taką, która daje się przedstawić dokładnie). Dlatego równość nie jest spełniona.

Ze względu na niedokładności związane z przedstawieniem liczb nie wszystkie algorytmy równoważne matematycznie będą jednocześnie równoważne numerycznie (patrz rozdział 3). I znowu, zilustrujmy to prostym przykładem:

```
#include <iostream>

int main()
{
   float x=77777.0, y=7.0;
   float y1 = 1.0/y;
   float z= x/y;
   float z1 = x*y1;
   if(z==z1)
     std::cout << "Równe :)\n";
   else
     std::cout << "Nierówne :(\n";
}
```

Po wykonaniu tego programu znowu okaże się, wbrew oczekiwaniom wielu początkujących numeryków, że wartości zmiennych `z` i `z1` nie są równe. Podobnie, jak w poprzednim przykładzie, przyczyną jest to, że $y = 1.0/7.0$ nie ma dokładnej reprezentacji w standardzie IEEE.

Następny przykład ilustruje kolejny efekt związany z niedokładnością reprezentacji liczb dziesiętnych - bity nieznaczące:

```
#include <iostream>

int main()
{
   float y=1000.2;
   float x=y-1000.0;
   std::cout << x <<"\n";
}
```

Liczba $1000,0$ ma dokładną reprezentację zmiennopozycyjną, natomiast $1000,2$ daje się przedstawić tylko w przybliżeniu. Dlatego komputerowy wynik powyższego działania będzie się różnił trochę od wartości dokładnej $0,2$:

```
0.200012
```

Rozszerzona precyzja

Rodzina procesorów Intela należy do systemów o rozszerzonej precyzji (ang. *extended-based systems*) , które wewnętrznie zapisują liczby z dokładnością większą, niż wymaga tego używany typ danych. Pozwala to na ogół liczyć dokładniej niż na architekturach pozbawionych tej właściwości (np. większość procesorów RISC), czasami prowadzi jednak do pewnych niepożądanych efektów ubocznych:

```
#include <iostream>

int main()
{
   float a=10.0,b=3.0,x=10.0,y=3.0;
   float c=a/b;
   float z=c-(x/y);
   std::cout << z <<"\n";
}
```

Iloraz a/b liczony jest w rozszerzonej precyzji. Wynik zostaje przekształcony do pojedynczej precyzji i zapamiętany w zmiennej `c`. Jeżeli przy tym przekształceniu następuje utrata dokładności, `z` będzie raczej różne od zera:

```
-7.94729e-08
```

Porównywanie liczb zmiennopozycyjnych

Porównywanie liczb zmiennoprzecinkowych samo w sobie może prowadzić do dziwnych wyników, zwłaszcza na systemach, które przechowują liczby w rozszerzonej precyzji. Dla przykładu rozważmy następujący program:

```
#include <iostream>

int main()
{
   float x=3.0/7.0;
   if(x==3.0/7.0)
     std::cout << "Równe :)\n";
   else
     std::cout << "Nierówne :(\n";
}
```

Po skompilowaniu programu na 32-bitowy procesor zgodny z architekturą x86 Intela, na ekranie zobaczymy znany już wynik:

```
Nierówne :(
```

Powodem takiego zachowania programu jest znowu rozszerzona precyzja, z jaką liczby przechowywane są w wewnętrznych rejestrach i jej utrata przy konwersji

na typ używany w programie. Dlatego należy unikać bezpośrednich porównań liczb zmiennoprzecinkowych, a jeżeli algorytm wymaga od nas takiego porównywania, bezpieczniej jest traktować liczby jako równe, jeżeli moduł ich różnicy jest mniejszy od małej, z góry zadanej wartości:

```
#include <iostream>
#include <cmath>

int main()
{
   float eps=1.0e-6;
   float x=3.0/7.0;
   if(fabs(x-3.0/7.0)<eps)
     std::cout << "Równe :)\n";
   else
     std::cout << "Nierówne :(\n";
}
```

```
Równe :)
```

Chociaż metoda dała pożądany wynik, należy być świadomym jej ograniczeń. Określenie odpowiedniego ϵ nie zawsze jest sprawą oczywistą. Ponadto, relacja zdefiniowana poprzez

$$a \sim b \Leftrightarrow |a - b| < \epsilon$$

nie jest relacją równoważności, ponieważ z $a \sim b$ i $b \sim c$ wcale nie wynika $a \sim c$.

Jeżeli w programie wykonujemy wiele porównań i nie chcemy do każdego z nich dobierać nowego ϵ, wówczas wybieramy jego wartość tak, aby określał maksymalny błąd względny (patrz podrozdział 2.3), jaki gotowi jesteśmy popełnić, traktując dwie liczby zmiennoprzecinkowe jako równe, a następnie porównywać moduł ich różnicy nie z samym ϵ, a raczej z iloczynem ϵ i modułu jednej z porównywanych liczb:

```
#include <iostream>
#include <cmath>

int main()
{
   float eps=1.0e-6;
   float x=3.0/7.0;
   if(fabs(x-3.0/7.0)<eps*fabs(x))
     std::cout << "Równe :)\n";
   else
     std::cout << "Nierówne :(\n";
}
```

```
Równe :)
```

Dzięki temu porównanie zostanie „dostrojone” do wielkości liczb, z którymi aktualnie pracujemy.

2.3 Dokładność w obliczeniach numerycznych

2.3.1 Źródła błędów

Na wyniki obliczeń numerycznych wpływa wiele rodzajów błędów ograniczających ich dokładność [31, 35]:

- błędy danych wejściowych,
- zaokrąglenia w czasie obliczeń,
- błędy zakresu,
- błędy obcięcia,
- zaniedbywalne składniki,
- utrata cyfr znaczących,
- uproszczenia modelu matematycznego,
- błędy człowieka i maszynowe.

Z błędami danych wejściowych mamy do czynienia wówczas, gdy wartości wejściowe obarczone są niepewnością (jak to zwykle bywa w pomiarach fizycznych). Do tej klasy zaliczamy też niedokładności powstające przy przekształcaniu danych wejściowych do reprezentacji zmiennopozycyjnej. Wiemy już, że liczb maszynowych jest skończenie wiele, więc nawet jeśli dane wejściowe znane są dokładnie, jest wysoce nieprawdopodobne, że wszystkie one będą reprezentowane dokładnie w pamięci komputera.

Podobnymi z natury są również zaokrąglenia w czasie obliczeń. Jeśli wynik nie będzie liczbą maszynową, zostanie zaokrąglony do najbliższej z nich.

Z błędami zakresu mieliśmy już do czynienia wcześniej. Wynikają one z faktu, że niektóre działania, nawet jeśli ich argumenty leżą w dozwolonym zakresie liczb, mogą poza ten zakres wyprowadzić.

Błędy obcięcia powstają wtedy, gdy proces obliczania granicy zostaje przerwany przed osiągnięciem wartości granicznej. Typowym przykładem jest przybliżanie sumy szeregu nieskończonego sumą skończonej ilości składników.

Zaniedbywalne składniki i utrata cyfr znaczących to w zasadzie błędy zaokrągleń. Jednak mają one tak duże znaczenie, że zostały ujęte w oddzielnych kategoriach i zostaną opisane w dalszej części tego rozdziału.

Uproszczenia modelu matematycznego to chleb powszedni każdego fizyka. Mamy z nimi do czynienia wszędzie tam, gdzie do opisu rzeczywistej sytuacji używa się wyidealizowanego modelu matematycznego.

Do kategorii błędów człowieka i maszynowych należą wszelkie pomyłki oraz niedokładności wynikające z używania wadliwego sprzętu bądź oprogramowania.

Niektórych błędów w obliczeniach można uniknąć, na inne ma się niewielki wpływ. W każdym razie wskazane jest, aby podczas obliczeń numerycznych testować pośrednie ich etapy i/lub wyniki końcowe. W problemach fizycznych można sprawdzać np. zasady zachowania odpowiednich wielkości (oczywiście ze skończoną dokładnością). Ogólnie testowanie nie jest zadaniem łatwym, często wymaga dużego doświadczenia i dodatkowych informacji o problemie, pozwala jednak lepiej kontrolować sam proces obliczeń oraz jego wyniki.

2.3.2 Błędy bezwzględne i względne

Niech x będzie wartością przybliżoną wielkości X.

Definicja Błędem bezwzględnym wartości x nazywamy $\epsilon_x = x - X$.

Definicja Błędem względnym wartości x nazywamy $\rho_x = \frac{\epsilon_x}{X}$, jeśli $X \neq 0$.

Błędy można również zdefiniować z przeciwnym do powyższych definicji znakiem. Ważne jest tylko, aby wybraną konwencję stosować konsekwentnie.

Często znana jest tylko wartość przybliżona x oraz błąd bezwzględny ϵ_x, natomiast „prawdziwa" wartość X pozostaje nieznana. Wówczas możemy przybliżyć błąd względny wartością $\rho_x \simeq \epsilon_x/x$, pod warunkiem, że $x \neq 0$.

2.3.3 Cyfry poprawne i znaczące

Mówiąc o ułamku dziesiętnym, możemy mówić o cyfrach ułamkowych oraz o cyfrach istotnych. Cyfry ułamkowe to wszystkie cyfry po kropce dziesiętnej. Natomiast przy cyfrach istotnych nie uwzględnia się zer na początku ułamka.

Przykład Liczba 0,00245 ma pięć cyfr ułamkowych, ale tylko 3 cyfry istotne. 12,12 ma cztery cyfry istotne i dwie ułamkowe.

Jeżeli moduł błędu wartości x nie przewyższa $\frac{1}{2} * 10^{-t}$, to mówimy, że x ma t **poprawnych cyfr ułamkowych**. Cyfry istotne występujące w x do pozycji t–tej po kropce nazywamy **cyframi znaczącymi**.

Przykład Liczba $0,001234 \pm 0,0000004$ ma pięć cyfr poprawnych (ponieważ $0,000004 < \frac{1}{2} * 10^{-5}$) i trzy znaczące.

Liczba cyfr poprawnych daje pojęcie o wielkości błędu bezwzględnego, a liczba cyfr znaczących - o wielkości błędu względnego.

2.3.4 Ucinanie i zaokrąglanie

Istnieją dwie metody skracania liczb do danej długości t cyfr ułamkowych [36]. Ucinanie polega na odrzucaniu cyfr na prawo od t–tej. W metodzie tej wartość błędu bezwzględnego może dochodzić do 10^{-t}. Ponadto wprowadzany błąd ma systematycznie znak przeciwny niż sama liczba.

Zaokrąglanie polega na wyborze, wśród liczb mających t cyfr ułamkowych, liczby najbliższej do danej. Niech $|X| = 0.\alpha_1\alpha_2 \ldots \alpha_t\alpha_{t+1} \ldots$. Wówczas:

$$|x| := \begin{cases} 0.\alpha_1\alpha_2 \ldots \alpha_t & \text{jeśli } 0 \leqslant \alpha_{t+1} \leqslant 4 \\ 0.\alpha_1\alpha_2 \ldots \alpha_t + 10^{-t} & \text{jeśli } \alpha_{t+1} \geqslant 5 \end{cases} \tag{2.7}$$

Czasami w granicznym przypadku, kiedy $\alpha_{t+1} = 5$, można zwiększać α_t o 1, jeśli jest ona nieparzysta lub pozostawiać bez zmian, jeśli jest parzysta. Wówczas błędy dodatnie i ujemne są jednakowo częste. Jednak maszyny liczące, w których wykonywane jest zaokrąglanie, dodają zawsze 1 w granicznym przypadku, gdyż jest to łatwiejsze w realizacji. Błąd zaokrąglenia leży w przedziale $(-\frac{1}{2} * 10^{-t}, \frac{1}{2} * 10^{-t})$.

Przykład Skracanie liczby do trzech cyfr ułamkowych:

liczba	zaokrąglanie	ucinanie
0,2379	0,238	0,237
-0,2379	-0,238	-0,237
0,2375	0,238	0,237

2.3.5 Przenoszenie się błędów

O przenoszeniu się błędów mówimy wtedy, gdy jakaś wielkość wyliczana jest z innych, obarczonych błędami. W takim wypadku wskazane jest oszacować niedokładność wyniku naszych obliczeń.

Dodawanie i odejmowanie

Zastanówmy się najpierw, jak błędy zachowują się przy dodawaniu i odejmowaniu. Niech

$$\begin{aligned} \epsilon_x &= x - X, \\ \epsilon_y &= y - Y. \end{aligned} \tag{2.8}$$

Wówczas

$$\epsilon_{x\pm y} = (x \pm y) - (X \pm Y) = (x - X) \pm (y - Y) = \epsilon_x \pm \epsilon_y, \tag{2.9}$$

czyli

$$|\epsilon_{x\pm y}| \leqslant |\epsilon_x| + |\epsilon_y| . \tag{2.10}$$

Stosując zasadę indukcji matematycznej, powyższy wynik możemy uogólnić:

Twierdzenie 2.3.1 *Oszacowanie błędu bezwzględnego wyniku dodawania lub odejmowania jest sumą oszacowań błędów bezwzględnych składników.*

Pouczające będzie również sprawdzenie, co dzieje się z błędem względnym. Otóż

$$|\rho_{x\pm y}| = \frac{|\epsilon_{x\pm y}|}{|X \pm Y|} \leqslant \frac{|X| \frac{|\epsilon_x|}{|X|} + |Y| \frac{|\epsilon_y|}{|Y|}}{|X \pm Y|} = \frac{|X| |\rho_x| + |Y| |\rho_y|}{|X \pm Y|} \tag{2.11}$$

Teraz powinno być jasne, dlaczego przy utracie cyfr znaczących (paragraf 2.3.7), kiedy $|X - Y| \sim 0$, błąd względny ulega wzmocnieniu.

Mnożenie i dzielenie

Rozważmy teraz błąd popełniony przy mnożeniu dwóch liczb,

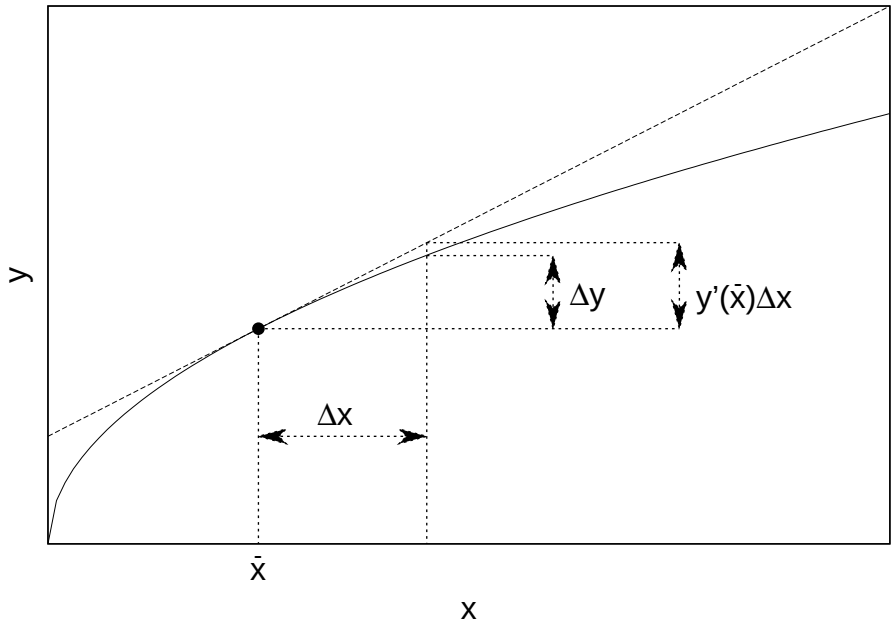
$$\epsilon_{xy} = xy - XY = (X \pm \epsilon_x)(y \pm \epsilon_y) - XY \simeq X\epsilon_y \pm Y\epsilon_x, \tag{2.12}$$

o ile tylko $\epsilon_x, \epsilon_y \ll 1$. Wówczas

$$|\epsilon_{xy}| \leqslant |Y| |\epsilon_x| + |X| |\epsilon_y| , \tag{2.13}$$

i w efekcie końcowym otrzymamy

$$|\rho_{xy}| = \frac{|\epsilon_{xy}|}{|XY|} \leqslant \frac{|Y| |\epsilon_x| + |X| |\epsilon_y|}{|XY|} = |\rho_x| + |\rho_y| . \tag{2.14}$$



Rysunek 2.2: Przybliżenie Δy za pomocą różniczki funkcji y.

W analogiczny sposób można pokazać, że błąd względny popełniony przy dzieleniu dwóch liczb ma to samo oszacowanie:

$$\left|\rho_{x/y}\right| \leqslant |\rho_x| + |\rho_y|\,, \tag{2.15}$$

co znowu da się prosto uogólnić do następującego twierdzenia:

Twierdzenie 2.3.2 *W mnożeniu i dzieleniu oszacowania błędu względnego argumentów (o ile tylko błędy te są $\ll 1$) dodają się.*

Ogólny wzór na przenoszenie się błędów

Rozważmy najpierw funkcję $y(x)$ jednej zmiennej. Chcemy oszacować błąd $\Delta y = y(\tilde{x}) - y(x^{(0)})$, gdzie $\tilde{x}$ jest przybliżoną wartością argumentu, którego wartość dokładna wynosi $x^{(0)}$. Jak wynika z rysunku 2.2, Δy można przybliżyć za pomocą różniczki funkcji y. Wielkość $|y'(x)|$ traktuje się przy tym jako miarę wrażliwości $y(x)$ na zakłócenie argumentu x.

Ogólnie, jeśli y jest funkcją $x = (x_1, \ldots, x_n)$, $\tilde{x} = (\tilde{x}_1, \ldots, \tilde{x}_n)$ to wartość przybliżona argumentu $x_0 = (x_1^0, \ldots, x_n^0)$, oraz $\Delta x_i = \tilde{x}_i - x_i^0$ i $\Delta y = y(\tilde{x}) - y(x_0)$, wówczas zachodzi:

Twierdzenie 2.3.3 *Ogólny wzór na przenoszenie się błędów ma postać:*

$$\Delta y \approx \sum_{i=1}^{n} \frac{\partial y}{\partial x_i}(\tilde{x})\Delta x_i \quad \Rightarrow |\Delta y| \leqslant \sum_{i=1}^{n} |\frac{\partial y}{\partial x_i}(\tilde{x})||\Delta x_i|. \tag{2.16}$$

Dowód w [36].

Z twierdzenia 2.3.3 wynika, że Δy aproksymuje się za pomocą różniczki zupełnej funkcji y. Stosując w praktyce wzór (2.16), oblicza się wartości $|\partial y/\partial x_i|$ w punkcie $\tilde{x}$ i zwiększa się je nieco (5%-10%) dla bezpieczeństwa.

Zaburzenia eksperymentalne

Analiza błędów ma duże znaczenie w planowaniu obliczeń, np. w wyborze algorytmów, w wyborze długości kroku w całkowaniu numerycznym itd. Jednak w wielu zadaniach obliczeniowych związki między danymi wejściowymi i danymi wyjściowymi są tak złożone, że trudno byłoby zastosować wzory na przenoszenie się błędów. Wówczas bada się wrażliwość danych wyjściowych na błędy danych wejściowych w testowych obliczeniach z zaburzeniami.

2.3.6 Zaniedbywalne składniki

Rozważmy następujący przykład:

```
octave:1> x=10^10; y=x; z=1/x;
octave:2> x+z-y
ans = 0
octave:3> x-y+z
ans =  1.0000e-10
```

Chociaż oba wyrażenia są matematycznie równoważne, dają numerycznie różne wyniki. Przyczyną jest znowu skończona ilość bitów w mantysie. Jeżeli dodajemy do siebie dwie liczby o różnych wykładnikach, muszą one zostać sprowadzone do wspólnej cechy. Na ogół odbywa się to przez denormalizację mniejszej z liczb. Jeżeli rzędy tych liczb różnią się od siebie o więcej niż 52 bity, wówczas w procesie denormalizacji mniejsza liczba zostanie zastąpiona zerem. Tak jest w pierwszym z powyższych wyrażeń.

2.3.7 Utrata cyfr znaczących

Utrata cyfr znaczących (ang. *loss of significance*) to dość częsty błąd przy odejmowaniu dużych, zbliżonych do siebie wartością liczb. Aby zilustrować tę klasę błędów na przykładzie, przyjrzyjmy się dwóm funkcjom zdefiniowanym następująco [35]:

$$f_1(x) = \sqrt{x}\left(\sqrt{x+1} - \sqrt{x}\right), \quad f_2(x) = \frac{\sqrt{x}}{\sqrt{x+1} + \sqrt{x}}. \tag{2.17}$$

Matematycznie są to znowu równoważne wyrażenia, dlatego moglibyśmy się spodziewać, że dadzą one ten sam wynik dla dowolnych x. Jednak po wykonaniu programu[3]

```
#!/usr/bin/octave -qf
% loss.m
% program ilustrujący znoszenie
% się składników przy odejmowaniu
% JS, 16.08.2006
```

[3]W niektórych systemach linuksowych ścieżka do programu GNU Octave może być inna niż na wydruku. Pamiętajmy również o nadaniu plikowi atrybutu wykonywalności.

```
clear
function res=f1(x)
  res=sqrt(x)*(sqrt(x+1)-sqrt(x));
endfunction

function res=f2(x)
  res=sqrt(x)/(sqrt(x+1)+sqrt(x));
endfunction

x=1;
for k=1:15
  printf("x=%15.0f, f1(x)=%20.18f, f2(x)=%20.18f\n",x,f1(x),f2(x));
  x=10*x;
endfor
```

okaże się, że wcale tak nie jest:

```
x=              1, f1(x)=0.414213562373095145, f2(x)=0.414213562373095090
x=             10, f1(x)=0.488088481701514754, f2(x)=0.488088481701515475
x=            100, f1(x)=0.498756211208899458, f2(x)=0.498756211208902733
x=           1000, f1(x)=0.499875062461021868, f2(x)=0.499875062460964859
x=          10000, f1(x)=0.499987500624854420, f2(x)=0.499987500624960890
x=         100000, f1(x)=0.499998750005928860, f2(x)=0.499998750006249937
x=        1000000, f1(x)=0.499999875046341913, f2(x)=0.499999875000062488
x=       10000000, f1(x)=0.499999987401150925, f2(x)=0.499999987500000576
x=      100000000, f1(x)=0.500000005558831617, f2(x)=0.499999998749999952
x=     1000000000, f1(x)=0.500000077997506343, f2(x)=0.499999999874999990
x=    10000000000, f1(x)=0.499999441672116518, f2(x)=0.499999999987500054
x=   100000000000, f1(x)=0.500004449631168080, f2(x)=0.499999999998750000
x=  1000000000000, f1(x)=0.500003807246685028, f2(x)=0.499999999999874989
x= 10000000000000, f1(x)=0.499194546973835973, f2(x)=0.499999999999987510
x=100000000000000, f1(x)=0.502914190292358398, f2(x)=0.499999999999998723
```

Funkcje zwracają różne wartości. Ponadto, podczas gdy $f_2(x)$ dąży jednostajnie do wartości granicznej $0,5$, $f_1(x)$ „skacze" wokół niej.

Aby wyjaśnić to zachowanie, załóżmy, że $x = 10^{15}$. Wówczas:

$$\begin{aligned} \sqrt{x+1} &= 3,162277660168381 \times 10^7 = 31622776,60168381 \\ \sqrt{x} &= 3,162277660168379 \times 10^7 = 31622776.60168379 \end{aligned}$$

Obie liczby mają 16 cyfr znaczących. Największa z tych cyfr jest rzędu 10^8, najmniejsza - 10^{-8}. Dlatego ostatnia cyfra znacząca ich sumy i różnicy będzie na ósmym miejscu po przecinku. Ale

$$\begin{aligned} \sqrt{x+1} + \sqrt{x} &= 63245553.20336761 \\ \sqrt{x+1} - \sqrt{x} &\simeq 0.00000002 \end{aligned}$$

W wyniku odejmowania tych liczb liczba cyfr znaczących zmalała z 16 do tylko jednej, dodatkowo jeszcze obarczonej błędem zaokrąglenia. Dlatego właśnie funkcje $f_1(x)$ i $f_2(x)$ dają numerycznie różne wyniki.

Rozdział 3

Algorytmy numeryczne i ich złożoność

Algorytm to nic innego jak przepis na przekształcenie danych wejściowych do pożądanego wyniku końcowego. To samo zadanie numeryczne można najczęściej rozwiązać kilkoma odmiennymi algorytmami, z których nie wszystkie dają równoważne wyniki[1]. Dlatego w praktyce często będziemy stawać przed koniecznością wyboru algorytmu, który produkuje najmniejszy błąd w danych wyjściowych, wykonuje się najszybciej bądź najoszczędniej gospodaruje dostępnymi zasobami. Określenie przydatności algorytmu nie jest proste i często wymaga użycia skomplikowanego aparatu matematycznego. My jednak ograniczymy się tutaj jedynie do metod uproszczonych, które dają jedynie przybliżone, choć na ogół wystarczające, oszacowanie przydatności algorytmu.

3.1 Algorytm numeryczny

Algorytmem numerycznym nazywać będziemy jednoznacznie ustaloną sekwencję skończenie wielu operacji elementarnych, która określa, jak przekształcić dane wejściowe, aby rozwiązać określony problem. Formalnie można to ująć w następujący sposób:

Definicja Określmy zadanie numeryczne w postaci $y = f(x)$, gdzie $f : \mathbf{R}^n \to \mathbf{R}^m$. Algorytmem numerycznym nazywać będziemy złożenie funkcji

$$f = \phi^{(k)} \circ \phi^{(k-1)} \circ \dots \phi^{(0)}, \tag{3.1}$$

których kolejne wykonywanie na danych wejściowych x doprowadzi do przybliżonego rozwiązania zadania numerycznego,

$$\tilde{y} = \left(\phi^{(k)} \circ \phi^{(k-1)} \circ \dots \phi^{(0)}\right) x. \tag{3.2}$$

Powyższa definicja sprawia wrażenie bardzo formalnej, jednak najczęściej tak zdefiniowany algorytm daje się stosunkowo prosto zaimplementować w popularnych językach programowania.

[1] O tym, że dwa matematycznie równoważne wyrażenia dają różne wyniki numeryczne, przekonaliśmy się już w paragrafie 2.3.7

Przy dyskusji zaniedbywalnych składników i utraty cyfr znaczących (paragrafy 2.3.6 i 2.3.7) przekonaliśmy się, że wyrażenia równoważne matematycznie niekoniecznie dają te same wyniki numeryczne.

Definicja Algorytmy nazywamy *równoważnymi matematycznie*, jeśli dają te same wyniki dla jednakowych danych wejściowych, o ile tylko obliczenia wykonuje się bez zaokrągleń.

Definicja Algorytmy nazywamy *równoważnymi numerycznie*, gdy wyniki dla jednakowych danych wejściowych różnią się tylko o tyle, o ile mogą zmienić się dokładne dane wyjściowe zadania, gdy dane wejściowe zaburzy się o niewiele u, gdzie u jest względnym błędem zaokrąglenia lub ucięcia.

3.2 Uwarunkowanie algorytmu

Rozważmy ponownie zadanie numeryczne polegające na określeniu danych wyjściowych $y \in \mathbf{R}^m$ na podstawie danych wejściowych $x \in \mathbf{R}^n$, jeżeli powiązane są one ze sobą przekształceniem (algorytmem) $f : \mathbf{R}^n \to \mathbf{R}^m$. Na mocy twierdzenia 2.3.3 błąd bezwzględny i-tej składowej danych wyjściowych y wyraża się wzorem

$$\Delta y_i = \sum_{j=1}^{n} \frac{\partial f_i(x)}{\partial x_j} \Delta x_j + O(\Delta x^2). \tag{3.3}$$

Odpowiadający mu błąd względny wynosi

$$\begin{aligned} \frac{\Delta y_i}{y_i} &= \sum_{j=1}^{n} \frac{\partial f_i(x)}{\partial x_j} \frac{\Delta x_j}{y_i} + O\left(\frac{\Delta x^2}{y_i}\right) \qquad (3.4) \\ &= \sum_{j=1}^{n} \frac{\partial f_i(x)}{\partial x_j} \frac{x_j}{f_j(x)} \frac{\Delta x_j}{x_j} + O\left(\frac{\Delta x^2}{y_i}\right) = \sum_{j=1}^{n} k_{ij}(x) \frac{\Delta x_j}{x_j} + O\left(\frac{\Delta x^2}{y_i}\right) \end{aligned}$$

Współczynniki k_{ij} nazywamy względnymi wskaźnikami uwarunkowania. Określają one, jak bardzo błędy w danych wejściowych wpływają na wartość wyniku. Innymi słowy stanowią miarę przydatności algorytmu numerycznego.

Jeżeli $|k_{ij}| > 1$, mamy do czynienia ze wzmocnieniem błędu danych wejściowych. W przeciwnym wypadku błędy są tłumione w trakcie obliczeń (wysoce pożądane!). Jeśli $|k_{ij}| \gg 1$, mówimy, że algorytm jest źle uwarunkowany. Wówczas niewielki błąd w danych wejściowych powoduje duże niedokładności w wyniku.

3.3 Notacja O

Notacja O [37] służy do zapisu szybkości wzrostu. W przypadku algorytmów nazywana jest ona czasami **złożonością teoretyczną**.

Definicja Niech f i g będą ciągami liczb rzeczywistych. Piszemy

$$f(n) = O(g(n)) \tag{3.5}$$

n	10	20	30	40	50	60
$O(n)$	0,00001s	0,00002s	0,00003s	0,00004s	0,00005s	0,00006s
$O(n^2)$	0,0001s	0,0004s	0,0009s	0,0016s	0,0025s	0,0036
$O(n^3)$	0,001s	0,008s	0,027s	0,064s	0,125s	0,216s
$O(2^n)$	0,001s	1,048s	17,9min	12,7dni	35,7lat	366w
$O(3^n)$	0,059s	58min	6,5lat	3855w	$227*10^6$w	$1{,}3*10^{13}$w
$O(n!)$	3,6s	771w	$8{,}4*10^{16}$w	$2{,}6*10^{32}$w	$9{,}6*10^{47}$w	$2{,}6*10^{66}$w

Tabela 3.1: Czasy wykonania algorytmów różnego typu w zależności od rozmiaru danych wejściowych przy założeniu, że dla $n = 1$ każdy z nich wykonuje się 10^{-6}s. Przez w oznaczono wiek (100 lat).

wtedy, gdy istnieje stała dodatnia C taka, że

$$|f(n)| \leqslant C|g(n)| \tag{3.6}$$

dla dostatecznie dużych wartości n.

Powyższa definicja dopuszcza, aby nierówność nie była spełniona dla pewnej liczby małych wartości n. W praktyce $f(n)$ oznacza ciąg, którym właśnie się zajmujemy (np. górne ograniczenie czasu działania algorytmu), a $g(n)$ jest prostym ciągiem o znanej szybkości wzrostu.

Notacja O nie podaje dokładnego czasu wykonania algorytmu. Pozwala jednak odpowiedzieć na pytanie, jak ten czas będzie rósł z rozmiarem danych wejściowych. Notacja ma kilka przydatnych własności:

1. W hierarchii ciągów 1 , $\log_2 n$,..., $\sqrt[4]{n}$, $\sqrt[3]{n}$, $\sqrt{n}$, n, $n \ln n$, $n\sqrt{n}$, n^2, n^3, ..., 2^n, $n!$, n^n każdy z nich jest O od wszystkich ciągów na prawo od niego.
2. Jeśli $f(n) = O(g(n))$ i c jest stałą, to $c * f(n) = O(g(n))$.
3. Jeśli $f(n) = O(g(n))$ i $h(n) = O(g(n))$, to $f(n) + h(n) = O(g(n))$.
4. Jeśli $f(n) = O(a(n))$ i $g(n) = O(b(n))$, to $f(n) * g(n) = O(a(n) * b(n))$.
5. Jeśli $a(n) = O(b(n))$ i $b(n) = O(c(n))$, to $a(n) = O(c(n))$.
6. $O(a(n)) + O(b(n)) = O(\max\{|a(n)|, |b(n)|\})$.
7. $O(a(n)) * O(b(n)) = O(a(n) * b(n))$.

Ich dowód pozostawiam zainteresowanym czytelnikom.

3.4 Złożoność obliczeniowa

Jak używać notacji O do określania szybkości algorytmów? Otóż stwierdzenie, że pewien algorytm jest klasy $O(n!)$ będziemy interpretować w następujący sposób: dla danych wejściowych o rozmiarze x czas wykonania algorytmu jest proporcjonalny do $x!$

W tabeli 3.1 zestawione są czasy wykonania algorytmów różnego typu. Czasy wyliczone zostały przy założeniu, że elementarny czas (dla $n = 1$) wynosi 10^{-6}s. Jak widzimy, szacowanie złożoności obliczeniowej algorytmu może zaoszczędzić

nam dość długiego, przynajmniej w niektórych przypadkach, oczekiwania na efekty jego działania.

Dokładne wyznaczenie złożoności obliczeniowej algorytmu nie zawsze jest zadaniem łatwym. Jednak w praktyce najczęściej wystarczy trzymać się następujących zasad [38]:

1. w analizie programu zwracamy uwagę tylko na najbardziej "czasochłonne" operacje,
2. wybieramy wiersz programu znajdujący się w najgłębiej położonej instrukcji iteracyjnej, następnie zliczamy ilość jego wywołań i na tej podstawie wnioskujemy o klasie algorytmu.

Przykład Przeanalizujmy poniższy fragment programu:

```
while(i<N)
  {
   while(j<=N)
     {
      suma=suma+2;
      j=j+1;
     }
   i=i+2;
  }
```

Wiersz `suma=suma+2` będzie tu wykonywany $\frac{N(N+1)}{2}$ razy. Powyższy fragment jest zatem klasy $O(n^2)$ (rozmiarem danych jest tu wielkość N).

Przykład W paragrafie 2.1.3 przedstawiony został prosty program na obliczanie silni z liczby całkowitej w oparciu o rekurencyjną definicję tej funkcji:

$$\begin{aligned} 0! &= 1, \\ n! &= n * (n-1)!, \quad \text{dla } n \geqslant 1. \end{aligned} \tag{3.7}$$

Odpowiedni fragment programu ma postać (zapomnijmy na chwilę o ograniczeniu związanym ze skończoną ilością liczb możliwych do przedstawienia na komputerze):

```
int silnia (int n)
{
   if(n<=1)
     return 1;
   else
     return n*silnia(n-1);
}
```

Zakładamy, że najbardziej czasochłonną operacją jest tutaj sprawdzenie warunku w instrukcji `if`. Niech czas sprawdzenia jednego warunku wynosi t_c. Wówczas czas wykonania algorytmu $T(n)$ da się zapisać w postaci

$$\begin{aligned} T(0) &= t_c, \\ T(n) &= t_c + T(n-1), \quad \text{dla } n \geqslant 1. \end{aligned} \tag{3.8}$$

Powyższe wzory nie pozwalają jeszcze określić klasy algorytmu. Jeżeli jednak rozpiszemy równania (3.8) jawnie dla wszystkich n,

$$\begin{aligned} T(n) &= t_c + T(n-1), \\ T(n-1) &= t_c + T(n-2), \\ &\vdots \\ T(1) &= t_c + T(0), \\ T(0) &= t_c, \end{aligned}$$

to po dodaniu ich stronami otrzymamy następującą zależność:

$$T(n) = (n+1)t_c. \tag{3.9}$$

Stąd wynika, że algorytm jest klasy $O(n)$.

3.5 Profilowanie programów

Omówione w poprzednim podrozdziale metody przydają się do analizy fragmentów kodu. Aby powiedzieć coś na temat całego programu, w podobny sposób powinniśmy oszacować złożoność wszystkich ważnych części programu. W przypadku skomplikowanych zadań numerycznych taka analiza może być w najlepszym przypadku zajęciem bardzo czasochłonnym, w najgorszym - niemożliwym do wykonania w rozsądnym czasie. Wówczas przydatnym może okazać się profilowanie (ang. *profiling*), znana programistom technika wykorzystywana do optymalizacji kodu. Profilowanie pozwala stwierdzić, ile czasu potrzeba na wykonanie poszczególnych części naszego programu, jakie funkcje są wywoływane oraz do jakich innych te wcześniejsze się odwołują.

Pamiętajmy jednak o tym, że optymalizować powinniśmy działający już kod. Wielu programistów wpada w pułapkę tworzenia programu od początku z myślą, aby był jak najbardziej wydajny. Podejście takie wydłuża czas tworzenia programu dość znacznie, niekiedy nawet w nieskończoność. Dlatego przy optymalizowaniu kodu bądźmy ostrożni, kierując się myślą Williama A. Wulfa [39]:

> ... więcej komputerowych grzechów popełnionych zostało w imię wydajności niż z każdego innego powodu, włączając w to czystą głupotę.

C++ i Fortran

Istnieje wiele narzędzi do profilowania programów pisanych w C++ i Fortranie. W systemach linuksowych popularnym narzędziem jest `gprof` [40], który współpracuje m.in. z zestawem kompilatorów gcc i intelowskim Fortranem. Program działa w linii poleceń, ale istnieją również do niego nakładki graficzne, np. `kprof` [41] przeznaczony dla środowiska KDE.

Program, który chcemy profilować, musi być odpowiednio skompilowany. W przypadku kompilatorów gcc służą do tego flagi `-pg` oraz `-g`:

```
g++ -g  -o program program.cpp -pg
```

Kompilator Intela wymaga natomiast flag `-p -g`:

```
ifort -O -p -g -o program program.f90
```

Po skompilowaniu uruchamiamy program tak, jak to zwykle czynimy. Będzie on działał nieco wolniej niż po normalnej kompilacji, co spowodowane jest operacjami pomiaru czasu i zapisu do pliku.

Informacje na temat działania programu zapisane zostaną do pliku o nazwie `gmon.out`. Jeśli plik taki istniał wcześniej, zostanie nadpisany. Odczytujemy je za pomocą `gprof`. Składnia wywołania programu jest następująca[2]:

```
gprof [opcje] [program_binarny] [gmon.out] [> plik_tekstowy]
```

Za pomocą opcji decydujemy m.in. w jakiej formie `gprof` ma wyświetlić interesujące nas dane z pliku `gmon.out`. Możliwe są

profil płaski (ang. *flat profile*) - podsumowanie, ile razy wykonane zostały poszczególne funkcje i ile czasu procesora zajęły;

graf wywołań (ang. *call graph*) - dla każdej funkcji pokazuje, przez jakie funkcje została ona wywołana i jakie sama wywołuje;

źródło z przypisami (ang. *annotated source*) - wydruk kodu źródłowego, na którym każda linia opatrzona jest etykietą określającą, ile razy została wykonana.

Przykładowy profil płaski programu ma następującą postać:

```
Flat profile:

Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 75.16      0.03     0.03    50000     0.00     0.00  wahadlo::der(double, double*, double*)
 25.05      0.04     0.01    10000     0.00     0.00  wahadlo::rk4_step(double, double*, double*, double)
  0.00      0.04     0.00        2     0.00     0.00  std::ios_base::setf(std::_Ios_Fmtflags)
  0.00      0.04     0.00        2     0.00     0.00  std::ios_base::precision(long)
  0.00      0.04     0.00        2     0.00     0.00  std::operator|=(std::_Ios_Fmtflags&, std::_Ios_Fmtflags)
  0.00      0.04     0.00        2     0.00     0.00  std::operator|(std::_Ios_Fmtflags, std::_Ios_Fmtflags)
  0.00      0.04     0.00        2     0.00     0.00  std::operator|(std::_Ios_Openmode, std::_Ios_Openmode)
  0.00      0.04     0.00        1     0.00     0.00  global constructors keyed to _ZN7wahadloC2Eddddd
  0.00      0.04     0.00        1     0.00     0.00  global constructors keyed to main
  0.00      0.04     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.04     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.04     0.00        1     0.00    34.07  wahadlo::rk4()
  0.00      0.04     0.00        1     0.00     6.01  wahadlo::euler()
  0.00      0.04     0.00        1     0.00     0.00  wahadlo::wahadlo(double, double, double, double, double)
  0.00      0.04     0.00        1     0.00     0.00  wahadlo::~wahadlo()
```

Funkcje są sortowane według malejącego czasu wykonania, następnie malejącej liczby wywołań, a potem alfabetycznie według nazw. Znaczenie poszczególnych pól można sprawdzić w dokumentacji programu `gprof` [40]. Najważniejsza dla nas informacja jest taka, że 75% czasu program spędza na wykonywaniu funkcji `wahadlo::der`, gdyby więc zaszła potrzeba jego przyspieszenia, należałoby zacząć od zoptymalizowania tej funkcji.

Python

W bibliotece standardowej Pythona znajdują się trzy moduły do profilowania programów: `profile`, `hotshot` i `timeit` [42]. Przy tym dwa pierwsze potrafią profilować całe skrypty, natomiast ostatni służy do pomiaru czasu wykonywania małych wycinków kodu.

Sposób użycia modułu `hotshot` demonstruje poniższy skrypt:

[2]Argumenty w nawiasach są opcjonalne.

```
#!/usr/bin/python
#silnia.py  - przyklad zastosowania
#profilera w Pythonie
#JS, 18.08.2006

import hotshot,hotshot.stats

def silnia(x):
    if x<=1:
        return 1
    return x*silnia(x-1)

def main():
    print silnia(100)

profiler = hotshot.Profile("logfile.dat")
profiler.run("main()")
profiler.close()

stats = hotshot.stats.load("logfile.dat")
stats.print_stats(20)
```

Po wykonaniu tego skryptu na ekranie zobaczymy, oprócz wyniku działania funkcji `silnia`, profil programu:

```
         102 function calls (3 primitive calls) in 0.001 CPU seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.001    0.001 <string>:1(?)
        0    0.000             0.000          profile:0(profiler)
    100/1    0.001    0.000    0.001    0.001 silnia.py:8(silnia)
        1    0.001    0.001    0.001    0.001 silnia.py:13(main)
```

GNU Octave

W porównaniu z C++, Fortranem czy Pythonem profilowanie skryptów w GNU Octave jest trudniejsze, ponieważ nie dostajemy do dyspozycji gotowych narzędzi do analizowania działania programu. Jednak dzięki wbudowanym funkcjom do pomiaru czasu, np. `cputime` [43], możemy na własną rękę mierzyć czas wykonania poszczególnych funkcji:

```
#!/usr/bin/octave -qf
%wydajnosc.m - porownanie wydajnosci
%wbudowanego i 'recznego' (w stylu C)
%mnozenia macierzowego
%JS, 30.10.2003

A=rand(100);
b=rand(100,1);

t=cputime;
v=A*b;
t1=cputime-t;
printf("wbudowane mnozenie: %f\n",t1);
```

```
w=zeros(100,1);
t=cputime;
for n=1:100,
  for m=1:100
    w(n)=w(n)+A(n,m)*b(m);
  end
end
t2=cputime-t;
printf("mnozenie 'reczne': %f\n",t2);
```

Po wykonaniu tego skryptu okaże się, że tak charakterystyczne dla języków C++ i Fortran pętle w GNU Octave nie zawsze są najlepszym wyborem:

```
wbudowane mnozenie: 0.004000
mnozenie 'reczne': 0.452029
```

Jeżeli oprócz samego GNU Octave zainstalowaliśmy również pakiet `octave-forge` [44], do analizowania szybkości wykonania algorytmu w zależności od rozmiaru danych wejściowych możemy użyć funkcji `speed()`. Wywołujemy ją w następujący sposób:

```
speed(f,init,nmax,f2,tol,err)
```

gdzie `f` to wyrażenie do przetestowania, a `nmax` oznacza maksymalną liczbę testów. Jeżeli `f` potrzebuje pewnych parametrów na wejściu, określamy je za pomocą `init`. Możliwe jest porównanie ze sobą dwóch wyrażeń - to drugie podajemy jako argument `f2`. Przy tym, jeśli `tol=inf`, porównywane są tylko czasy ich wykonania, w przeciwnym razie również zwracane przez nie wartości (za pomocą `assert(v1,v2,tol,err)`).

Aby zobaczyć `speed()` w akcji, zmodyfikujmy nieco przykład przedstawiony powyżej:

```
#!/usr/bin/octave -qf
%speed_wydajnosc.m - porownanie wydajnosci
%mnozenia macierzowego za pomoca funkcji
%speed()
%JS, 30.10.2003

disp("Wykorzystanie funkcji speed()");

function [A,b]=init(x)
  A=rand(x);
  b=rand(x,1);
endfunction

function w = oct(A,b,x)
  w=zeros(x,1);
  w=A*b;
endfunction

function w = mat(A,b,x)
  w=zeros(x,1);
  for i=1:x,
    for j=1:x
      w(i)=w(i)+A(i,j)*b(j);
```

```
    end
  end
endfunction

speed('oct(A,b,x);','x=k+1,[A,b]=init(x);',
                100,'mat(A,b,x);',inf);

pause
```

Powyższy kod wymaga kilka słów komentarza. W obu funkcjach wykonujących mnożenie macierzowe (`oct` i `mat`) tworzony jest najpierw wektor `w`, chociaż tak naprawdę ta operacja jest niezbędna tylko do poprawnego działania funkcji `mat` (dlaczego?). Alokacja wektora i wypełnienie go zerami kosztuje jednak trochę czasu, dlatego dodałem ją również do drugiej funkcji, aby być pewnym, że obie funkcję różnią się tylko i wyłącznie sposobem mnożenia wektorów. Ponadto, w drugim argumencie funkcji `speed` pojawiała się zmienna `k`. To zmienna, w której przechowywany jest numer testu wykonywanego na wyrażeniach `f` i `f2`.

Po wykonaniu skryptu w konsoli tekstowej pojawi się szereg komunikatów, z których najważniejszy to

```
mean runtime ratio of mat(A,b,x); / oct(A,b,x); : 260.927
```

co oznacza, że mnożenie za pomocą pętli odbywa się około 260 razy wolniej od wbudowanego mnożenia macierzowego. Funkcja `speed()` przedstawia również zebrane informacje w postaci wykresów. Aby nie zniknęły one błyskawicznie z ekranu po wykonaniu skryptu, zakończyliśmy go poleceniem `pause`.

Rozdział 4

Układy równań liniowych

Jednym z podstawowych zadań algebry liniowej jest rozwiązanie układu równań

$$\mathbf{A}\vec{x} = \vec{b}, \tag{4.1}$$

gdzie $\mathbf{A}$ jest macierzą $m \times n$ znanych współczynników, $\vec{b}$ - wektorem m danych liczb, a $\vec{x}$ - wektorem n niewiadomych. Układ ten może mieć nieskończenie wiele rozwiązań, jedno rozwiązanie lub nie mieć ich wcale. Chociaż warunki istnienia rozwiązań układu (4.1) są znane [45] i istnieją też gotowe wzory na wyliczenie $\vec{x}$ w wielu przypadkach, to jego numeryczne rozwiązanie może się okazać dość trudnym zadaniem.

Dostępne metody numerycznego rozwiązywania układów równań liniowych możemy podzielić na dwie grupy. Metody dokładne (bezpośrednie) przy braku błędów zaokrągleń dają dokładne rozwiązanie po skończonej liczbie przekształceń układu wyjściowego. Metody iteracyjne natomiast pozwalają na wyznaczenie zbieżnego ciągu rozwiązań przybliżonych. Te pierwsze są prawie zawsze najbardziej efektywne w przypadku układów o macierzach pełnych. Ich wadą jest duże obciążenie pamięci oraz (w pewnych przypadkach) niestabilność ze względu na błędy zaokrągleń. Te drugie stosują się do układów, których macierze są rzadkie (tzn. większość elementów tych macierzy jest równa zero).

Jako fizycy często mamy do czynienia z zadaniami, które sprowadzają się do rozwiązania odpowiedniego układu równań liniowych. Ponadto w analizie numerycznej wiele algorytmów opartych jest o takie układy. Dlatego niniejszy rozdział będzie jednym z dłuższych i z pewnością najważniejszym rozdziałem tego skryptu.

4.1 Pojęcia podstawowe

Rozpoczniemy od przypomnienia kilku pojęć z algebry liniowej, które okażą się przydatne w dalszych rozważaniach (patrz np. [45–48]).

4.1.1 Normy

Operując na wektorach i macierzach często będziemy posługiwać się pojęciem normy [47].

Definicja Normą w przestrzeni $\mathbf{R}^n$ nazywamy funkcję

$$\|\cdot\| : \mathbf{R}^n \to \langle 0, +\infty) \tag{4.2}$$

o następujących własnościach:

1. $\|\vec{x}\| \geqslant 0$ dla każdego $x \in \mathbf{R}^n$,
2. $\|\alpha\vec{x}\| = |\alpha|\|\vec{x}\|$ dla każdego $\alpha \in \mathbf{R}$ i każdego $\vec{x} \in \mathbf{R}^n$,
3. $\|\vec{x}_1 - \vec{x}_2\| \leqslant \|\vec{x}_1\| + \|\vec{x}_2\|$ dla każdej pary $\vec{x}_1, \vec{x}_2 \in \mathbf{R}^n$ (nierówność trójkąta),
4. $\|\vec{x}\| = 0 \Leftrightarrow \vec{x} = 0$.

W przestrzeni $\mathbf{R}^n$, której elementami są wektory $\vec{x} = [x_1, x_2, \ldots, x_n]^{\mathrm{T}}$, można wprowadzić wiele norm. Te najczęściej stosowane w obliczeniach numerycznych to

$$\begin{aligned} \|\vec{x}\|_1 &= |x_1| + |x_2| + \cdots + |x_n|, \\ \|\vec{x}\|_2 &= \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}, \\ \|\vec{x}\|_\infty &= \max\left\{|x_1|, |x_2|, \ldots, |x_n|\right\}. \end{aligned} \tag{4.3}$$

Powyższe normy są równoważne w tym sensie, że jeśli ciąg wektorów $\vec{x}_1$, $\vec{x}_2$, $\vec{x}_3$, ... dąży do wektora zerowego w jednej normie, to zbieżność zachodzi również w dowolnej innej.

Obok normy wektora posługiwać się również będziemy normą macierzy.

Definicja Normą macierzy $\mathbf{A}$ nazywamy

$$\|\mathbf{A}\|_{pq} = \max_{\vec{x} \in \mathbf{R}^n, \vec{x} \neq 0} \frac{\|\mathbf{A}\vec{x}\|_q}{\|\vec{x}\|_p}. \tag{4.4}$$

Przy tym, jeżeli $p = q$, będziemy pisać $\|\mathbf{A}\|_p$.

Najczęściej stosowane normy macierzy to:

$$\begin{aligned} \|\mathbf{A}\|_1 &= \max_{j=1,\ldots,n} \sum_{i=1}^{n} |a_{ij}|, \\ \|\mathbf{A}\|_2 &= \sqrt{\lambda_{max}}, \\ \|\mathbf{A}\|_\infty &= \max_{i=1,\ldots,n} \sum_{j=1}^{n} |a_{ij}|, \end{aligned} \tag{4.5}$$

gdzie λ_{max} to największa wartość własna macierzy $\mathbf{A}^{\mathrm{T}}\mathbf{A}$. Ponieważ wyliczenie wartości własnej jest na ogół zadaniem trudnym, zamiast $\|\mathbf{A}\|_2$ używa się dość często łatwiejszej do wyznaczenia normy euklidesowej.

Definicja Euklidesową normą macierzy (normą Schura, normą Frobeniusza) nazywamy

$$\|\mathbf{A}\|_E = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2}. \tag{4.6}$$

Norma ta spełnia warunek zgodności z $\|\cdot\|_2$, tzn.:

$$\|\mathbf{A}\vec{x}\|_2 \leqslant \|\mathbf{A}\|_E \|\vec{x}\|_2. \tag{4.7}$$

4.1.2 Wyznaczniki

Każdej macierzy kwadratowej

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{21} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \tag{4.8}$$

przyporządkować można pewną liczbę, tzw. wyznacznik.

Definicja Wyznacznikiem macierzy $\mathbf{A}$ nazywamy liczbę

$$\det \mathbf{A} = \sum_f (-1)^{I_f} a_{1\alpha_1} a_{2\alpha_2} \dots a_{n\alpha_n}, \tag{4.9}$$

gdzie $\sum_f$ oznacza sumowanie po wszystkich permutacjach liczb naturalnych $1, 2, \dots, n$, a I_f to liczba inwersji w permutacji f.

Wzór (4.9) ma raczej niewielkie znaczenie praktyczne. Dlatego możemy próbować policzyć wyznacznik z rozwinięcia Laplace'a wzdłuż i-tego wiersza lub j-tej kolumny,

$$\begin{aligned} \det \mathbf{A} &= \sum_{j=1}^{n} a_{ij} A_{ij} \\ \det \mathbf{A} &= \sum_{j=1}^{n} a_{jk} A_{jk} \end{aligned} \tag{4.10}$$

gdzie A_{ij} to dopełnienie algebraiczne elementu a_{ij} macierzy $\mathbf{A}$. Jednak pojawia się tu pewien problem. Otóż wyliczenie wyznacznika na podstawie dowolnego ze wzorów (4.10) wymaga $n!$ mnożeń, dlatego można je stosować dla bardzo małych n. Przy dużych rozmiarach macierzy ich zastosowanie jest raczej wykluczone (patrz tabela 3.1). Na szczęście istnieją bardziej wydajne sposoby obliczania wyznaczników macierzy. Poznamy je w dalszej części tego rozdziału.

4.1.3 Macierze trójkątne

Macierzą trójkątną dolną (lewą) nazywać będziemy macierz

$$\mathbf{L} = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix}. \tag{4.11}$$

Podobnie macierz trójkątną górną (prawą) zdefiniujemy jako

$$\mathbf{R} = \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & r_{21} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{nn} \end{pmatrix}. \tag{4.12}$$

Sumy, iloczyny i odwrotności macierzy trójkątnych tego samego rodzaju są znowu macierzami trójkątnymi. Bardzo łatwo jest też wyliczyć wyznaczniki takich macierzy. Z rozwinięcia Laplace'a wynika mianowicie, że

$$\begin{aligned} \det \mathbf{L} &= l_{11} l_{22} \dots l_{nn}, \\ \det \mathbf{R} &= r_{11} r_{22} \dots r_{nn}. \end{aligned} \tag{4.13}$$

A zatem wyznaczniki są po prostu równe iloczynom elementów leżących na głównych przekątnych.

4.1.4 Układy równań liniowych

W przypadku ogólnym układ (4.1) ma postać

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2, \qquad (4.14) \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n &= b_m, \qquad (4.15) \end{aligned}$$

przy czym m i n są dowolne. Rozważmy macierz rozszerzoną układu, zdefiniowaną jako

$$\mathbf{D} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{21} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{pmatrix}. \tag{4.16}$$

Zachodzi następujące twierdzenie

Twierdzenie 4.1.1 ***(Capellego)*** *Warunkiem koniecznym i wystarczającym rozwiązywalności dowolnego układu równań liniowych jest, aby rząd r macierzy $\mathbf{A}$ układu był równy rzędowi macierzy rozszerzonej $\mathbf{D}$.*

Jeśli powyższy warunek jest spełniony, to układ (4.14) posiada rozwiązanie zależne od $n - r$ parametrów, gdzie n to liczba niewiadomych. W przypadku $n = r$ istnieje jednoznaczne rozwiązanie.

Z kursu algebry liniowej powinniśmy pamiętać, że rozwiązanie układu (4.14) w przypadku nieosobliwej macierzy kwadratowej $\mathbf{A}$ można wyznaczyć ze wzorów Cramera,

$$x_k = \frac{\det \mathbf{A_k}}{\det \mathbf{A}}, \quad k = 1, 2, \dots, n, \tag{4.17}$$

gdzie macierz $\mathbf{A}_k$ powstaje z macierzy $\mathbf{A}$ przez zastąpienie k–tej kolumny przez wektor b. Niestety, metoda ta wymaga dużego nakładu obliczeń (wyznaczniki) i może prowadzić do dużych błędów w rozwiązaniu. Dlatego rzadko się ją stosuje w obliczeniach numerycznych.

4.2 Metody dokładne dla układów określonych $(m = n)$

4.2.1 Analiza zaburzeń

Zanim zapoznamy się z metodami rozwiązywania układów równań liniowych, zastanówmy się, jakich błędów możemy spodziewać się w wyniku.

Jak wiadomo, wynik niedokładnego działania w arytmetyce zmiennopozycyjnej możemy przedstawić jako wynik działania nieobarczonego błędami wykonanego na zaburzonych argumentach[1] [36, 50]. Jeżeli posłużymy się tą techniką i w równaniu (4.1) zastąpimy macierz $\mathbf{A}$ macierzą zaburzoną $\mathbf{A} + \delta\mathbf{A}$, a wektor wyrazów wolnych $\vec{b}$ wektorem zaburzonym $\vec{b} + \delta\vec{b}$, to zamiast rozwiązania $\vec{x}$ układu (4.1) znajdziemy rozwiązanie $\vec{x} + \delta\vec{x}$ układu

$$(\mathbf{A} + \delta\mathbf{A})(\vec{x} + \delta\vec{x}) = \vec{b} + \delta\vec{b}. \tag{4.18}$$

Wielkość $\delta\vec{x}$ zależeć przy tym będzie nie tylko od zaburzeń danych wejściowych $\delta\mathbf{A}$ i $\delta\vec{b}$, ale również od uwarunkowania układu. Poniżej zostanie wyjaśnione, co należy rozumieć pod tym pojęciem.

$\delta\mathbf{A} = 0$ **i** $\delta\vec{b} \neq 0$

Analizę błędów zacznijmy od przypadku, w którym tylko wektor wyrazów wolnych jest zaburzony. Z równania (4.18) otrzymamy

$$\begin{aligned} \mathbf{A}(\vec{x} + \delta\vec{x}) &= \vec{b} + \delta\vec{b}, \\ \mathbf{A}\vec{x} + \mathbf{A}\delta\vec{x} &= \vec{b} + \delta\vec{b}, \\ \mathbf{A}\delta\vec{x} &= \delta\vec{b}, \\ \delta\vec{x} &= \mathbf{A}^{-1}\delta\vec{b}, \end{aligned} \tag{4.19}$$

i dla dowolnych norm norm wektorów $\delta\vec{b}$ i $\delta\vec{x}$ oraz indukowanej przez nie normy macierzy $\mathbf{A}^{-1}$ możemy napisać

$$\|\delta\vec{x}\|_p \leqslant \|\mathbf{A}^{-1}\|_{qp}\|\delta\vec{b}\|_q. \tag{4.20}$$

Jeśli $\vec{x} \neq 0$, to

$$\begin{aligned} \frac{\|\delta\vec{x}\|_p}{\|\vec{x}\|_p} &\leqslant \frac{\|\mathbf{A}^{-1}\|_{qp}\|\delta\vec{b}\|_q}{\|\vec{x}\|_p} = \frac{\|\mathbf{A}^{-1}\|_{qp}\|\vec{b}\|_q}{\|\vec{x}\|_p}\frac{\|\delta\vec{b}\|_q}{\|\vec{b}\|_q} \\ &= \frac{\|\mathbf{A}^{-1}\|_{qp}\|\mathbf{A}\vec{x}\|_q}{\|\vec{x}\|_p}\frac{\|\delta\vec{b}\|_q}{\|\vec{b}\|_q} = \|\mathbf{A}^{-1}\|_{qp}\|\mathbf{A}\|_{pq}\frac{\|\delta\vec{b}\|_q}{\|\vec{b}\|_q} \\ &= K_{pq}\frac{\|\delta\vec{b}\|_q}{\|\vec{b}\|_q}, \end{aligned} \tag{4.21}$$

gdzie

$$K_{pq} = \|\mathbf{A}^{-1}\|_{qp}\|\mathbf{A}\|_{pq} \tag{4.22}$$

to wskaźnik uwarunkowania zadania. Dla danego układu wskaźnik nie jest określony jednoznacznie, jego wartość zależy od wyboru norm. Jednak niezależnie od tego, na które normy się zdecydujemy, o zadaniu będziemy mówić, że jest dobrze uwarunkowane, jeżeli wskaźnik będzie miał wartość bliską jedności. Im wskaźnik jest większy, tym zadanie gorzej uwarunkowane, ponieważ na mocy (4.21) nawet niewielkie zaburzenie w wektorze wyrazów wolnych jest wówczas wzmacniane i może spowodować duży błąd w wyniku.

[1] W literaturze często mówi się w tym przypadku o interpretacji Wilkinsona [49].

Przykład Rozważmy układ

$$\mathbf{A} = \begin{pmatrix} 1,2969 & 0,8648 \\ 0,2161 & 0,1441 \end{pmatrix}, \quad \mathbf{A}^{-1} = 10^8 \begin{pmatrix} 0,1441 & -0,8648 \\ -0,2161 & 1,2969 \end{pmatrix}. \tag{4.23}$$

Mamy

$$\|\mathbf{A}\|_\infty = 2,1617, \quad \|\mathbf{A}^{-1}\|_\infty = 1,513 * 10^8, \tag{4.24}$$

oraz

$$K = \|\mathbf{A}^{-1}\|_\infty \|\mathbf{A}\|_\infty \approx 3,3 * 10^8. \tag{4.25}$$

Stąd wynika, że przy rozwiązaniu układu w najgorszym wypadku możemy utracić 8 miejsc istotnych dokładności (złe uwarunkowanie).

W przypadku dużych macierzy wyliczenie wskaźnika (4.22) może być czasochłonne. Dlatego w praktyce często jako kryterium uwarunkowania stosuje się porównanie wartości wyznacznika macierzy $\mathbf{A}$ z jej elementami. Jeżeli jest on dużo mniejszy niż najmniejszy element macierzy, wówczas zadanie jest na ogół źle uwarunkowane.

$\delta\mathbf{A} \neq 0$ **i** $\delta\vec{b} = 0$

Z równania macierzowego

$$(\mathbf{A} + \delta\mathbf{A})(\vec{x} + \delta\vec{x}) = \vec{b} \tag{4.26}$$

wynika

$$\delta\vec{x} = -\mathbf{A}^{-1}\delta\mathbf{A}(\vec{x} + \delta\vec{x}). \tag{4.27}$$

Wówczas

$$\|\delta\vec{x}\| \leqslant \|\mathbf{A}^{-1}\|\|\delta\mathbf{A}\|\|\vec{x} + \delta\vec{x}\|, \tag{4.28}$$

czyli

$$\frac{\|\delta\vec{x}\|}{\|\vec{x} + \delta\vec{x}\|} \leqslant \|\mathbf{A}^{-1}\|\|\delta\mathbf{A}\| = K\frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|}. \tag{4.29}$$

Dla uproszczenia notacji opuściliśmy indeksy norm. Podobnie, jak w poprzednio, również w tym przypadku wielkość końcowego błędu zależy od wskaźnika uwarunkowania i jest wzmacniana dla układów, dla których wskaźnik ten jest duży.

$\delta\mathbf{A} \neq 0$ **i** $\delta\vec{b} \neq 0$

Nawet, jeżeli $\mathbf{A}$ i $\vec{b}$ są znane dokładnie, zwykle nie będą miały dokładnej reprezentacji maszynowej. Dlatego najczęściej będziemy mieli do czynienia z sytuacją, w której $\delta\mathbf{A} \neq 0$ i $\delta\vec{b} \neq 0$.

Załóżmy, że zaburzenie $\delta\mathbf{A}$ jest na tyle małe, że macierz $\mathbf{A} + \delta\mathbf{A}$ pozostaje nieosobliwa. Wówczas z równania

$$(\mathbf{A} + \delta\mathbf{A})(\vec{x} + \delta\vec{x}) = \vec{b} + \delta\vec{b}. \tag{4.30}$$

otrzymamy

$$\delta\vec{x} = -\mathbf{A}^{-1}\left(\delta\vec{b} - \delta\mathbf{A}\vec{x} - \delta\mathbf{A}\delta\vec{x}\right). \tag{4.31}$$

Stąd

$$\|\delta\vec{x}\| \leqslant \|\mathbf{A}^{-1}\| \left(\|\delta\vec{b}\| + \|\delta\mathbf{A}\|\|\vec{x}\| + \|\delta\mathbf{A}\|\|\delta\vec{x}\| \right), \tag{4.32}$$

czyli

$$\|\delta\vec{x}\| \leqslant \frac{1}{1 - \|\mathbf{A}^{-1}\|\|\delta\mathbf{A}\|} \|\mathbf{A}^{-1}\| \left(\|\delta\vec{b}\| + \|\delta\mathbf{A}\|\|\vec{x}\| \right). \tag{4.33}$$

Z równości $\mathbf{A}\vec{x} = \vec{b}$ wynika

$$\frac{\|\vec{b}\|}{\|\vec{x}\|\|\mathbf{A}\|} \leqslant 1. \tag{4.34}$$

Dlatego ostatecznie otrzymamy

$$\begin{aligned} \frac{\|\delta\vec{x}\|}{\|\vec{x}\|} &\leqslant \frac{1}{1 - \|\mathbf{A}^{-1}\|\|\delta\mathbf{A}\|} \|\mathbf{A}^{-1}\|\|\mathbf{A}\| \left(\frac{\|\delta\vec{b}\|}{\|\vec{b}\|} \frac{\|\vec{b}\|}{\|\vec{x}\|\|\mathbf{A}\|} + \frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|} \right) \\ &\leqslant \frac{K}{1 - \|\mathbf{A}^{-1}\|\|\delta\mathbf{A}\|} \left(\frac{\|\delta\vec{b}\|}{\|\vec{b}\|} + \frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|} \right). \end{aligned} \tag{4.35}$$

4.2.2 Układy z macierzami trójkątnymi

Jeżeli macierz układu (4.1) jest macierzą trójkątną, rozwiązuje się go szczególnie łatwo. Dla ustalenia uwagi przyjmijmy, że $\mathbf{A}$ jest macierzą trójkątną górną. Aby istniało jednoznaczne rozwiązanie, macierz musi być nieosobliwa. Innymi słowy, wszystkie elementy na głównej przekątnej macierzy $\mathbf{A}$ muszą być różne od zera. Nasz układ będzie miał postać

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \qquad (4.36) \\ &\vdots \\ a_{nn}x_n &= b_n \qquad (4.37) \\ & \qquad (4.38) \end{aligned}$$

Zwróćmy uwagę, że składową x_n wektora niewiadomych $\vec{x}$ obliczymy natychmiast z ostatniego równania powyższego układu. Podstawiając wynik do przedostatniego równania wyznaczymy x_{n-1}. Procedurę kontynuujemy aż do wyliczenia x_1. Rozwiązanie zapisze się wzorem

$$\begin{aligned} x_n &= \frac{b_n}{a_{nn}}, \qquad (4.39) \\ x_i &= \frac{b_i - \sum_{k=i+1}^{n} a_{ik}x_k}{a_{ii}}, \ i = n-1, n-2, \ldots, 1. \end{aligned}$$

Ponieważ obliczenia zaczynamy od ostatniej składowej wektora niewiadomych, metoda ta nazywana jest podstawianiem w tył. Do znalezienia $\vec{x}$ musimy wykonać

$$M = \frac{1}{2}n^2 + \frac{1}{2}n$$

mnożeń i dzieleń oraz

$$D = \frac{1}{2}n^2 - \frac{1}{2}n$$

dodawań. Koszt obliczeń jest więc niewiele większy od kosztu mnożenia wektora przez macierz trójkątną!

W podobny sposób znajdziemy rozwiązanie układu z macierzą trójkątną dolną, tym razem wykonując podstawianie w przód:

$$\begin{aligned} x_1 &= \frac{b_1}{a_{11}}, \\ x_i &= \frac{b_i - \sum_{k=1}^{i-1} a_{ik}x_k}{a_{ii}},\ i = 2, 3, \ldots, n. \end{aligned} \tag{4.40}$$

Koszt obliczeń jest oczywiście taki sam jak poprzednio.

Wiele metod numerycznego rozwiązywania układów równań z dowolnymi macierzami polega na sprowadzeniu układu wyjściowego do postaci trójkątnej, a następnie zastosowaniu jednego ze wzorów (4.39)-(4.40).

4.2.3 Eliminacja Gaussa

Jedną z takich właśnie metod jest eliminacja Gaussa. Chociaż nazwana tak na cześć Carla Friedricha Gaussa, po raz pierwszy zaprezentowana została dużo wcześniej, bo już około 150 roku p.n.e w słynnym chińskim podręczniku matematyki „Dziewięć rozdziałów sztuki matematycznej" [51].

Dla ułatwienia założymy, że macierz układu jest wymiaru 3×3, tzn.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1, \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2, \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3. \end{aligned} \tag{4.41}$$

(4.42)

Naszym celem jest sprowadzenie tego układu do postaci trójkątnej. Odejmując od drugiego wiersza układu (4.41) pierwszy pomnożony przez a_{21}/a_{11}, a od trzeciego pierwszy pomnożony przez a_{31}/a_{11} otrzymamy

$$\begin{aligned} a_{11}^{(0)}x_1 + a_{12}^{(0)}x_2 + a_{13}^{(0)}x_3 &= b_1^{(0)}, \\ a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 &= b_2^{(1)}, \\ a_{32}^{(1)}x_2 + a_{33}^{(1)}x_3 &= b_3^{(1)}, \end{aligned} \tag{4.43}$$

(4.44)

gdzie

$$a_{ij}^{(0)} = a_{ij}, \quad b_i^{(0)} = b_i, \quad i, j = 1, 2, 3 \tag{4.45}$$

oraz

$$a_{ij}^{(1)} = a_{ij}^{(0)} - \frac{a_{i1}^{(0)}}{a_{11}^{(0)}}a_{1j}^{(0)}, \quad b_i^{(1)} = b_i^{(0)} - \frac{a_{i1}^{(0)}}{a_{11}^{(0)}}b_1^{(0)}, \quad i, j = 2, 3. \tag{4.46}$$

Aby wyrugować zmienną x_2 z trzeciego równania w układzie (4.43), odejmujemy od niego drugie pomnożone przez $a_{32}^{(1)}/a_{22}^{(1)}$:

$$\begin{aligned} a_{11}^{(0)}x_1 + a_{12}^{(0)}x_2 + a_{13}^{(0)}x_3 &= b_1^{(0)}, \\ a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 &= b_2^{(1)}, \\ a_{33}^{(2)}x_3 &= b_3^{(2)}, \end{aligned} \tag{4.47}$$

(4.48)

gdzie

$$a_{ij}^{(2)} = a_{ij}^{(1)} - \frac{a_{i2}^{(1)}}{a_{22}^{(1)}} a_{2j}^{(1)}, \quad b_i^{(2)} = b_i^{(1)} - \frac{a_{i2}^{(1)}}{a_{22}^{(1)}} b_2^{(1)}, \quad i,j = 3. \tag{4.49}$$

Wzory na współczynniki macierzy i wyrazy wolne w każdym kroku eliminacji Gaussa łatwo jest uogólnić na przypadek macierzy dowolnego rozmiaru:

$$\begin{aligned} a_{ij}^{(k)} &= a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} a_{kj}^{(k-1)}, \quad i,j = k+1, k+2, \ldots, n, \\ b_i^{(k)} &= b_i^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} b_k^{(k-1)}, \quad i,j = k+1, k+2, \ldots, n. \end{aligned} \tag{4.50}$$

Tym sposobem rzeczywiście otrzymaliśmy układ trójkątny, który można teraz rozwiązać podstawianiem wstecz (4.39),

$$x_i = \frac{b_i^{(i-1)} - \sum_{j=i+1}^{n} a_{ij}^{(i-1)} x_j}{a_{ii}^{(i-1)}}, \quad i = n, n-1, \ldots, 1. \tag{4.51}$$

Całkowity nakład obliczeń w eliminacji Gaussa to

$$M = \frac{1}{3}n^3 + n^2 - \frac{1}{3}$$

mnożeń i dzieleń, oraz

$$D = \frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n$$

dodawań, z czego większa część przypada na sprowadzenie układu do postaci trójkątnej. Liczba operacji jest bez porównania mniejsza niż w przypadku wzorów Cramera.

4.2.4 Wybór elementu podstawowego

Eliminacja Gaussa w formie przedstawionej powyżej nie jest niezawodna.

Przykład Rozważmy układ

$$\begin{pmatrix} 0 & 2 & 2 \\ 3 & 3 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}. \tag{4.52}$$

Jego macierz jest nieosobliwa, a zatem istnieje jednoznaczne rozwiązanie. Mimo to eliminacja Gaussa zawodzi już w pierwszym kroku, ponieważ algorytm wymaga dzielenia przez a_{11}, które tutaj jest równe 0.

Dlatego najczęściej stosuje się pewną modyfikację metody Gaussa, zwaną częściowym wyborem elementu podstawowego.

Definicja Elementem podstawowym nazywamy ten element macierzy $\mathbf{A}$, za pomocą którego dokonujemy eliminacji zmiennej z dalszych równań.

Do tej pory elementami podstawowymi były jedynie elementy wyjściowej macierzy **A** stojące na głównej diagonali. Zauważmy jednak, że rozwiązanie równania nie zmieni się, jeżeli zamienimy kolejność wierszy w układzie równań. Własność tę można wykorzystać, aby uniknąć problemów związanych z dzieleniem przez zero.

Rozważmy ponownie układ (4.52), tym razem biorąc pod uwagę jego macierz rozszerzoną. Dodatkowo notować będziemy położenie wierszy w każdym kroku:

$$\left(\begin{array}{ccc|c} 0 & 2 & 2 & 1 \\ 3 & 3 & 0 & 3 \\ 1 & 0 & 1 & 2 \end{array}\right) \begin{array}{c} : \\ : \\ : \end{array} \begin{array}{c} w1 \\ w2 \\ w3 \end{array} \tag{4.53}$$

Wiemy już, że $a_{11} = 0$ nie może być elementem podstawowym. Dlatego skorzystamy ze wspomnianej powyżej własności układów równań i zamienimy ze sobą wiersze w macierzy układu, tak aby nowy element diagonalny w jej pierwszym wierszu był różny od zera:

$$\left(\begin{array}{ccc|c} 3 & 3 & 0 & 3 \\ 0 & 2 & 2 & 1 \\ 1 & 0 & 1 & 2 \end{array}\right) \begin{array}{c} : \\ : \\ : \end{array} \begin{array}{c} w1^{(1)} \\ w2^{(1)} \\ w3^{(1)} \end{array} \tag{4.54}$$

Teoretycznie jest wszystko jedno, czy pierwszy wiersz zamienimy z drugim, czy z trzecim. Jednak ze względu na błędy zaokrągleń w i-tym kroku eliminacji Gaussa powinniśmy się kierować wartościami elementów w i-tej kolumnie i wybierać wiersz, który ma największy element [48].

Po zamianie wierszy możemy na macierzy (4.54) wykonać pierwszy krok eliminacji Gaussa:

$$\begin{array}{lcl} w1^{(1)} & \rightarrow & \\ w2^{(1)} - (a_{21}^{(1)}/a_{11}^{(1)}) \times w1^{(1)} & \rightarrow & \\ w3^{(1)} - (a_{31}^{(1)}/a_{11}^{(1)}) \times w1^{(1)} & \rightarrow & \end{array} \left(\begin{array}{ccc|c} 3 & 3 & 0 & 3 \\ 0 & 2 & 2 & 1 \\ 0 & -1 & 1 & 1 \end{array}\right) \begin{array}{c} : \\ : \\ : \end{array} \begin{array}{c} w1^{(2)} \\ w2^{(2)} \\ w3^{(2)} \end{array} \tag{4.55}$$

W kolejnym kroku nie musimy zamieniać wierszy ze sobą:

$$\begin{array}{lcl} w1^{(2)} & \rightarrow & \\ w2^{(2)} & \rightarrow & \\ w3^{(2)} - (a_{32}^{(2)}/a_{22}^{(2)}) \times w2^{(2)} & \rightarrow & \end{array} \left(\begin{array}{ccc|c} 3 & 3 & 0 & 3 \\ 0 & 2 & 2 & 1 \\ 0 & 0 & 2 & 3/2 \end{array}\right) \begin{array}{c} : \\ : \\ : \end{array} \begin{array}{c} w1^{(3)} \\ w2^{(3)} \\ w3^{(3)} \end{array} \tag{4.56}$$

Końcowe rozwiązanie znajdziemy ze wzoru (4.51):

$$\begin{aligned} x_3 &= \frac{b_3^{(3)}}{a_{33}^{(3)}} = \frac{3}{4}, \\ x_2 &= \frac{b_2^{(3)} - a_{23}^{(3)} x_3}{a_{22}^{(3)}} = -\frac{1}{4}, \\ x_1 &= \frac{b_1^{(3)} - a_{12}^{(3)} x_2 - a_{13}^{(3)} x_3}{a_{11}^{(3)}} = \frac{5}{4}. \end{aligned} \tag{4.57}$$

Zastanówmy się teraz, co stanie się w sytuacji, kiedy eliminację Gaussa zastosujemy do układu z macierzą osobliwą.

Przykład Rozważmy układ

$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}. \tag{4.58}$$

Po kilku krokach dojdziemy do sytuacji (sprawdzić!):

$$\left(\begin{array}{ccc|c} 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right). \tag{4.59}$$

Same zera w ostatnim wierszu sygnalizują, że wyjściowa macierz była osobliwa, a zatem nie istnieje rozwiązanie jednoznaczne. Ponadto, ponieważ ostatni element wektora wyrazów wolnych jest również równy zero, mamy tu do czynienia z nieskończoną liczbą rozwiązań.

Częściowy wybór elementu podstawowego zalecany jest również dla układów, których macierze nie mają zerowych elementów na głównej przekątnej, ponieważ w większości przypadków prowadzi do redukcji błędów zaokrągleń. Zdarzają sie jednak przypadki, w których dokładność po jego zastosowaniu może się pogorszyć. O tym, jak delikatną sprawą jest odpowiednia strategia wyboru elementu, mogą świadczyć poniższe dwa przykłady [35].

Przykład Rozważmy układ

$$\begin{pmatrix} 10^{-15} & 1 \\ 1 & 10^{11} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1+10^{-15} \\ 10^{11}+1 \end{pmatrix}. \tag{4.60}$$

Eliminacja Gaussa bez wyboru elementu podstawowego da wprawdzie poprawne rozwiązanie[2],

$$\left(\begin{array}{cc|c} 10^{-15} & 1 & 1+10^{-15} \\ 1 & 10^{11} & 10^{11}+1 \end{array}\right) \xrightarrow{eliminacja} \left(\begin{array}{cc|c} 1 & 10^{15} & 1+10^{15} \\ 0 & 10^{11}-10^{15} & 10^{11}-10^{15} \end{array}\right)$$

$$\xrightarrow{podstawianie} \vec{x} = \begin{pmatrix} 1 \\ 1 \end{pmatrix},$$

ale tylko pod warunkiem, że wszystkie obliczenia odbywają się dokładnie. Błędy zaokrągleń spowodują, że wynik będzie znacznie odbiegał od idealnego:

$$\xrightarrow{eliminacja} \left(\begin{array}{cc|c} 1 & 9.999999999999999e+14 & 1.000000000000001e+015 \\ 0 & -9.998999999999999e+014 & -9.999000000000000e+014 \end{array}\right)$$

$$\xrightarrow{podstawianie} \vec{x} = \begin{pmatrix} 8.750000000000000e-001 \\ 1.000000000000000e+000 \end{pmatrix}.$$

Lepszy wynik uzyskamy, dokonując częściowego wyboru elementu podstawowego:

$$\left(\begin{array}{cc|c} 10^{-15} & 1 & 1+10^{-15} \\ 1 & 10^{11} & 10^{11}+1 \end{array}\right) \xrightarrow{zamiana\ wierszy} \left(\begin{array}{cc|c} 1 & 10^{11} & 10^{11}+1 \\ 10^{-15} & 1 & 1+10^{-15} \end{array}\right)$$

[2]W celu poprawienia dokładności pierwszy wiersz układu równań został przeskalowany, tzn. obustronnie pomnożony przez 10^{15}. Zabieg ten będziemy stosować również w drugim przykładzie.

$$\xrightarrow{eliminacja} \left(\begin{array}{cc|c} 1 & 1.000e+011 & 1.000000000010000e+011 \\ 0 & 9.999e-001 & 9.999000000000001e-001 \end{array}\right)$$

$$\xrightarrow{podstawianie} \vec{x} = \begin{pmatrix} 9.999847412109375e-001 \\ 1.000000000000000e+000 \end{pmatrix}.$$

Przykład Weźmy teraz pod lupę inny układ, którego dokładne rozwiązanie to $\vec{x} = (1,1)^{\mathrm{T}}$, a mianowicie

$$\begin{pmatrix} 10^{-14.6} & 1 \\ 1 & 10^{15} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1+10^{-14.6} \\ 10^{15}+1 \end{pmatrix}. \tag{4.61}$$

Eliminacja Gaussa da poprawny wynik,

$$\xrightarrow{eliminacja} \left(\begin{array}{cc|c} 1 & 3.981071705534969e+014 & 3.981071705534979e+014 \\ 0 & 6.018928294465030e+014 & 6.018928294465030e+014 \end{array}\right)$$

$$\xrightarrow{podstawianie} \vec{x} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Natomiast, jeżeli zamienimy wiersze zgodnie ze strategią częściowego wyboru elementu podstawowego, wynik znacznie się pogorszy:

$$\left(\begin{array}{cc|c} 10^{-14.6} & 1 & 1+10^{-14.6} \\ 1 & 10^{15} & 10^{15}+1 \end{array}\right) \xrightarrow{zamiana\ wierszy} \left(\begin{array}{cc|c} 1 & 10^{15} & 10^{15}+1 \\ 10^{-14.6} & 1 & 1+10^{-14.6} \end{array}\right)$$

$$\xrightarrow{eliminacja} \left(\begin{array}{cc|c} 1 & 1.000e+015 & 1.000000000000001e+015 \\ 0 & -1.5118864315095819 & -1.5118864315095821 \end{array}\right)$$

$$\xrightarrow{podstawianie} \vec{x} = \begin{pmatrix} 0.750000000000000 \\ 1.000000000000002 \end{pmatrix}$$

W powyższych przykładach po cichu zastosowaliśmy skalowanie jednego z równań w celu poprawienia dokładności. Zabieg taki nazywany jest równoważeniem układu równań. Poniższy przykład powinien wyjaśnić jego sens.

Przykład Rozważmy układ

$$\begin{pmatrix} 1 & 10000 \\ 1 & 0,0001 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 10000 \\ 1 \end{pmatrix}. \tag{4.62}$$

Układ ten ma rozwiązanie $x_1 = x_2 = 0,9999$, poprawnie zaokrąglone do czterech cyfr dziesiętnych.

Przyjmijmy a_{11} jako element podstawowy i poszukajmy rozwiązań układu w trzycyfrowej arytmetyce zmiennopozycyjnej. Otrzymamy następujące, złe rozwiązanie

$$x_1 = 0.00, \quad x_2 = 1.00. \tag{4.63}$$

Pomnóżmy teraz pierwsze równanie przez 10^{-4},

$$\begin{pmatrix} 0,0001 & 1 \\ 1 & 0,0001 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}. \tag{4.64}$$

Wybierając a_{21} jako element podstawowy, otrzymamy

$$x_1 = 1.00, \quad x_2 = 1.00, \tag{4.65}$$

co w trzycyfrowej arytmetyce jest wynikiem dobrym.

Jeżeli nie jesteśmy pewni wyniku, oprócz równoważenia układu równań możemy spróbować zmienić strategię wyboru elementu podstawowego, biorąc pod uwagę nie tylko wartości elementów macierzy w i-tej kolumnie, ale również w i-tym wierszu. Metody tej nie stosuje się często, ponieważ zamiana kolumn (i odpowiednia zamiana kolejności składowych w wektorze niewiadomych) wiąże się z dodatkowymi kosztami w obliczeniach.

4.2.5 Macierze odwrotne

Jak już zostało wspomniane, eliminacji Gaussa możemy używać do rozwiązania wielu układów różniących się tylko, wyrazem wolnym,

$$\mathbf{A}\vec{x}_1 = \vec{b}_1, \quad \mathbf{A}\vec{x}_2 = \vec{b}_2, \quad \ldots, \mathbf{A}\vec{x}_N = \vec{b}_N. \tag{4.66}$$

Tworząc macierze z wektorów $\vec{x}_i$ i $\vec{b}_i$, zapiszemy układy (4.66) w zwartej formie

$$\mathbf{A}\left(\vec{x}_1\vec{x}_2\ldots\vec{x}_N\right) = \left(\vec{b}_1\vec{b}_2\ldots\vec{b}_N\right), \tag{4.67}$$

czyli

$$\mathbf{AX} = \mathbf{B}. \tag{4.68}$$

Formalne rozwiązanie ostatniego równania macierzowego ma postać

$$\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}. \tag{4.69}$$

Jeżeli $\mathbf{B}$ będzie macierzą jednostkową, znajdziemy w ten sposób macierz odwrotną do macierzy $\mathbf{A}$. Wystarczy zatem za wektory $\vec{b}_i$ wziąć kolejne kolumny macierzy jednostkowej i rozwiązać układy (4.66), a uzyskane rozwiązania będą kolumnami macierzy odwrotnej.

4.2.6 Eliminacja Jordana

Rozważmy raz jeszcze układ równań

$$\begin{array}{ccccccccc}
a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \ldots & + & a_{1n}^{(1)}x_n & = & b_1^{(1)} \\
a_{21}^{(1)}x_1 & + & a_{22}^{(1)}x_2 & + & \ldots & + & a_{2n}^{(1)}x_n & = & b_2^{(1)} \\
 & & & & & & & \vdots & \\
a_{n1}^{(1)}x_1 & + & a_{n2}^{(1)}x_2 & + & \ldots & + & a_{nn}^{(1)}x_n & = & b_n^{(1)}
\end{array} \tag{4.70}$$

Dzielimy pierwsze równanie obustronnie przez $a_{11}^{(1)}$,a następnie od i–tego wiersza $(i = 2, 3, \ldots, n)$ odejmujemy pierwszy pomnożony przez $a_{i1}^{(1)}$,

$$\begin{array}{ccccccccc}
x_1 & + & a_{12}^{(2)}x_2 & + & \ldots & + & a_{1n}^{(2)}x_n & = & b_1^{(2)} \\
 & & a_{22}^{(2)}x_2 & + & \ldots & + & a_{2n}^{(2)}x_n & = & b_2^{(2)} \\
 & & & & & & & \vdots & \\
 & & a_{n2}^{(2)}x_2 & + & \ldots & + & a_{nn}^{(2)}x_n & = & b_n^{(2)}
\end{array} \tag{4.71}$$

W kolejnym kroku dzielimy drugie równanie obustronnie przez $a_{22}^{(2)}$ i odejmujemy od i–tego wiersza ($i = 1, 3, 4, \ldots, n$) wiersz drugi pomnożony przez $a_{i2}^{(2)}$,

$$\begin{array}{ccccccccc} x_1 & & + & \ldots & + & a_{1n}^{(3)} x_n & = & b_1^{(3)} \\ & x_2 & + & \ldots & + & a_{2n}^{(3)} x_n & = & b_2^{(3)} \\ & & & & & & \vdots & \\ & & & \ldots & + & a_{nn}^{(3)} x_n & = & b_n^{(3)} \end{array} \tag{4.72}$$

Po $(n-1)$ eliminacjach otrzymujemy układ

$$\begin{array}{ccccc} x_1 & & & = & b_1^{(n)} \\ & x_2 & & = & b_2^{(n)} \\ & & \ddots & \vdots & \\ & & x_n & = & b_n^{(n)} \end{array} \tag{4.73}$$

czyli gotowe rozwiązanie.

Metoda ta wymaga

$$M = \frac{1}{2}n^3 + \frac{1}{2}n^2, \quad D = \frac{1}{2}n^3 - \frac{1}{2}$$

działań, czyli liczby około półtora razy większej niż w przypadku eliminacji Gaussa. W celu zagwarantowania jej niezawodności należy zastosować odpowiedni wybór elementu podstawowego. Do jej zalet należą oszczędne gospodarowanie pamięcią oraz możliwość prostego określenia rozwiązania " obciętego " układu równań. Natomiast wadami są: duży nakład obliczeń oraz brak odpowiednika rozkładu **LU**, o którym będzie mowa w następnym paragrafie.

4.2.7 Rozkład LU

Wiemy już, że rozwiązanie układu równań z macierzami trójkątnymi jest szczególnie łatwe. Przypuśćmy zatem, że macierz **A** dowolnego układu da się przedstawić w postaci iloczynu macierzy trójkątnej dolnej **L** i trójkątnej górnej **U**,

$$\mathbf{A} = \mathbf{LU}. \tag{4.74}$$

Jeżeli macierz **A** jest nieosobliwa, zachodzi

$$\mathbf{A}^{-1} = (\mathbf{LU})^{-1} = \mathbf{U}^{-1}\mathbf{L}^{-1}, \tag{4.75}$$

a więc rozwiązanie układu (4.1) da się przedstawić w postaci

$$\vec{x} = \mathbf{A}^{-1}\vec{b} = \mathbf{U}^{-1}\left(\mathbf{L}^{-1}\vec{b}\right). \tag{4.76}$$

Aby znaleźć rozwiązanie $\vec{x}$ układu dysponując rozkładem LU jego macierzy, wystarczy rozwiązać dwa układy trójkątne:

$$\begin{array}{rcl} \mathbf{L}\vec{y} & = & \vec{b}, \\ \mathbf{U}\vec{x} & = & \vec{y}. \end{array} \tag{4.77}$$

Eliminacja Gaussa a rozkład LU

Okazuje się, że jednym ze sposobów uzyskania rozkładu LU jest omówiona już eliminacja Gaussa. Zwróćmy mianowicie uwagę, że przekształcenie

$$\mathbf{A}^{(1)}x = b^{(1)} \quad \rightarrow \quad \mathbf{A}^{(2)}x = b^{(2)} \tag{4.78}$$

jest równoważne pomnożeniu obu stron układu $\mathbf{A}^{(1)}x = b^{(1)}$ przez macierz

$$\mathbf{L}^{(1)} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ -l_{21} & 1 & 0 & \dots & 0 \\ -l_{31} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -l_{n1} & 0 & 0 & \dots & 1 \end{pmatrix}, \tag{4.79}$$

gdzie

$$l_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}}, \quad i = 2, 3, \dots, n. \tag{4.80}$$

W ten sposób otrzymujemy dwa równania:

$$\mathbf{L}^{(1)}\mathbf{A}^{(1)} = \mathbf{A}^{(2)}, \quad \mathbf{L}^{(1)}b^{(1)} = b^{(2)}. \tag{4.81}$$

Podobnie,

$$\mathbf{L}^{(2)}\mathbf{A}^{(2)} = \mathbf{A}^{(3)}, \quad \mathbf{L}^{(2)}b^{(2)} = b^{(3)}, \tag{4.82}$$

przy czym

$$\mathbf{L}^{(2)} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & -l_{32} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & -l_{n2} & 0 & \dots & 1 \end{pmatrix}, \quad l_{i2} = \frac{a_{i2}^{(2)}}{a_{22}^{(2)}}, \quad i = 3, \dots, n. \tag{4.83}$$

Ostatecznie,

$$\mathbf{L}^{(n-1)}\mathbf{L}^{(n-2)} \dots \mathbf{L}^{(1)}\mathbf{A}^{(1)} = \mathbf{A}^{(n)} \tag{4.84}$$

oraz

$$\mathbf{L}^{(n-1)}\mathbf{L}^{(n-2)} \dots \mathbf{L}^{(1)}b^{(1)} = b^{(n)}. \tag{4.85}$$

Macierze $\mathbf{L}^{(i)}$, $i = 1, \dots, n-1$, są nieosobliwe, możemy więc zapisać

$$\mathbf{A}^{(1)} = (\mathbf{L}^{(1)})^{-1}(\mathbf{L}^{(2)})^{-1} \dots (\mathbf{L}^{(n)})^{-1}\mathbf{A}^{(n)}. \tag{4.86}$$

Ponadto,

$$(\mathbf{L}^{(1)})^{-1} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ l_{21} & 1 & 0 & \dots & 0 \\ l_{31} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & 0 & 0 & \dots & 1 \end{pmatrix}, \quad (\mathbf{L}^{(2)})^{-1} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & l_{32} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & l_{n2} & 0 & \dots & 1 \end{pmatrix}, \text{ itd.} \tag{4.87}$$

oraz

$$\mathbf{L} \equiv (\mathbf{L}^{(1)})^{-1}(\mathbf{L}^{(2)})^{-1}\dots(\mathbf{L}^{(n)})^{-1} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ l_{21} & 1 & 0 & \dots & 0 \\ l_{31} & l_{32} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \dots & 1 \end{pmatrix}. \tag{4.88}$$

Z drugiej strony wiemy, że $\mathbf{A}^{(n)}$ jest macierzą trójkątną górną. Oznaczając $\mathbf{A}^{(n)}$ przez $\mathbf{U}$, rzeczywiście otrzymamy rozkład (4.74) macierzy $\mathbf{A}$.

Zapamiętując macierze $\mathbf{L}$ i $\mathbf{U}$, możemy szybko rozwiązać wiele układów różniących się tylko kolumnami wyrazów wolnych. Ponadto, chcąc oszczędzać pamięć maszyny liczącej, możemy zapisywać elementy tych macierzy w miejsce elementów macierzy $\mathbf{A}$,

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{pmatrix} \rightarrow \begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ l_{21} & u_{22} & u_{23} & \dots & u_{2n} \\ l_{31} & l_{32} & u_{33} & \dots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \dots & u_{nn} \end{pmatrix}. \tag{4.89}$$

Nie każdą macierz nieosobliwą można przedstawić w postaci (4.74). Aby rozkład istniał, wszystkie minory główne macierzy muszą być różne od zera. Jednak, jeżeli eliminację Gaussa można przeprowadzić do końca, rozkład LU na pewno istnieje.

Jeżeli eliminacja Gaussa wymaga zamiany wierszy, wówczas zamiast rozkładu LU macierzy $\mathbf{A}$ znajdziemy rozkład permutacji jej wierszy,

$$\mathbf{PA} = \mathbf{LU}, \tag{4.90}$$

gdzie $\mathbf{P}$ to macierz permutacji. Jej znaczenie najlepiej jest zilustrować na przykładzie:

$$\mathbf{PA} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} a_{31} & a_{32} & a_{33} \\ a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}. \tag{4.91}$$

Macierz permutacji ma następującą własność:

$$\mathbf{P}^{\mathrm{T}}\mathbf{P} = \mathbf{I} \quad \Rightarrow \quad \mathbf{P}^{\mathrm{T}} = \mathbf{P}^{-1}. \tag{4.92}$$

Stąd wynika

$$\mathbf{A} = \mathbf{P}^{\mathrm{T}}\mathbf{LU}. \tag{4.93}$$

Metoda Doolittle'a

Rozkładu LU możemy poszukać również w inny sposób, traktując równość (4.74),

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}, \tag{4.94}$$

jako układ n^2 równań dla n^2 niewiadomych l_{ij} i u_{ij}:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ l_{21}u_{11} & l_{21}u_{12}+u_{22} & l_{21}u_{13}+u_{23} \\ l_{31}u_{11} & l_{31}u_{12}+l_{32}u_{22} & l_{31}u_{13}+l_{32}u_{23}+u_{33} \end{pmatrix}. \tag{4.95}$$

Stąd

$$\begin{array}{lll} u_{11}=a_{11}, & u_{12}=a_{12}, & u_{13}=a_{13}, \\ l_{21}=\frac{a_{21}}{u_{11}}, & u_{22}=a_{21}-l_{21}u_{12}, & u_{23}=a_{23}-l_{21}u_{13} \\ l_{31}=\frac{a_{31}}{u_{11}}, & l_{32}=\frac{a_{32}-l_{31}u_{12}}{u_{22}}, & u_{33}=a_{33}-l_{31}u_{13}-l_{32}u_{23}. \end{array} \tag{4.96}$$

W przypadku ogólnym elementy macierzy **L** i **U** obliczamy kolejno dla $i = 1, 2, \ldots, n$ ze wzorów

$$\begin{aligned} u_{ij} &= a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}, \quad j = i, i+1, \ldots, n, \\ l_{ji} &= \frac{a_{ji} - \sum_{k=1}^{i-1} l_{jk}u_{ki}}{u_{ii}}, \quad j = i+1, i+2, \ldots, n. \end{aligned} \tag{4.97}$$

Ta metoda obliczania rozkładu **LU** macierzy **A** wymaga $M = \frac{1}{3}n^3 - \frac{1}{3}n$ mnożeń i $D = \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$ dodawań. Jeżeli do tego dodamy liczbę działań potrzebną do rozwiązania dwóch trójkątnych układów równań, otrzymamy $M = \frac{1}{3}n^3 + n^2 - \frac{1}{3}n$ i $D = \frac{1}{3}n^3 + \frac{1}{3}n^2 - \frac{5}{6}n$, czyli tyle samo, co w metodzie Gaussa.

Metoda Doolitle'a staje się niezawodna dopiero w połączeniu z wyborem elementu podstawowego. Wiersze powinniśmy zamieniać ze sobą miejscami tak, aby element u_{ii} we wzorze (4.97) był jak największy.

Przykład Chcemy wyznaczyć rozkład LU macierzy

$$\begin{pmatrix} 20 & 31 & 23 \\ 30 & 24 & 18 \\ 15 & 32 & 21 \end{pmatrix} \tag{4.98}$$

metodą Doolittle'a z częściowym wyborem elementu podstawowego. W tym celu wprowadzamy dodatkową kolumnę indeksującą wiersze

$$\begin{pmatrix} 20 & 31 & 23 \\ 30 & 24 & 18 \\ 15 & 32 & 21 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}. \tag{4.99}$$

Element podstawowy wybieramy tak, aby element u_{ii} występujący w drugim ze wzorów (4.97) miał jak największą wartość.

Dla $i = 1$ w zależności od tego, czy na pierwszym miejscu ustawimy wiersz pierwszy, drugi czy trzeci, uzyskamy odpowiednio $u_{11} = 20$, $u_{11} = 30$ oraz $u_{11} = 15$. Dlatego zamieniamy miejscami wiersz pierwszy z drugim,

$$\begin{pmatrix} 30 & 24 & 18 \\ 20 & 31 & 23 \\ 15 & 32 & 21 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix}. \tag{4.100}$$

Korzystając ze wzorów (4.97) otrzymujemy:

$$\begin{array}{lll} u_{11} = 30, & u_{12} = a_{21} = 24, & u_{13} = a_{13} = 18, \\ l_{21} = \frac{2}{3}, & l_{31} = \frac{1}{2}. & \end{array} \tag{4.101}$$

Wartości te wpisujemy do macierzy **A**

$$\begin{pmatrix} 30 & 24 & 18 \\ \frac{2}{3} & 31 & 23 \\ \frac{1}{2} & 32 & 21 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix}. \tag{4.102}$$

Dla $i = 2$ otrzymamy

$$u_{22} = a_{22} - a_{21}a_{12} = 31 - \frac{2}{3} * 24 = 15$$

lub

$$u_{22} = a_{32} - a_{31}a_{12} = 32 - \frac{1}{2} * 24 = 20$$

w zależności od tego, czy na drugim miejscu ustawimy wiersz drugi czy trzeci. Zamieniamy wiersze miejscami,

$$\begin{pmatrix} 30 & 24 & 18 \\ \frac{1}{2} & 32 & 21 \\ \frac{2}{3} & 31 & 23 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}. \tag{4.103}$$

Ze wzorów (4.97) znajdujemy:

$$u_{22} = 20, \quad u_{23} = a_{23} - a_{21}a_{13} = 12, \quad u_{32} = \frac{15}{20}. \tag{4.104}$$

Uzyskane wartości wpisujemy do macierzy:

$$\begin{pmatrix} 30 & 24 & 18 \\ \frac{1}{2} & 20 & 12 \\ \frac{2}{3} & \frac{3}{4} & 23 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}. \tag{4.105}$$

Dla $i = 3$ obliczamy

$$u_{33} = 2. \tag{4.106}$$

Stąd

$$\begin{pmatrix} 30 & 24 & 18 \\ \frac{1}{2} & 20 & 12 \\ \frac{2}{3} & \frac{3}{4} & 2 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}. \tag{4.107}$$

W ten sposób w miejsce macierzy **A** otrzymaliśmy rozkład **LU** macierzy, która składa się z wierszy 2, 3 i 1 macierzy wyjściowej **A**.

Metoda Crouta

Do tej pory zakładaliśmy po cichu, że elementy diagonalne macierzy **L** są równe 1. Jeżeli dla odmiany przyjmiemy, że to **U** ma na głównej przekątnej same jedynki,

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{pmatrix}, \tag{4.108}$$

i ponownie potraktujemy powyższe wyrażenie jak równanie na niewiadome elementy macierzy trójkątnych, otrzymamy metodę zaproponowaną przez Crouta [15].

4.2.8 Wyznaczniki

Rozkład LU macierzy pozwala natychmiast wyliczyć jej wyznacznik. Zachodzi mianowicie

$$\det \mathbf{A} = \det(\mathbf{LU}) = \det \mathbf{L} \det \mathbf{U} = \begin{cases} u_{11}u_{22}\dots u_{nn}, & \text{jeśli } l_{ii} = 1,\ i = 1,\dots,n. \\ l_{11}l_{22}\dots l_{nn}, & \text{jeśli } u_{ii} = 1,\ i = 1,\dots,n. \end{cases} \tag{4.109}$$

4.2.9 Specjalne typy macierzy

Macierze dominujące diagonalnie

Istnieje pewna klasa macierzy, dla których nie trzeba przeprowadzać wyboru elementu podstawowego.

Definicja Macierz kwadratową $\mathbf{A}$ nazywamy diagonalnie dominującą, jeżeli

$$|a_{ii}| \geqslant \sum_{\substack{k=1 \\ k\neq i}}^{n} |a_{ik}|, \quad i = 1, 2, \dots, n. \tag{4.110}$$

Jeżeli nierówności są ostre, mówimy o macierzy silnie diagonalnie dominującej.

Definicja Macierz $\mathbf{A}$ jest diagonalnie dominująca kolumnowo, jeżeli $\mathbf{A}^{\mathrm{T}}$ jest diagonalnie dominująca, tzn.

$$|a_{ii}| \geqslant \sum_{\substack{k=1 \\ k\neq i}}^{n} |a_{ki}|, \quad i = 1, 2, \dots, n. \tag{4.111}$$

Zachodzi następujące twierdzenie:

Twierdzenie 4.2.1 *Jeżeli macierz* $\mathbf{A}$ *jest nieosobliwa i diagonalnie dominująca kolumnowo, to przy eliminacji metodą Gaussa nie ma potrzeby przestawiania wierszy.*

Dowód polega na pokazaniu, że po wyeliminowaniu m niewiadomych, uzyskana macierz jest nadal macierzą diagonalnie dominującą kolumnowo. Można go znaleźć np. w [50].

Macierze trójdiagonalne

Dość często będziemy mieli do czynienia z tzw. macierzami trójdiagonalnymi,

$$\mathbf{T} = \begin{pmatrix} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & & 0 & \\ & a_3 & b_3 & c_3 & & & \\ & & a_4 & b_4 & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & 0 & & & \ddots & b_{n-1} & c_{n-1} \\ & & & & & a_n & b_n \end{pmatrix}. \tag{4.112}$$

Rozkład LU takiej macierzy ma postać

$$\mathbf{L} = \begin{pmatrix} 1 & & & 0 \\ l_2 & \ddots & & \\ & \ddots & \ddots & \\ 0 & & l_n & 1 \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} u_1 & c_1 & & 0 \\ & \ddots & \ddots & \\ & & \ddots & c_{n-1} \\ 0 & & & u_n \end{pmatrix}, \tag{4.113}$$

gdzie

$$\begin{aligned} u_1 &= b_1, \\ l_i &= \frac{a_i}{u_{i-1}}, \\ u_i &= b_i - l_i c_{i-1}, \quad i = 2, 3, \ldots, n, \end{aligned} \tag{4.114}$$

i wymaga tylko $O(n)$ operacji. Przy tym, metoda jest niezawodna o ile tylko macierz $\mathbf{T}$ jest diagonalnie dominująca kolumnowo, co często spełnione jest w praktycznych zagadnieniach.

4.2.10 Błędy zaokrągleń

Rozwiązanie układu równań (4.1) za pomocą rozkładu LU możemy podzielić na trzy etapy. Pierwszy to wyznaczenie samego rozkładu, w drugim rozwiązujemy równanie $\mathbf{L}\vec{y} = \vec{b}$, natomiast w trzecim - równanie $\mathbf{U}\vec{x} = \vec{y}$.

Znalezione w pierwszym etapie macierze $\mathbf{L}$ i $\mathbf{U}$ spełniają warunek

$$\mathbf{LU} = \mathbf{A} + \mathbf{E}, \tag{4.115}$$

gdzie macierz $\mathbf{E}$ to błąd rozkładu. Obliczone w etapie drugim i trzecim $\vec{y}$ i $\vec{x}$ możemy potraktować jako dokładne rozwiązania układów

$$\begin{aligned} (\mathbf{L} + \delta\mathbf{L})\vec{y} &= \vec{b}, \\ (\mathbf{U} + \delta\mathbf{U})\vec{x} &= \vec{y}. \end{aligned} \tag{4.116}$$

$$\tag{4.117}$$

Stąd

$$(\mathbf{A} + \mathbf{E} + \mathbf{L}\delta\mathbf{U} + \delta\mathbf{L}\mathbf{U} + \delta\mathbf{L}\delta\mathbf{U})\vec{x} = \vec{b}. \tag{4.118}$$

Można pokazać [50], że zaburzenie

$$\delta\mathbf{A} = \mathbf{E} + \mathbf{L}\delta\mathbf{U} + \delta\mathbf{L}\mathbf{U} + \delta\mathbf{L}\delta\mathbf{U} \tag{4.119}$$

ma oszacowanie

$$\frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|} \leqslant \epsilon \left(\frac{9}{2}n^3 + \frac{61}{2}n^2 - 18n - 16 \right) + O(\epsilon). \tag{4.120}$$

gdzie ϵ to dokładność maszynowa. Stąd wynika, że

$$\frac{\|\delta\vec{x}\|}{\|\vec{x}\|} \leqslant \frac{\alpha}{1-\alpha}, \tag{4.121}$$

gdzie

$$\alpha = \epsilon K O(\frac{9}{2}n^3), \tag{4.122}$$

K oznacza wskaźnik uwarunkowania, a $O(\frac{9}{2}n^3)$ oznacza wielomian ze wzoru (4.120).

4.2.11 Inne rozkłady macierzy

Rozkład macierzy **A** na iloczyn **LU** nie jest jedynym możliwym rozkładem (patrz np. [15, 50]). W niniejszym paragrafie omówimy wybrane z nich.

Rozkład Cholesky'ego (Banachiewicza)

Jeżeli macierz układu jest macierzą symetryczną, tzn.

$$a_{ij} = a_{ji}, \quad i, j = 1, \ldots, n, \tag{4.123}$$

i dodatnio określoną,

$$\vec{x}^{\mathrm{T}} \mathbf{A} \vec{x} > 0 \quad \text{dla każdego } \vec{x}, \tag{4.124}$$

to istnieje dla niej bardziej wydajny od LU rozkład na macierze trójkątne, a mianowicie

$$\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathrm{T}}, \tag{4.125}$$

gdzie **L** to macierz trójkątna dolna. Traktując (4.125) jako układ równań ze względu na elementy macierzy **L** znajdziemy:

$$l_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2 \right)^{1/2}, \tag{4.126}$$

$$l_{ji} = \frac{1}{l_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik} l_{jk} \right), \quad j = i+1, i+2, \ldots, n. \tag{4.127}$$

Ilość operacji potrzebna do znalezienia rozkładu Cholesky'ego jest o połowę mniejsza w porównaniu z LU. Dodatkową zaletą, związaną z własnościami macierzy **A**, jest niezawodność (metoda nie wymaga wyboru elementu podstawowego) i stabilność numeryczna.

Rozkład SVD

W niektórych sytuacjach przydatny może się okazać rozkład macierzy na wartości osobliwe, czyli SVD (ang. *Singular Value Decomposition*).

Twierdzenie 4.2.2 *Każdą macierz* $\mathbf{A} \in \mathbf{R}^{m \times n}$ *rzędu* r *możemy przedstawić jako*

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}, \quad \mathbf{\Sigma} = \begin{pmatrix} \mathbf{\Sigma}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix} \in \mathbf{R}^{m \times n}, \tag{4.128}$$

gdzie $\mathbf{U} \in \mathbf{R}^{m \times m}$ *i* $\mathbf{V} \in \mathbf{R}^{n \times n}$ *są macierzami ortogonalnymi, a*

$$\mathbf{\Sigma}_1 = \mathrm{diag}\left(\sigma_1, \ldots, \sigma_r\right), \tag{4.129}$$

przy czym

$$\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant \sigma_r > 0. \tag{4.130}$$

Elementy σ_i *macierzy* $\mathbf{\Sigma}$ *nazywane są wartościami osobliwymi macierzy* $\mathbf{A}$.

Dowód w [48].

Praktyczny algorytm obliczania rozkładu macierzy na wartości osobliwe znaleźć można w [52–54]. Tutaj, za [55], podamy tylko jego szkic. Otóż w pierwszym kroku należy przekształcić macierz $\mathbf{A}$ do postaci

$$\mathbf{A} = \mathbf{QCH}^{\mathrm{T}} \tag{4.131}$$

gdzie $\mathbf{C}$ to macierz dwudiagonalna, a $\mathbf{Q}$ i $\mathbf{H}$ są iloczynami macierzy odpowiadających transformacji Householdera [48]. Następnie nadajemy macierzy $\mathbf{C}$ postać diagonalną,

$$\mathbf{C} = \mathbf{U}'\mathbf{\Sigma}'\mathbf{V}'^{\mathrm{T}}, \tag{4.132}$$

gdzie $\mathbf{U}'$ i $\mathbf{V}'$ to iloczyny macierzy opisujących transformację Givensa [48]. W trzecim kroku porządkuje się elementy diagonalne macierzami ortogonalnymi $\mathbf{U}''$ i $\mathbf{V}''$, wyrażającymi się poprzez iloczyny macierzy permutacji,

$$\mathbf{\Sigma} = \mathbf{U}''^{\mathrm{T}}\mathbf{\Sigma}'\mathbf{V}''. \tag{4.133}$$

Macierze $\mathbf{U}$ i $\mathbf{V}$ rozkładu SVD to po prostu

$$\mathbf{U} = \mathbf{QU}'\mathbf{U}'', \quad \mathbf{V} = \mathbf{HV}'\mathbf{V}''. \tag{4.134}$$

Rozkład SVD stosuje się do przybliżonych rozwiązań układów z macierzami osobliwymi albo prawie osobliwymi [15], a także do układów niedookreślonych (rozdział 4.4) i nadokreślonych (rozdział 4.5). Ponadto pozwala m.in. wyznaczyć numeryczny rząd macierzy oraz jej wskaźnik uwarunkowania [48].

Rozkład QR

Innym bardzo ważnym rozkładem macierzy $\mathbf{A}$, najczęściej wykorzystywanym w zagadnieniach na wartości własne (patrz rozdział 9), ale przydatnym również przy rozwiązywaniu układów równań liniowych, jest rozkład

$$\mathbf{A} = \mathbf{QR}, \tag{4.135}$$

gdzie $\mathbf{R}$ jest macierzą trójkątną górną, a $\mathbf{Q}$ - macierzą ortogonalną,

$$\mathbf{Q}^{\mathrm{T}}\mathbf{Q} = \mathbf{1}. \tag{4.136}$$

Do wyznaczenia tego rozkładu stosuje się zmodyfikowaną metodę Grama-Schmidta [36]. Polega ona na obliczeniu ciągu macierzy

$$\mathbf{A} = \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(n+1)} = \mathbf{Q}, \tag{4.137}$$

gdzie $\mathbf{A}^{(k)}$ ma postać

$$\mathbf{A}^{(k)} = \left(\vec{q}_1, \ldots, \vec{q}_{k-1}, \vec{a}_k^{(k)}, \ldots, \vec{a}_n^{(k)}\right). \tag{4.138}$$

Kolumny $\vec{q}_1,\ldots,\vec{q}_{k-1}$ są $k-1$ początkowymi kolumnami macierzy $\mathbf{Q}$, a kolumny $\vec{a}_k^{(k)}, \ldots, \vec{a}_n^{(k)}$ powinny być ortogonalne do $\vec{q}_1,\ldots,\vec{q}_{k-1}$. Ortogonalność w k-tym kroku kolumn od $k+1$ do n względem $\vec{q}_k$ zapewnia się w następujący sposób:

$$\vec{q}_k = \vec{a}_k^{(k)}, \quad d_k = \vec{q}_k^{\mathrm{T}}\vec{q}_k, \quad r_{kk} = 1, \quad \vec{a}_j^{k+1} = \vec{a}_j^{(k)} - r_{jk}\vec{q}_k, \tag{4.139}$$

$$r_{jk} = \frac{\vec{q}_k^{\mathrm{T}}\vec{a}_j^{(k)}}{d_k}, \quad j = k+1, \ldots, n. \tag{4.140}$$

Po n krokach ($k = 1, \ldots, n$) otrzymamy macierze $\mathbf{Q} = (\vec{q}_1, \ldots, \vec{q}_n)$ i $\mathbf{R} = (r_{kj})$ o pożądanych własnościach.

4.2.12 Iteracyjne poprawianie rozwiązań

Z dotychczasowych rozważań jasno wynika, że rozwiązanie układu równań $\mathbf{A}\vec{x} = \vec{b}$ dowolną metodą bezpośrednią będzie zwykle obarczone pewnym błędem (patrz par. 4.2.1 i 4.2.10). Błąd ten możemy wykryć, sprawdzając, jak bardzo tzw. wektor reszt

$$\vec{r} = \vec{b} - \mathbf{A}\vec{x} \tag{4.141}$$

różni się od zera. Powinniśmy przy tym liczyć $\vec{r}$ z dokładnością większą niż dokładność uzyskanego rozwiązania. W przeciwnym bowiem razie, ze względu na znoszenie się składników, możemy uzyskać $\vec{r} = 0$ dla $\vec{x}$, które nie jest rozwiązaniem równania lub odwrotnie, $\vec{r} \neq 0$ dla $\vec{x}$, które to równanie rozwiązuje. Aby się o tym przekonać, rozważmy następujący przykład.

Przykład Układ

$$\begin{pmatrix} 0,99 & 0,70 \\ 0,70 & 0,50 \end{pmatrix} \vec{x} = \begin{pmatrix} 0,54 \\ 0,36 \end{pmatrix} \tag{4.142}$$

ma rozwiązanie dokładne

$$\vec{x}_{dok} = \begin{pmatrix} 0,80 \\ -0,36 \end{pmatrix}.$$

Obliczmy najpierw $\vec{r}$ w arytmetyce zmiennopozycyjnej o dwóch miejscach dziesiętnych w mantysie, dokonując zaokrągleń. Otrzymamy

$$\begin{aligned} \vec{r}(\vec{x}_{dok}) &= \begin{pmatrix} 0,54 \\ 0,36 \end{pmatrix} - \begin{pmatrix} 0,99 & 0,70 \\ 0,70 & 0,50 \end{pmatrix} \begin{pmatrix} 0,80 \\ -0,36 \end{pmatrix} \\ &= \begin{pmatrix} 0,54 - 0,79 + 0,25 \\ 0,38 - 0,56 + 0,18 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{aligned} \tag{4.143}$$

Czy możemy stąd wnioskować, że $\vec{x}_{dok}$ jest dokładnym rozwiązaniem równania? Otóż nie, ponieważ dla np.

$$\vec{x}_1 = \begin{pmatrix} 0,02 \\ 0,74 \end{pmatrix}$$

mamy również

$$\begin{aligned} \vec{r}(\vec{x}_1) &= \begin{pmatrix} 0,54 \\ 0,36 \end{pmatrix} - \begin{pmatrix} 0,99 & 0,70 \\ 0,70 & 0,50 \end{pmatrix} \begin{pmatrix} 0,02 \\ 0,74 \end{pmatrix} \\ &= \begin{pmatrix} 0,54 - 0,02 - 0,52 \\ 0,38 - 0,01 - 0,37 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \end{aligned} \tag{4.144}$$

mimo że $\vec{x}_1$ rozwiązaniem równania nie jest i różni się dość sporo od rozwiązania dokładnego,

$$\|\vec{x}_{dok} - \vec{x}_1\|_\infty = 1,1.$$

Policzmy teraz wektory reszt z większą liczbą miejsc dziesiętnych w mantysie. Otrzymamy

$$\begin{aligned} \vec{r}(\vec{x}_{dok}) &= \begin{pmatrix} 0,54 \\ 0,36 \end{pmatrix} - \begin{pmatrix} 0,99 & 0,70 \\ 0,70 & 0,50 \end{pmatrix} \begin{pmatrix} 0,80 \\ -0,36 \end{pmatrix} \\ &= \begin{pmatrix} 0,54 - 0,792 + 0,252 \\ 0,38 - 0,56 + 0,18 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{aligned} \tag{4.145}$$

oraz

$$\begin{aligned}\vec{r}(\vec{x}_1) &= \begin{pmatrix} 0,54 \\ 0,36 \end{pmatrix} - \begin{pmatrix} 0,99 & 0,70 \\ 0,70 & 0,50 \end{pmatrix} \begin{pmatrix} 0,02 \\ 0,74 \end{pmatrix} \\ &= \begin{pmatrix} 0,54 - 0,0198 - 0,518 \\ 0,38 - 0,014 - 0,37 \end{pmatrix} = \begin{pmatrix} 0,0022 \\ -0,004 \end{pmatrix}. \end{aligned} \tag{4.146}$$

Dopiero teraz widać, że $\vec{x}_{dok}$ jest rozwiązaniem naszego układu równań, natomiast $\vec{x}_1$ nim nie jest.

Załóżmy na moment, że dla wyznaczonego rozwiązania $\vec{x}$ potrafimy dokładnie obliczyć wektor reszt. Wówczas przy dokładnych obliczeniach moglibyśmy znaleźć poprawkę $\delta\vec{x}$ taką, że

$$\vec{x} + \delta\vec{x} = \vec{x}_{dok}, \tag{4.147}$$

gdzie $\vec{x}_{dok}$ oznacza rozwiązanie dokładne. Rzeczywiście, ponieważ zachodzi

$$\vec{r} = \vec{b} - \mathbf{A}\vec{x} = \mathbf{A}\vec{x}_{dok} - \mathbf{A}\vec{x} = \mathbf{A}(\vec{x}_{dok} - \vec{x}) = \mathbf{A}\delta\vec{x}, \tag{4.148}$$

wystarczy, że rozwiążemy ostatni układ równań. Jeżeli przy tym do rozwiązania układu $\mathbf{A}\vec{x} = \vec{b}$ stosowaliśmy rozkład LU macierzy $\mathbf{A}$ i zapamiętaliśmy macierze trójkątne, to poprawkę $\delta\vec{x}$ znajdziemy szczególnie szybko, wykonując n^2 mnożeń i $n^2 - n$ dodawań.

Oczywiście, w rzeczywistych obliczeniach numerycznych nie potrafimy liczyć dokładnie i zamiast poprawki $\delta\vec{x}$ znajdziemy tylko poprawkę przybliżoną $\delta\vec{x} + \delta(\delta\vec{x})$. Jednak do ulepszonego rozwiązania $\vec{x} + \delta\vec{x}$ możemy znaleźć kolejną poprawkę. W ten sposób otrzymaliśmy przepis na iteracyjne poprawianie rozwiązań otrzymanych metodami dokładnymi:

1. rozwiąż układ równań $\mathbf{A}\vec{x}^{(1)} = \vec{b}$ stosując rozkład LU macierzy $\mathbf{A}$,
2. oblicz wektor reszt $\vec{r}^{(1)} = \vec{b} - \mathbf{A}\vec{x}^{(1)}$ (w podwójnej precyzji),
3. jeśli $\|\vec{r}^{(1)}\|_\infty \leqslant \|\mathbf{A}\vec{x}^{(1)}\|_\infty u$ (lub $\|\vec{r}^{(1)}\|_\infty \leqslant \|\vec{b}\|_\infty u$), gdzie u to jednostka maszynowa, przerwij obliczenia. Jeżeli nie , to ...
4. oblicz $\delta\vec{x}^{(1)}$ i wyznacz $\vec{x}^{(2)} = \vec{x}^{(1)} + \delta\vec{x}^{(1)}$,
5. oblicz $\vec{r}^{(2)} = \vec{b} - \mathbf{A}\vec{x}^{(2)}$ i przejdź ponownie do punktu 3.

Jeżeli macierz układu jest źle uwarunkowana, może się zdarzyć, że metoda ta nie doprowadzi do rozwiązania bliższego dokładnemu. Wtedy warto jest spróbować liczyć wszystkie wielkości w podwójnej precyzji. W pozostałych przypadkach metoda pozwala na wyznaczenie rozwiązania, którego wektor reszt jest rzędu $u\|\vec{b}\|_\infty$.

4.3 Metody iteracyjne dla układów określonych

Metoda poprawiania rozwiązań przedstawiona w paragrafie 4.2.12 to szczególny przypadek metod iteracyjnych, startujących z pewnego przybliżenia początkowego, które potem jest stopniowo ulepszane aż do uzyskania dostatecznie dokładnego rozwiązania. Podczas gdy tam wartość początkowa była rozwiązaniem

metody bezpośredniej, a więc miała już kilka cyfr dokładnych, teraz omówimy metody startujące z wartości, które na ogół będziemy zgadywać.

Metody iteracyjne stosuje się najczęściej do dużych układów rzadkich, tzn. takich, których macierze zawierają w większości zera.

4.3.1 Pojęcia podstawowe

Omówienie metod iteracyjnych rozpoczniemy od wprowadzenia pewnych pojęć, które okażą się użyteczne w dalszych rozważaniach.

Definicja Promieniem spektralnym $\rho(\mathbf{A})$ macierzy $\mathbf{A}$ nazywamy liczbę

$$\rho(\mathbf{A}) = \max_{i=1,\ldots,n} |\lambda_i|, \tag{4.149}$$

przy czym λ_i są wartościami własnymi macierzy $\mathbf{A}$.

Dla dowolnej normy macierzowej zgodnej z normą wektorów obowiązuje

$$|\lambda_i| \leqslant \|\mathbf{A}\|, \tag{4.150}$$

dla każdego $i = 1, \ldots, n$. Zatem

$$\rho(\mathbf{A}) \leqslant \|\mathbf{A}\|_p, \quad p = 1, 2, \infty, E. \tag{4.151}$$

Rozważmy ciąg wektorów $\vec{x}^{(0)}$, $\vec{x}^{(1)}$, $\ldots$, $\vec{x}^{(i)}$, określony dla dowolnego wektora $\vec{x}^{(0)}$ w następujący sposób:

$$\vec{x}^{(i+1)} = \mathbf{M}\vec{x}^{(i)} + \vec{w}, \quad i = 0, 1, \ldots, \tag{4.152}$$

gdzie $\mathbf{M}$ jest pewną macierzą kwadratową, a $\vec{w}$ wektorem.

Twierdzenie 4.3.1 *Ciąg określony wzorem (4.152) przy dowolnym wektorze $x^{(0)}$ jest zbieżny do jedynego punktu granicznego wtedy i tylko wtedy, gdy*

$$\rho(\mathbf{M}) < 1. \tag{4.153}$$

Dowód w [50].

Z nierówności (4.151) wynika, że dla zbieżności ciągu (4.152) wystarczy warunek $\|\mathbf{M}\| < 1$ dla dowolnej normy macierzy zgodnej z normą wektorów. Stąd wynika już praktyczny sposób konstruowania metod iteracyjnych stosowanych do rozwiązywania układu równań $\mathbf{A}\vec{x} = \vec{b}$: wystarczy tak dobrać macierz $\mathbf{M}$, aby ciąg

$$\vec{x}^{(i+1)} = \mathbf{M}\vec{x}^{(i)} + \vec{w}, \quad i = 0, 1, \ldots \tag{4.154}$$

był zbieżny, tzn. $\rho(\mathbf{M}) < 1$, oraz żeby spełniony był warunek zgodności

$$\vec{x}_{dok} = \mathbf{M}\vec{x}_{dok} + \vec{w}, \tag{4.155}$$

gdzie $\vec{x}_{dok}$ to dokładne rozwiązanie równania $\mathbf{A}\vec{x} = \vec{b}$. Teoretycznie wystarczy wziąć dowolną macierz $\mathbf{M}$ o promieniu spektralnym mniejszym od 1, a następnie wyliczyć $\vec{w}$ ze wzoru (4.155),

$$\vec{w} = (\mathbf{I} - \mathbf{M})\mathbf{A}^{-1}\vec{b}, \tag{4.156}$$

jednak wymagałoby to wyliczenia macierzy $\mathbf{A}^{-1}$. Dlatego na ogół postępuje się inaczej. Otóż zakłada się, że

$$\vec{w} = \mathbf{N}\vec{b}, \tag{4.157}$$

gdzie $\mathbf{N}$ jest pewną macierzą kwadratową. Z warunku zgodności mamy

$$\vec{x}_{dok} = \mathbf{M}\vec{x}_{dok} + \vec{w} \quad \Rightarrow (\mathbf{A}^{-1} - \mathbf{N} - \mathbf{M}\mathbf{A}^{-1})\vec{b} = 0 \quad \Rightarrow \mathbf{M} = \mathbf{I} - \mathbf{N}\mathbf{A}, \tag{4.158}$$

co prowadzi do

$$\vec{x}^{(i+1)} = (\mathbf{I} - \mathbf{N}\mathbf{A})\vec{x}^{(i)} + \mathbf{N}\vec{b}. \tag{4.159}$$

Powyższa rodzina iteracyjna wektorów $\vec{x}^{(i)}$ umożliwi wyznaczenie dokładnego rozwiązania $\vec{x}_{dok}$ równania $\mathbf{A}\vec{x} = \vec{b}$, jeżeli tylko

$$\rho(\mathbf{I} - \mathbf{N}\mathbf{A}) < 1. \tag{4.160}$$

Przy pewnych szczególnych własnościach macierzy układu $\mathbf{A}$ istnieją stosunkowo proste metody wyboru macierzy $\mathbf{N}$ takiej, aby warunek zbieżności (4.160) był spełniony.

Podczas oceny efektywności metod iteracyjnych, oprócz ilości działań niezbędnych do wykonania, potrzebnej pamięci i wielkości błędów zaokrągleń, ważne jest również określenie, jak szybko maleje wektor błędu

$$\vec{e}^{(i)} = x^{(i)} - \vec{x}_{dok}. \tag{4.161}$$

Może się bowiem okazać, że mimo spełnionego warunku zbieżności zagadnienie jest na tyle źle uwarunkowane, że osiągnięcie zadowalającej dokładności w rozsądnym czasie jest niemożliwe.

Przykład Niech

$$\mathbf{M} = \begin{pmatrix} \frac{1}{2} & 1 & & & \\ & \frac{1}{2} & 1 & & \\ & & \frac{1}{2} & \ddots & \\ & & & \ddots & 1 \\ & & & & \frac{1}{2} \end{pmatrix}, \vec{w} = \begin{pmatrix} -\frac{1}{2} \\ -\frac{1}{2} \\ \vdots \\ -\frac{1}{2} \\ \frac{1}{2} \end{pmatrix}. \tag{4.162}$$

Mamy tutaj $\rho(\mathbf{M}) = \frac{1}{2}$, a więc dla dowolnego $\vec{x}^{(0)}$ ciąg określony wzorem (4.152) dąży do $\vec{x}_{dok} = (1, \ldots, 1)^{\mathrm{T}}$. Przyjmijmy $\vec{x}^{(0)} = 0$. Wówczas, mimo że dla dostatecznie dużych indeksów i błąd będzie mały i będzie dążył do zera, w początkowych iteracjach będzie rósł,

$$\|\vec{e}^{(0)}\|_\infty = 1, \quad \|\vec{e}^{(1)}\|_\infty = \frac{3}{2}, \quad \|\vec{e}^{(2)}\|_\infty = \frac{9}{4}, \ldots,$$

co czasami może uniemożliwić numeryczne wyznaczenie rozwiązania.

Inne niekorzystne zjawisko to błędy zaokrągleń, które w skrajnym przypadku mogą doprowadzić do uzyskania

$$\vec{x}^{(i+1)} = \vec{x}^{(0)}. \tag{4.163}$$

Powstanie wówczas ciąg wektorów, który nie jest zbieżny do rozwiązania. Niestety, przed taką sytuacją bardzo trudno się ustrzec.

4.3.2 Metoda Jacobiego

Zapiszmy macierz układu równań w postaci []

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}, \tag{4.164}$$

gdzie macierze $\mathbf{L}$, $\mathbf{D}$ i $\mathbf{U}$ to odpowiednio macierz poddiagonalna, diagonalna i naddiagonalna.

Przykład

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 4 & 0 & 0 \\ 7 & 8 & 0 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix} + \begin{pmatrix} 0 & 2 & 3 \\ 0 & 0 & 6 \\ 0 & 0 & 0 \end{pmatrix} \tag{4.165}$$

Jako macierz $\mathbf{N}$ we wzorze (4.159) wybierzemy

$$\mathbf{N} = \mathbf{D}^{-1}. \tag{4.166}$$

Wówczas

$$\begin{aligned} \mathbf{M_J} &= \mathbf{I} - \mathbf{NA} = \mathbf{I} - \mathbf{D}^{-1}\mathbf{A} \\ &= \mathbf{I} - \mathbf{D}^{-1}(\mathbf{L} + \mathbf{D} + \mathbf{U}) = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U}) \end{aligned} \tag{4.167}$$

i wzór (4.159) prowadzi do tzw. wzoru Jacobiego

$$\mathbf{D}\vec{x}^{(i+1)} = -(\mathbf{L} + \mathbf{U})\vec{x}^{(i)} + \vec{b}, \quad i = 0, 1, 2, \ldots \tag{4.168}$$

Aby wzór ten był niezawodny, należy wcześniej w razie konieczności tak pozmieniać kolejność równań w układzie $\mathbf{A}\vec{x} = \vec{b}$, aby na diagonali macierzy układu były tylko elementy niezerowe. W tym celu

1. spośród kolumn z elementem zerowym na diagonali wybieramy tę, w której jest największa liczba zer;
2. w kolumnie tej wybieramy element o największym module i tak przestawiamy wiersze, aby znalazł się on na głównej przekątnej; wiersz ustalamy i pomijamy go w dalszych rozważaniach;
3. spośród pozostałych kolumn z elementem zerowym na diagonali wybieramy tę o największej liczbie zer i wracamy do punktu 2 aż do usunięcia wszystkich zer z głównej przekątnej.

Przykład Rozważmy macierz

$$\begin{pmatrix} 0 & 0 & 1 & 2 \\ 2 & 1 & 0 & 2 \\ 7 & 3 & 0 & 1 \\ 0 & 5 & 0 & 0 \end{pmatrix}.$$

Najwięcej zer znajduje się w kolumnie trzeciej, a element o największym module w tej kolumnie to element a_{13}. Zamieniamy miejscami wiersze 1 i 3, tak, aby element ten znalazł się na diagonali,

$$\begin{pmatrix} 7 & 3 & 0 & 1 \\ 2 & 1 & 0 & 2 \\ 0 & 0 & 1 & 2 \\ 0 & 5 & 0 & 0 \end{pmatrix}.$$

Zero na diagonali znajduje się jeszcze w kolumnie czwartej, a element o największym module w niej to a_{24}. Zamieniamy więc miejscami wiersze 2 i 4,

$$\begin{pmatrix} 7 & 3 & 0 & 1 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 1 & 2 \\ 2 & 1 & 0 & 2 \end{pmatrix}.$$

W ten sposób otrzymaliśmy macierz, którą można już zastosować w metodzie Jacobiego.

Należy podkreślić, że zamiana wierszy w macierzy gwarantuje jedynie, że będzie istniała macierz odwrotna do macierzy $\mathbf{D}$. Natomiast spełnienie warunku zbieżności metody Jacobiego, tzn.

$$\rho(-\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})) < 1, \tag{4.169}$$

nie jest gwarantowane w każdym przypadku. Można jedynie pokazać, że jest tak zawsze, jeżeli macierz $\mathbf{A}$ jest silnie diagonalnie dominująca lub silnie diagonalnie dominująca kolumnowo.

4.3.3 Metoda Gaussa–Seidla

Podobnie, jak w metodzie Jacobiego, również i tutaj rozkładamy macierz układu na sumę macierzy poddiagonalnej, diagonalnej i naddiagonalnej (w razie konieczności odpowiednio przestawiając wiersze),

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}. \tag{4.170}$$

Przyjmujemy

$$\mathbf{N} = (\mathbf{D}+\mathbf{L})^{-1}, \tag{4.171}$$

co prowadzi do

$$\mathbf{M}_{GS} = -(\mathbf{D}+\mathbf{L})^{-1}\mathbf{U}, \tag{4.172}$$

oraz

$$\mathbf{D}\vec{x}^{(i+1)} = -\mathbf{L}\vec{x}^{(i+1)} - \mathbf{U}\vec{x}^{(i)} + b, \quad i = 0, 1, 2, \ldots \tag{4.173}$$

Na pierwszy rzut oka powyższe równanie wygląda tak, jakby niewiadome występowały po obu stronach jednocześnie. Zauważmy jednak, że przy obliczaniu pierwszej współrzędnej szukanego wektora po prawej stronie równania (4.173) nie wystąpi żadna współrzędna wektora $\vec{x}^{(i+1)}$, zatem $x_1^{(i+1)}$ możemy wyznaczyć bez problemów. Przy obliczaniu $x_2^{(i+1)}$ prawa strona równości będzie zależała tylko od $x_1^{(i+1)}$, więc również będziemy mogli bez problemów ją obliczyć itd. Ogólnie, przy obliczaniu kolejnej składowej szukanego wektora będziemy korzystali z wyznaczonych już poprzednio składowych.

Podobnie, jak w metodzie Jacobiego, odpowiednie przestawienie wierszy, tak aby na diagonali znajdowały się tylko elementy niezerowe, nie gwarantuje w ogólnym przypadku spełnienia warunku zbieżności

$$\rho(\mathbf{M}_{GS}) = \rho(-(\mathbf{D}+\mathbf{L})^{-1}\mathbf{U}) < 1. \tag{4.174}$$

Jeżeli jednak potrafimy uzasadnić, że dla danej macierzy $\mathbf{A}$ metoda Jacobiego jest zbieżna oraz macierz $\mathbf{M}_J$ ma nieujemne elementy, to zbieżna jest również metoda Gaussa–Seidla. Przy tym

$$\rho(\mathbf{M}_{GS}) < \rho(\mathbf{M}_J) < 1,$$

a więc w tym wypadku metoda Gaussa–Seidla jest zbieżna asymptotycznie szybciej niż metoda Jacobiego. Ponadto zbieżność metody jest gwarantowana, jeśli macierz $\mathbf{A}$ układu równań jest:

- symetryczna, dodatnio określona;
- silnie diagonalnie dominująca;
- silnie diagonalnie dominująca kolumnowo.

4.3.4 Analiza błędów zaokrągleń

Jeżeli w każdej iteracji zamiast wartości $\mathbf{M}\vec{x}^{(i)} + \vec{w}$ obliczamy

$$\mathbf{M}\vec{x}^{(i)} + \vec{w} + \vec{\delta}^{(i)}, \tag{4.175}$$

gdzie $\delta^{(i)}$ jest błędem zaokrągleń, to

$$\vec{x}^{(i+1)} = \mathbf{M}^{i+1}\vec{x}^{(0)} + \mathbf{M}^i\vec{w} + \ldots + \vec{w} + \vec{\delta}^{(i)} + \mathbf{M}\vec{\delta}(i-1) + \ldots + \mathbf{M}^i\vec{\delta}^{(0)}. \tag{4.176}$$

Zatem łączny błąd zaokrągleń wynosi

$$\vec{x}_{dok}^{(i+1)} - \vec{x}^{(i+1)} = \vec{\delta}^{(i)} + \mathbf{M}\vec{\delta}^{(i-1)} + \ldots + \mathbf{M}^i\vec{\delta}^{(0)}. \tag{4.177}$$

Jeżeli algorytm iteracyjny jest zbieżny i indeks iteracji jest dostatecznie duży, możemy przyjąć

$$\frac{1}{2}\|\vec{x}_{dok}\| < \|\vec{x}^{(j)}\| < 2\|\vec{x}_{dok}\|. \tag{4.178}$$

Stąd wynika, że jeżeli $\vec{x}_{dok} \neq 0$, to

$$\frac{\|\vec{x}_{dok}^{(i+1)} - \vec{x}^{(i+1)}\|}{\|\vec{x}^{(i+1)}\|} \leqslant \frac{2}{\|\vec{x}_{dok}\|}(\|\vec{\delta}^{(i)}\| + \|\mathbf{M}\| \cdot \|\vec{\delta}^{(i-1)}\| + \ldots + \|\mathbf{M}\|^i \cdot \|\vec{\delta}^{(0)}\|), \tag{4.179}$$

czyli

$$\frac{\|\vec{x}_{dok}^{(i+1)} - \vec{x}^{(i+1)}\|}{\|\vec{x}^{(i+1)}\|} \leqslant \frac{2\kappa}{\|\vec{x}_{dok}\|}(1 + \|\mathbf{M}\| + \ldots + \|\mathbf{M}\|^i), \tag{4.180}$$

gdzie κ to wspólne oszacowanie błędów $\vec{\delta}^{(j)}$, tzn.

$$\|\vec{\delta}^{(j)}\| < \kappa, \quad j = 0, 1, 2, \ldots, i. \tag{4.181}$$

Jeśli $\|\mathbf{M}\| < 1$, to

$$\frac{\|\vec{x}_{dok}^{(i+1)} - \vec{x}^{(i+1)}\|}{\|\vec{x}^{(i+1)}\|} < \frac{1}{1 - \|\mathbf{M}\|}\frac{2\kappa}{\|\vec{x}_{dok}\|}. \tag{4.182}$$

Gdy macierz układu jest macierzą silnie diagonalnie dominującą, można pokazać [50], że

$$\frac{\|\vec{x}_{dok}^{(i+1)} - \vec{x}^{(i+1)}\|_\infty}{\|\vec{x}^{(i+1)}\|_\infty} \leqslant \frac{1}{1 - \|\mathbf{M}_{GS}\|_\infty} \frac{12\alpha}{1-\alpha}, \tag{4.183}$$

gdzie

$$\alpha = \epsilon O(2n^2) \|\mathbf{D}\|_\infty \|\mathbf{D}^{-1}\|_\infty. \tag{4.184}$$

Z porównania oszacowania (4.183) z błędem eliminacji Gaussa wynika, że stosując metodę Gaussa–Seidla można zyskać na dokładności, jeżeli tylko wskaźnik $\|\mathbf{D}\|_\infty \|\mathbf{D}^{-1}\|_\infty$ jest mały w porównaniu ze wskaźnikiem uwarunkowania K_∞.

4.3.5 Nakłady obliczeń i warunki ich przerwania

W każdej iteracji, niezależnie od wyboru metody, wykonujemy około n^2 mnożeń (jeżeli macierz układu nie jest rzadka). Dla porównania, metody dokładne wymagają około $\frac{1}{3}n^3$ mnożeń do uzyskania rozwiązania. Zatem, aby metody dokładne i iteracyjne były porównywalne pod względem nakładu obliczeń, powinniśmy wykonać tylko około n iteracji. Jednak już proste przykłady pokazują, że liczba iteracji musi być dużo większa niż n, aby dokładność była zadowalająca, dlatego metody iteracyjne w przypadku ogólnym są nieefektywne.

Przykład Układ

$$\begin{pmatrix} 1 & \frac{3}{4} \\ \frac{3}{4} & 1 \end{pmatrix} \vec{x} = \begin{pmatrix} 448 \\ 448 \end{pmatrix}$$

ma rozwiązanie $x = (256, 256)^{\mathrm{T}}$. Stosując np. eliminację Gaussa, musimy wykonać 6 mnożeń, aby otrzymać wynik dokładny. Natomiast, jeżeli zastosujemy metodę Gaussa–Seidla, po ośmiu iteracjach (32 mnożenia)

$$\|x^{(8)} - \vec{x}_{dok}\| > 0,1.$$

Niezbędną do uzyskania zaplanowanej dokładności liczbę iteracji trudno jest przewidzieć. Dlatego na ogół w praktyce nie zakłada się konkretnej liczby iteracji z góry (jednak wbudowanie pewnego ograniczenia w program jest konieczne, aby uniknąć ewentualnych zapętleń lub braku zbieżności), lecz stosuje się testy na przerwanie obliczeń (tzw. testy stopu). Najczęściej są to testy typu

$$\|\vec{x}^{(i+1)} - \vec{x}^{(i)}\| < \Delta, \tag{4.185}$$

lub

$$\frac{1}{\|\vec{b}\|} \|\mathbf{A}\vec{x}^{(i+1)} - \vec{b}\| < \Delta, \tag{4.186}$$

gdzie Δ to żądana dokładność. Stosując powyższe testy należy być świadomym związanych z nimi ograniczeń. Jeżeli np. norma macierzy $\mathbf{A}$ jest mała, to wartość reszty

$$\|\mathbf{A}\vec{x}^{(i+1)} - \vec{b}\| = \|\mathbf{A}(\vec{x}^{(i+1)} - \vec{x}_{dok})\| \leqslant \|\mathbf{A}\| \cdot \|\vec{x}^{(i+1)} - \vec{x}_{dok}\| \tag{4.187}$$

może być mała, mimo dużego odchylenia wektora $\vec{x}^{(i+1)}$ od rozwiązania dokładnego $\vec{x}_{dok}$. Podobnie, ponieważ zachodzi

$$\|\vec{x}^{(i+1)} - \vec{x}^{(i)}\| = \|\vec{e}^{(i+1)} - \vec{e}^{(i)}\| = \|\mathbf{M}\vec{e}^{(i)} - \vec{e}^{(i)}\| = \|(\mathbf{M} - \mathbf{I})\vec{e}^{(i)}\|, \tag{4.188}$$

gdy norma macierzy $\mathbf{M}-\mathbf{I}$ jest mała, wektory $\vec{x}^{(i+1)}$ i $\vec{x}^{(i)}$ mogą się mało różnić, mimo że błąd $\vec{e}^{(i)}$ jest duży. Ponieważ

$$\|\mathbf{I}-\mathbf{M}\| \geqslant \rho(\mathbf{I}-\mathbf{M}) = 1-\rho(\mathbf{M}), \tag{4.189}$$

z sytuacją taką mamy do czynienia, gdy $\rho(\mathbf{M})$ jest bliskie 1 (wolna zbieżność algorytmu). Wówczas lepiej jest stosować test (4.186).

Testy (4.185)-(4.186) mogą się również okazać mało przydatne z powodu błędów zaokrągleń. Dlatego wektory reszt należy zawsze liczyć z dużą dokładnością.

4.4 Niedookreślone układy równań ($m < n$)

Układ równań liniowych, w którym liczba równań m jest mniejsza od liczby niewiadomych n, nazywamy układem niedookreślonym. Układy tego typu są dość często spotykane w praktyce (np. w zagadnieniach optymalizacji). Mimo to nie są one często dyskutowane w literaturze poświęconej metodom numerycznym (do wyjątków należą [15, 35]).

Układy niedookreślone nigdy nie są rozwiązywalne jednoznacznie. Jeżeli wektor wyrazów wolnych $\vec{b}$ należy do przestrzeni rozpinanej przez kolumny macierzy $\mathbf{A}$, wówczas układ

$$\mathbf{A}\vec{x} = \vec{b} \tag{4.190}$$

będzie miał nieskończenie wiele rozwiązań. W przeciwnym wypadku rozwiązań tych nie będzie wcale. Stąd wynika natychmiast następujące twierdzenie:

Twierdzenie 4.4.1 *Niedookreślony układ jednorodny ma zawsze nieskończenie wiele rozwiązań.*

Jeżeli rząd macierzy $\mathbf{A}$ jest równy liczbie równań, wówczas $\vec{b}$ zawsze będzie należał do przestrzeni rozpinanej przez $\mathbf{A}$. Innymi słowy, układ (4.190) będzie rozwiązywalny. Ogólne rozwiązanie takiego układu zapisze się w postaci:

$$\vec{x} = \vec{x}_p + \vec{x}_N, \tag{4.191}$$

gdzie $\vec{x}_p$ jest specjalnym rozwiązaniem równania (4.190), a $\vec{x}_N$ należy do jądra przekształcenia liniowego $\mathbf{A}$,

$$\mathbf{A}\vec{x}_N = 0. \tag{4.192}$$

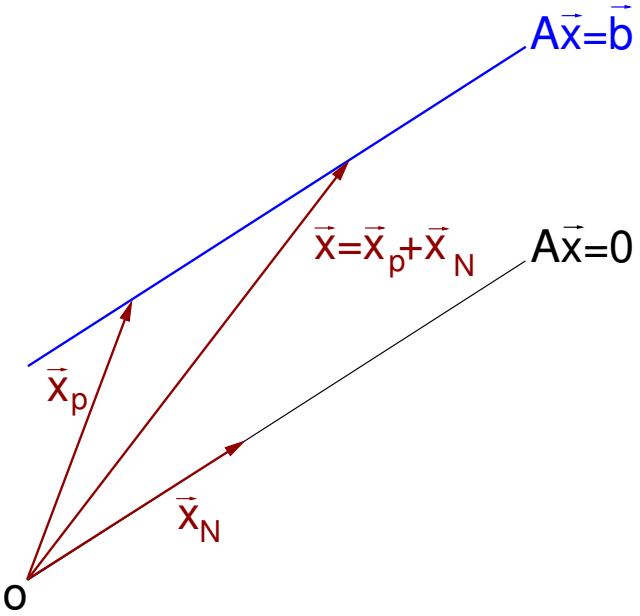
Na rys. 4.1 przedstawiona jest graficzna interpretacja rozwiązania (4.191) dla $m = 1$ i $n = 2$. Jak wiadomo, równanie z dwoma niewiadomymi definiuje prostą na płaszczyźnie. Każdy punkt na tej prostej jest wówczas rozwiązaniem naszego układu równań.

Nasuwa się pytanie, jak znaleźć rozwiązanie szczególne $\vec{x}_p$. Otóż zachodzi następujące twierdzenie:

Twierdzenie 4.4.2 *Jeżeli macierz $\mathbf{A} \in \mathbf{R}^{m\times n}$ ma rząd m, układ $\mathbf{A}\vec{x} = \vec{b}$ jest zawsze rozwiązywalny. Dla każdego $\vec{b}$ istnieje wówczas nieskończenie wiele rozwiązań, z których*

$$\vec{x}_p = \mathbf{A}^{\mathrm{T}}\left(\mathbf{A}\mathbf{A}^{\mathrm{T}}\right)^{-1}\vec{b} \tag{4.193}$$

jest tym o najmniejszej normie. Macierz $\mathbf{A}^{\mathrm{T}}\left(\mathbf{A}\mathbf{A}^{\mathrm{T}}\right)^{-1}$ nazywana jest przy tym macierzą pseudoodwrotną macierzy $\mathbf{A}$.



Rysunek 4.1: Graficzna interpretacja rozwiązania (4.191) dla $m = 1$ i $n = 2$.

Dowód Dla każdego $\vec{x}$ zachodzi

$$\vec{x}^{\mathrm{T}}\mathbf{A}\mathbf{A}^{\mathrm{T}}\vec{x} = \left(\mathbf{A}^{\mathrm{T}}\vec{x}\right)^{\mathrm{T}}\left(\mathbf{A}^{\mathrm{T}}\vec{x}\right) = \|\mathbf{A}^{\mathrm{T}}\vec{x}\|^2 \geqslant 0. \tag{4.194}$$

Ponadto, jeśli rank$\mathbf{A} = m$, to $\|\mathbf{A}^{\mathrm{T}}\vec{x}\| = 0$ wtedy i tylko wtedy, gdy $\vec{x} = 0$. To znaczy, że macierz $\mathbf{A}\mathbf{A}^{\mathrm{T}}$ jest dodatnio określona, a co za tym idzie - nieosobliwa. Ponieważ

$$\mathbf{A}\vec{x}_p = \mathbf{A}\mathbf{A}^{\mathrm{T}}\left(\mathbf{A}\mathbf{A}^{\mathrm{T}}\right)^{-1}\vec{b} = \vec{b}, \tag{4.195}$$

więc x_p rzeczywiście jest rozwiązaniem równania $\mathbf{A}\vec{x} = \vec{b}$. Pozostaje nam pokazać, że każde inne rozwiązanie ma normę większą od $\|\vec{x}_p\|$.

Niech $\vec{x}$ będzie innym rozwiązaniem naszego układu. Wówczas

$$\|\vec{x}\|^2 = \|\vec{x}_p + (\vec{x} - \vec{x}_p)\|^2 = \|\vec{x}_p\|^2 + \|\vec{x} - \vec{x}_p\|^2 + 2\vec{x}_p^{\mathrm{T}}(\vec{x} - \vec{x}_p). \tag{4.196}$$

Ponieważ z założenia $\mathbf{A}\vec{x}_p = \mathbf{A}\vec{x}$, trzeci wyraz w powyższym równaniu jest równy zero:

$$\vec{x}_p^{\mathrm{T}}(\vec{x} - \vec{x}_p) = \left[\mathbf{A}^{\mathrm{T}}(\mathbf{A}\mathbf{A}^{\mathrm{T}})^{-1}\vec{b}\right]^{\mathrm{T}}(\vec{x} - \vec{x}_p) = \vec{b}^{\mathrm{T}}(\mathbf{A}\mathbf{A}^{\mathrm{T}})^{-1}\mathbf{A}(\vec{x} - \vec{x}_p) = 0. \tag{4.197}$$

Stąd

$$\|\vec{x}\|^2 = \|\vec{x}_p\|^2 + \|\vec{x} - \vec{x}_p\|^2 \geqslant \|\vec{x}_p\|^2, \tag{4.198}$$

przy czym równość zachodzi tylko dla $\vec{x} = \vec{x}_p$. ∎

Przykład Rozważmy układ

$$\begin{pmatrix} 1 & 2 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = 3, \tag{4.199}$$

czyli

$$x_1 + 2x_2 = 3, \quad x_2 = -\frac{1}{2}x_1 + \frac{3}{2}. \tag{4.200}$$

Dowolne z tych dwóch wyrażeń jest rozwiązaniem układu (4.199). Rozwiązaniem o najmniejszej normie będzie

$$\vec{x}_p = \mathbf{A}^{\mathrm{T}} \left(\mathbf{A}\mathbf{A}^{\mathrm{T}}\right)^{-1} \vec{b} = \begin{pmatrix} 0,6 \\ 1,2 \end{pmatrix}. \tag{4.201}$$

Wektory należące do jądra przekształcenia $\mathbf{A}$ będą miały postać

$$\mathbf{A}\vec{x}_N = 0 \quad \rightarrow \quad x_{N2} = -\frac{1}{2}x_{N1}, \tag{4.202}$$

więc ogólne rozwiązanie jest następujące:

$$\vec{x} = \begin{pmatrix} 0,6 \\ 1,2 \end{pmatrix} + \alpha \begin{pmatrix} 1 \\ -0,5 \end{pmatrix}, \tag{4.203}$$

gdzie α jest dowolną liczbą rzeczywistą.

Istnieje kilka sposobów na wyznaczenie macierzy pseudoodwrotnej do macierzy $\mathbf{A}$. Możemy np. użyć rozkładu Cholesky'ego, SVD lub QR.

Rozkład Cholesky'ego

Ponieważ $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ jest macierzą symetryczną i dodatnio określoną (patrz dowód tw. 4.4.2), możemy skorzystać z rozkładu Cholesky'ego (par. 4.2.11) i rozłożyć ją na iloczyn dwóch macierzy trójkątnych,

$$\mathbf{A}^{\mathrm{T}}\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathrm{T}}. \tag{4.204}$$

Teraz wystarczy rozwiązać układy równań

$$\begin{aligned} \mathbf{L}\vec{w} &= \vec{b}, \\ \mathbf{L}^{\mathrm{T}}\vec{z} &= \vec{w}, \end{aligned} \tag{4.205}$$

i na tej podstawie wyliczyć $\vec{x}_p$,

$$\vec{x}_p = \mathbf{A}^{\mathrm{T}}\vec{z}. \tag{4.206}$$

Rozkład SVD

Rozkład macierzy na wartości osobliwe również może okazać się pomocny przy rozwiązywaniu niedookreślonych układów równań. Z równości

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}} \tag{4.207}$$

wynika mianowicie

$$\begin{aligned} \vec{x}_p &= \mathbf{A}^{\mathrm{T}} \left(\mathbf{A}\mathbf{A}^{\mathrm{T}}\right)^{-1} \vec{b} = \mathbf{V}\mathbf{\Sigma}\mathbf{U}^{\mathrm{T}} \left(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}\mathbf{V}\mathbf{\Sigma}\mathbf{U}^{\mathrm{T}}\right)^{-1} \vec{b} \\ &= \mathbf{V}\mathbf{\Sigma}\mathbf{U}^{\mathrm{T}} \left(\mathbf{U}\mathbf{\Sigma}\mathbf{\Sigma}\mathbf{U}^{\mathrm{T}}\right)^{-1} \vec{b} = \mathbf{V}\mathbf{\Sigma}\mathbf{U}^{\mathrm{T}} \left(\mathbf{U}^{\mathrm{T}}\right)^{-1} \mathbf{\Sigma}^{-1}\mathbf{\Sigma}^{-1}\mathbf{U}^{-1}\vec{b} \\ &= \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^{\mathrm{T}}\vec{b}. \end{aligned} \tag{4.208}$$

Zapisując rozkład (4.207) w postaci

$$\mathbf{A} = \sum_{i=1}^{r} \sigma_i \vec{u}_i \vec{v}_i^{\mathrm{T}}, \quad r = \mathrm{rank}\mathbf{A}, \tag{4.209}$$

gdzie $\vec{u}_i$ i $\vec{v}_i$ to kolumny macierzy $\mathbf{U}$ i $\mathbf{V}$, otrzymamy

$$\vec{x}_p = \sum_{i=1}^{r} \frac{\vec{u}_i^{\mathrm{T}} \vec{b}}{\sigma_i} \vec{v}_i. \tag{4.210}$$

Rozkład QR

Jeżeli dysponujemy rozkładem QR macierzy $\mathbf{A}^{\mathrm{T}}$,

$$\mathbf{A}^{\mathrm{T}} = \mathbf{QR}, \tag{4.211}$$

gdzie $\mathbf{R}$ jest macierzą trójkątną górną o dodatnich elementach, a $\mathbf{Q}$ macierzą ortogonalną, wówczas wyznaczymy $\vec{x}_p$ równie prosto. Zachodzi mianowicie

$$\begin{aligned} \vec{x}_p &= \mathbf{A}^{\mathrm{T}} \left(\mathbf{A}\mathbf{A}^{\mathrm{T}}\right)^{-1} \vec{b} = \mathbf{QR} \left(\mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{QR}\right)^{-1} \vec{b} \\ &= \mathbf{QR} \left(\mathbf{R}^{\mathrm{T}}\mathbf{R}\right)^{-1} \vec{b} = \mathbf{QR}\mathbf{R}^{-1} \left(\mathbf{R}^{\mathrm{T}}\right)^{-1} \vec{b} \\ &= \mathbf{Q} \left(\mathbf{R}^{\mathrm{T}}\right)^{-1} \vec{b}. \end{aligned} \tag{4.212}$$

Aby wyliczyć $\vec{x}_p$, musimy wyznaczyć $\left(\mathbf{R}^{\mathrm{T}}\right)^{-1} \vec{b}$. Ale to nic innego, jak rozwiązanie równania trójkątnego

$$\mathbf{R}^{\mathrm{T}} \vec{z} = \vec{b}. \tag{4.213}$$

Ostatecznie otrzymamy

$$\vec{x}_p = \mathbf{Q}\vec{z}. \tag{4.214}$$

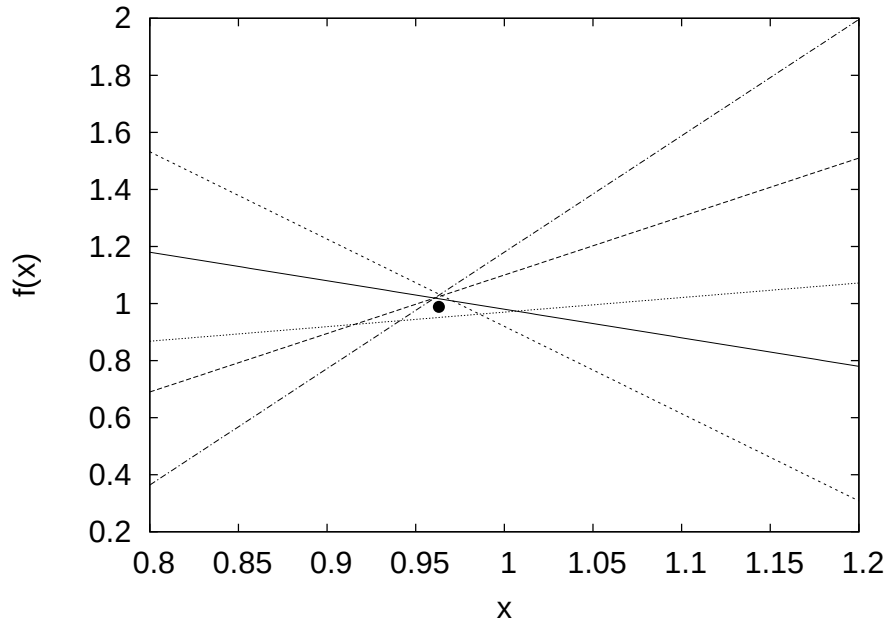
4.5 Nadokreślone układy równań ($m > n$)

Jeżeli równań (m) jest więcej niż niewiadomych (n), układ nazywamy nadokreślonym. W zależności od wektora wyrazów wolnych może on nie mieć wcale rozwiązań, mieć ich nieskończoną liczbę lub tylko jedno rozwiązanie jednoznaczne. W praktyce najczęściej będziemy mieć do czynienia z sytuacją, w której dokładne rozwiązanie takiego układu nie istnieje, ale możliwe jest na ogół znalezienie rozwiązania przybliżonego. Przykładem takiego zagadnienia jest regresja liniowa, często stosowana przy analizie danych z pomiarów fizycznych.

Równania $\mathbf{A}\vec{x} = \vec{b}$ dla macierzy $\mathbf{A} \in \mathbf{R}^{m \times n}$ przy $m > n$ nie można rozwiązać uniwersalnie, ponieważ rząd tej macierzy siłą rzeczy jest mniejszy od m. Rozwiązanie dokładne nie istnieje w ogóle, gdy wektor $\vec{b}$ nie należy do przestrzeni rozpinanej przez kolumny macierzy układu. W tym przypadku zadany układ można potraktować jak zadanie aproksymacyjne[3] i poszukać takiego $\vec{x}$, który zminimalizuje kwadrat normy wektora błędu,

$$\vec{e} = \mathbf{A}\vec{x} - \vec{b}. \tag{4.215}$$

[3] Więcej szczegółów w rozdziale 6.



Rysunek 4.2: Graficzna interpretacja układu (4.219). Symbol • reprezentuje rozwiązanie przybliżone.

Takie przybliżone rozwiązanie może okazać się bardzo użyteczne w wielu praktycznych zagadnieniach. A sama metoda poszukiwania $\vec{x}$ w ten sposób nazywa sie metodą najmniejszych kwadratów (ang. *least-square error*).

Szukamy zatem minimum wyrażenia

$$J = \frac{1}{2}\|\vec{e}\|_2^2 = \frac{1}{2}\|\mathbf{A}\vec{x} - \vec{b}\|_2^2 = \frac{1}{2}\left(\mathbf{A}\vec{x} - \vec{b}\right)^{\mathrm{T}}\left(\mathbf{A}\vec{x} - \vec{b}\right). \tag{4.216}$$

Z warunku na istnienie minimum,

$$\frac{\partial}{\partial \vec{x}} J = \mathbf{A}^{\mathrm{T}}\left(\mathbf{A}\vec{x} - \vec{b}\right) = 0, \tag{4.217}$$

znajdziemy

$$\vec{x}_p = \left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)^{-1}\mathbf{A}^{\mathrm{T}}\vec{b}. \tag{4.218}$$

Podobnie, jak w przypadku układów niedookreślonych, do obliczenia macierzy pseudoodwrotnej do macierzy $\mathbf{A}$ możemy wykorzystać rozkłady SVD lub QR macierzy $\mathbf{A}$.

Przykład Rozważmy układ

$$\begin{aligned} x + y &= 1,98 \\ 2,05 * x - y &= 0,95 \\ 3,06 * x + y &= 3,98 \\ -1,02 * x + 2 * y &= 0,92 \\ 4,08 * x - y &= 2,90 \end{aligned} \tag{4.219}$$

Jego rozwiązanie ma prostą interpretację geometryczną - to punkt przecięcia prostych zdefiniowanych poszczególnymi równaniami. Jak widać na rysunku 4.2, nie istnieje dokładne rozwiązanie tego układu. Natomiast jego rozwiązanie przybliżone na mocy (4.218) wynosi

$$\vec{x}_p = \begin{pmatrix} 0,963101 \\ 0,988543 \end{pmatrix} \tag{4.220}$$

z błędem

$$\|\mathbf{A}\vec{x}_p - \vec{b}\|_2 = 0,10636. \tag{4.221}$$

4.6 Funkcje biblioteczne - przykłady

W związku z rolą, jaką odgrywają układy równań liniowych w innych dziedzinach analizy numerycznej i w rozwiązywaniu praktycznych zagadnień, gotowe implementacje większości metod omówionych w tym rozdziale (i kilku innych, z omówienia których niestety musieliśmy zrezygnować) znajdziemy w wielu bibliotekach i środowiskach do obliczeń numerycznych.

C/C++

Korzystając z biblioteki GSL [12] normę $\|\cdot\|_2$ wektora $\vec{x}$ (w podwójnej precyzji) policzymy funkcją

*double gsl_blas_dnrm2(const gsl_vector *)*

Aby można było jej użyć, do programu należy dodać plik nagłówkowy `gsl_blas.h`:

```
#include <iostream>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_blas.h>

int main()
{
     int n=5;
     gsl_vector * v = gsl_vector_alloc (n);

     for (int i = 0; i < n; i++)
     {
        gsl_vector_set (v, i, 1.23 + i);
     }

     double norma = gsl_blas_dnrm2 (v);
     std::cout<< "Norma: " << norma << "\n";
}
```

Po skompilowaniu programu,

```
g++ vecnorm.cpp -lgsl -lgslcblas
```

na ekranie uzyskamy następujący wynik:

```
Norma: 7.88445
```

Rozkładu LU macierzy **A** dokonamy za pomocą

```
int gsl_linalg_LU_decomp(gsl_matrix * A, gsl_permutation * p,
                         int * signum)
```

Funkcja wykorzystuje eliminację Gaussa z częściowym wyborem elementu podstawowego. Orginalna macierz **A** jest niszczona - jej elementy nadpisywane są elementami macierzy rozkładu (patrz wzór (4.89)).

Dysponując już rozkładem LU, możemy rozwiązać układ równań dużo mniejszym nakładem obliczeń,

```
int gsl_linalg_LU_solve(const gsl_matrix * LU, gsl_permutation
                * p, const gsl_vector * b, gsl_vector * x)
```

a następnie poprawić rozwiązanie iteracyjnie,

```
int gsl_linalg_LU_refine(const gsl_matrix * A, const gsl_matrix
                * LU, gsl_permutation * p, const gsl_vector * b,
                gsl_vector * x, gsl_vector * residual)
```

W zmiennej `residual` przechowywany jest początkowy wektor reszt $\vec{r} = \mathbf{A}\vec{x} - \vec{b}$.

Przykład Rozważmy raz jeszcze układ (4.52),

$$\begin{pmatrix} 0 & 2 & 2 \\ 3 & 3 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}. \tag{4.222}$$

Do znalezienia jego rozwiązania z wykorzystaniem rozkładu LU i biblioteki GSL wystarczy prosty program:

```
#include <iostream>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_linalg.h>


int main()
{
     int n=3;
     double A_data[] = {0.0,2.0,2.0,
                        3.0,3.0,0.0,
                        1.0,0.0,1.0};

     double b_data[] = {1.0,3.0,2.0};

     gsl_matrix_view A
               = gsl_matrix_view_array (A_data, n, n);
     gsl_vector_view b
               = gsl_vector_view_array (b_data, n);
     gsl_vector *x = gsl_vector_alloc (n);
     int znak;
     gsl_permutation * p = gsl_permutation_alloc (n);

     gsl_linalg_LU_decomp (&A.matrix, p, &znak);
     gsl_linalg_LU_solve (&A.matrix, p, &b.vector, x);

     std::cout<<"x = \n";
     gsl_vector_fprintf (stdout, x, "%g");
     gsl_permutation_free (p);
}
```

Efekt jego działania jest następujący:

```
x =
1.25
-0.25
0.75
```

Rozkładu macierzy $\mathbf{A} \in \mathbf{R}^{m \times n}$ na iloczyn **QR** dokonamy za pomocą

```
int gsl_linalg_QR_decomp(gsl_matrix * A, gsl_vector * tau)
```

Funkcja ta wykorzystuje algorytm Householdera [48]. Wektor `tau` musi mieć wymiar $k = \min(m, n)$. Dysponując rozkładem QR macierzy, możemy np. rozwiązać określony,

```
int gsl_linalg_QR_solve(const gsl_matrix * QR, const
                gsl_vector * tau, const gsl_vector * b,
                gsl_vector * x)
```

lub nadokreślony układ równań,

```
int gsl_linalg_QR_lssolve(const gsl_matrix * QR, const
                gsl_vector * tau,  const gsl_vector * b,
                gsl_vector * x, gsl_vector * residual)
```

Przykład Do rozwiązania układu (4.219)

$$\begin{aligned} x + y &= 1,98 \\ 2,05 * x - y &= 0,95 \\ 3,06 * x + y &= 3,98 \\ -1,02 * x + 2 * y &= 0,92 \\ 4,08 * x - y &= 2,90 \end{aligned} \tag{4.223}$$

posłużymy się programem:

```
#include <iostream>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_linalg.h>


int main()
{
     int n=5;
     int m=2;
     double A_data[] = {1.0,1.0,
                        2.05,-1.0,
                        3.06,1.0,
                        -1.02,2.0,
                        4.08,-1.0};

     double b_data[] = {1.98,0.95,3.98,0.92,2.90};
     gsl_matrix_view A
          = gsl_matrix_view_array (A_data, n, m);
     gsl_vector_view b
          = gsl_vector_view_array (b_data, n);
```

```
    gsl_vector *tau = gsl_vector_alloc (m);
    gsl_vector *res = gsl_vector_alloc (n);
    gsl_vector *x = gsl_vector_alloc (m);

    gsl_linalg_QR_decomp (&A.matrix, tau);
    gsl_linalg_QR_lssolve (&A.matrix, tau, &b.vector,
                           x, res);

    std::cout<<"x = \n";
    gsl_vector_fprintf (stdout, x, "%g");
}
```

Wynik jest następujący:

```
x =
0.963101
0.988543
```

Rozkład Cholesky'ego macierzy **A** wyznaczymy poleceniem

```
int gsl_linalg_cholesky_decomp (gsl_matrix * A)
```

a rozwiązania odpowiedniego układu poszukamy przy pomocy

```
int gsl_linalg_cholesky_solve (const gsl_matrix * cholesky,
                const gsl_vector * b, gsl_vector * x)
```

Podobnie, do rozkładu macierzy na wartości osobliwe służy

```
int gsl_linalg_SV_decomp (gsl_matrix * A, gsl_matrix * V,
                gsl_vector * S, gsl_vector * work)
```

Funkcja ta wykorzystuje algorytm Goluba-Reinscha [48]. Dysponując rozkładem SVD macierzy **A**, odpowiadający jej układ rozwiążemy funkcją

```
int gsl_linalg_SV_solve (gsl_matrix * U, gsl_matrix * V,
                gsl_vector * S, const gsl_vector * b,
                gsl_vector * x)
```

Fortran

LAPACK [13] (ang. *Linear Algebra PACKage*) to jedna z bardziej popularnych bibliotek numerycznych przeznaczonych dla Fortrana. Oferuje ona procedury do rozwiązywania układów równań liniowych i zagadnień na wartości własne. Pozwala również faktoryzować macierze na wiele różnych sposobów.

Do rozwiązania określonego układu równań liniowych użyjemy np. procedury `DGESV`[4],

DGESV(N,NRHS,A,LDA,IPIV,B,LDB,INFO)

gdzie `N` określa wymiar macierzy układu `A`, `NRHS` to liczba wektorów wyrazów wolnych, czyli kolumn w macierzy `B` (możliwe jest rozwiązywanie kilku układów równań jednocześnie). Procedura wykorzystuje eliminację Gaussa z wyborem elementu podstawowego. W macierzy permutacji `IPIV` zakodowana jest

[4]Litera „D" na początku nazwy oznacza, że obliczenia wykonane zostaną w podwójnej precyzji. Gdybyśmy byli zainteresowani pojedynczą precyzją, wystarczy że we wszystkich omawianych tu procedurach zmienimy „D" na „S".

kolejność kolumn. `LDA` i `LDB` to wiodące wymiary macierzy `A` i `B`. Przy tym $LDA, LDB \geqslant \max(1, N)$. Macierz wyrazów wolnych `B` jest niszczona w trakcie obliczeń. Do niej wpisywane jest rozwiązanie układu. Wartość zmiennej `INFO` informuje nas o tym, czy obliczenia zakończyły się powodzeniem ($INFO = 0$), czy też nie.

Przykład Rozważmy raz jeszcze układ (4.52),

$$\begin{pmatrix} 0 & 2 & 2 \\ 3 & 3 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}. \tag{4.224}$$

Wykorzystanie procedury `DGESV` do jego rozwiązania ilustruje następujący program:

```
program url1
  implicit none
  integer, parameter :: N=3
  integer :: i,j
  real(kind=8), dimension (N) :: b
  real(kind=8), dimension (N,N) :: A
  integer, dimension (N,N) :: P
  integer :: info
  data b /1.d0,3.d0,2.d0/, &
       ((A(i,j), j=1,N),i=1,N) /0.d0,2.d0,&
       2.d0,3.d0,3.d0,0.d0,1.d0,0.d0,1.d0/

  call DGESV(N,1,A,N,P,b,N,info)
  write(6,'(a2)') "x="
  write(6,'(f10.6)') b
end program url1
```

Po skompilowaniu programu,

```
ifort url_ex.f90 -llapack
```

lub

```
gfortran url_ex.f90 -llapack
```

i jego uruchomieniu otrzymamy

```
x=
  1.250000
 -0.250000
  0.750000
```

W przypadku niedookreślonych lub nadokreślonych układów równań posłużymy się procedurą

DGELS(TRANS,M,N,NRHS,A,LDA,B,LDB,WORK,LWORK,INFO)

W porównaniu z `DGESV` pojawiło się tu kilka nowych argumentów. Za pomocą `TRANS` określamy, czy interesuje nas układ $\mathbf{A}\vec{x} = \vec{b}$, czy raczej $\mathbf{A}^{\mathrm{T}}\vec{x} = \vec{b}$. Ponieważ macierz układu jest teraz prostokątna ($\mathbf{A} \in \mathbf{R}^{M \times N}$), podajemy oba jej wymiary, `M` i `N`. Tablica `WORK` o wymiarze `LWORK` stanowi przestrzeń roboczą procedury `DGELS`. Przy tym

$$LWORK \geqslant \max\left(1, \min(M, N) + \max(\min(M, N), NRHS)\right).$$

Podobnie, jak poprzednio, wynik wpisywany jest do macierzy `B`.

Przykład Do rozwiązania układu (4.219)

$$\begin{aligned} x + y &= 1,98 \\ 2,05 * x - y &= 0,95 \\ 3,06 * x + y &= 3,98 \\ -1,02 * x + 2 * y &= 0,92 \\ 4,08 * x - y &= 2,90 \end{aligned} \tag{4.225}$$

z wykorzystaniem biblioteki LAPACK wystarczy program:

```
program url2
  implicit none
  integer, parameter :: M=5
  integer, parameter :: N=2
  integer, parameter :: LWORK=5
  integer :: i,j
  real(kind=8), dimension (M) :: b
  real(kind=8), dimension (M,N) :: A
  real(kind=8), dimension (LWORK) :: WORK
  integer :: info
  data b /1.98,0.95,3.98,0.92,2.90/, &
       ((A(i,j), j=1,N),i=1,M) /1.0,1.0,2.05,&
       -1.0,3.06,1.0,-1.02,2.0,4.08,-1.0/

  call DGELS('N',M,N,1,A,M,b,M,WORK,LWORK,info)
  write(6,'(a2)') "x="
  write(6,'(f10.6)') (b(i),i=1,N)
end program url2
```

Wynik działania programu jest następujący:

```
x=
  0.963101
  0.988543
```

Rozkład LU macierzy **A** policzymy za pomocą `DGETRF`,

DGETRF(M, N, A, LDA, IPIV, INFO)

Wynik działania tej procedury wykorzystywany jest przez `DGETRI`,

DGETRI(N, A, LDA, IPIV, WORK, LWORK, INFO)

do znalezienia macierzy odwrotnej do macierzy **A**.

Przykład Rozważmy macierz układu (4.52),

$$\mathbf{A} = \begin{pmatrix} 0 & 2 & 2 \\ 3 & 3 & 0 \\ 1 & 0 & 1 \end{pmatrix}. \tag{4.226}$$

Macierz odrotną do **A** znajdziemy za pomocą:

```
program url3
  implicit none
  integer, parameter :: N=3
  integer :: i,j
  real(kind=8), dimension (N) :: WORK
  real(kind=8), dimension (N,N) :: A,D
  integer, dimension (N,N) :: P
  integer :: info
  data ((A(i,j), j=1,N),i=1,N) /0.d0,2.d0,&
           2.d0,3.d0,3.d0,0.d0,1.d0,0.d0,1.d0/

  D=A
  call DGETRF(N,N,A,N,P,info)
  call DGETRI(N,A,N,P,WORK,N,info)
  write(6,'(a6)') "A^{-1}="
  write(6,'(3f10.6)') ((A(i,j),j=1,N),i=1,N)
  write(6,'(a9)') "A*A^{-1}="
  write(6,'(3f10.6)') matmul(D,A)
end program url3
```

Wynik działania programu jest następujący:

```
A^{-1}
 -0.250000  0.166667  0.500000
  0.250000  0.166667 -0.500000
  0.250000 -0.166667  0.500000
A*A^{-1}=
  1.000000  0.000000  0.000000
  0.000000  1.000000  0.000000
  0.000000  0.000000  1.000000
```

Analizując kod powyższego programu warto zwrócić uwagę na dwie z wielu własności Fortrana 90, które czynią go tak wygodnym narzędziem w obliczeniach numerycznych. Przede wszystkim, macierze można przypisać innym macierzom przy pomocy operatora `=`. Po drugie, do mnożenia macierzowego służy wbudowana funkcja `matmul`.

Korzystając z LAPACKA, mamy oczywiście również do dyspozycji funkcje rozkładające macierz na wartości osobliwe lub iloczyn **QR**. Są to odpowiednio `DGESVD` i `DGEQP3`. Ich wywołanie jest bardzo podobne do omówionych już przykładów.

Python

Większość funkcji potrzebnych do rozwiązywania układów równań liniowych znajdziemy w module SciPy [16]. I tak, układ z macierzą kwadratową **A** rozwiążemy poleceniem `linalg.solve`[5],

```
>>> A=scipy.mat('[0 2 2;3 3 0;1 0 1]')
>>> b=scipy.mat('[1;3;2]')
>>> scipy.linalg.solve(A,b)
array([[ 1.25],
       [-0.25],
```

[5]Do wykonania dowolnego przykładu z tego paragrafu konieczne jest wykonanie polecenia `import scipy` w interpreterze Pythona.

```
       [ 0.75]])
```

które używa procedury `gesv` z biblioteki Lapack [13].

Macierz odwrotną do **A** wyznaczymy na dwa sposoby. Jeżeli macierz wyjściowa dana jest jako zwykła tablica, mamy do dyspozycji funkcję `linalg.inv`,

```
>>> A=[[0,2,2],[3,3,0],[1,0,1]]
>>> scipy.linalg.inv(A)
array([[-0.25       ,  0.16666667,  0.5       ],
       [ 0.25       ,  0.16666667, -0.5       ],
       [ 0.25       , -0.16666667,  0.5       ]])
```

Dodatkowo na obiektach `Matrix` możemy wykonać metodę `I`,

```
>>> B=scipy.mat(A)
>>> B.I
Matrix([[-0.25       ,  0.16666667,  0.5       ],
       [ 0.25       ,  0.16666667, -0.5       ],
       [ 0.25       , -0.16666667,  0.5       ]])
```

Jeżeli planujemy dużo operacji na macierzach, zalecane jest przechowywać je właśnie w formie obiektów `Matrix`. Do przekształcenia tablicy w macierz służy zastosowane powyżej polecenie `mat`.

Teoretycznie dysponując macierzą odwrotną możemy również rozwiązać układ równań liniowych,

```
>>> B.I*b
Matrix([[ 1.25],
       [-0.25],
       [ 0.75]])
```

Jednak ze względu na nakład obliczeń i niestabilność numeryczną lepiej jest stosować `linalg.solve`.

Funkcja `linalg.inv` korzysta z rozkładu LU macierzy. Gdyby interesował nas sam rozkład, znajdziemy go przy pomocy `linalg.lu`,

```
>>> P,L,U=scipy.linalg.lu(B)
>>> print P
[[ 0.  1.  0.]
 [ 1.  0.  0.]
 [ 0.  0.  1.]]
>>> print L
[[ 1.          0.          0.        ]
 [ 0.          1.          0.        ]
 [ 0.33333333 -0.5         1.        ]]
>>> print U
[[ 3.  3.  0.]
 [ 0.  2.  2.]
 [ 0.  0.  2.]]
```

Jeżeli dysponujemy już rozkładem LU macierzy układu, możemy rozwiązać go niewielkim nakładem obliczeń

```
>>> LU,P = scipy.linalg.lu_factor(B)
>>> scipy.linalg.lu_solve((LU,P),b)
array([[ 1.25],
       [-0.25],
       [ 0.75]])
```

Oczywiście, SciPy oferuje również funkcję do wyznaczenia pozostałych omawianych przez nas rozkładów: QR, SVD i Cholesky'ego. Będą to odpowiednio `linalg.qr`, `linalg.svd` i `linalg.cholesky`.

Macierz pseudoodrotną do macierzy **A** znajdziemy przy pomocy `linalg.pinv` lub `linalg.pinv2`. Polecenia różnią się zaimplementowanym algorytmem. Pierwsze korzysta z metody najmniejszych kwadratów, drugie - z rozkładu SVD,

```
>>> b=scipy.mat('[3]')
>>> A=scipy.mat('[1 2]')
>>> AI=scipy.linalg.pinv2(A)
>>> AI*b
Matrix([[ 0.6],
        [ 1.2]])
>>> c=scipy.mat('[1.98; 0.95; 3.98; 0.92;2.90]')
>>> B=scipy.mat('[1 1;2.05 -1;3.06 1;-1.02 2;4.08 -1]')
>>> BI=scipy.linalg.pinv2(B)
>>> BI*c
Matrix([[ 0.9631014 ],
        [ 0.98854334]])
```

Jak widać, znajomość macierzy pseudoodwrotnej pozwala nam na rozwiązanie zarówno układów niedookreślonych jak i nadokreślonych. W przypadku tych ostatnich możemy skorzystać z metody najmniejszych kwadratów bezpośrednio,

```
>>> x,res,rank,s = scipy.linalg.lstsq(B,c)
>>> print x
[[ 0.9631014 ]
 [ 0.98854334]]
```

Moduł SciPy daje nam również możliwość policzenia norm macierzy i wektorów. Służy do tego polecenie `linalg.norm`,

```
>>> scipy.linalg.norm(B*x-c)
0.10635929472686263
>>> scipy.linalg.norm(B*x-c,2)
0.10635929472686263
>>> scipy.linalg.norm(B*x-c,1)
0.22416978150060307
>>> scipy.linalg.norm(B*x-c,scipy.inf)
0.074723260500200728
```

GNU Octave

Do rozwiązania układu równań liniowych w GNU Octave służy operator mnożenia lewostronnego \. Ponieważ operator ten jest przeciążony, możemy stosować go do wszystkich typów równań:

```
octave:1> A2=[1 2]
A =

  1  2

octave:2> b2=[3]
b = 3
octave:3> A2\b2
```

```
ans =

  0.60000
  1.20000

octave:4> A=[0 2 2;3 3 0;1 0 1]
A =

  0  2  2
  3  3  0
  1  0  1

octave:5> b=[1;3;2]
b =

  1
  3
  2

octave:6> A\b
ans =

   1.25000
  -0.25000
   0.75000

octave:7> A3=[1 1;2.05 -1;3.06 1;-1.02 2;4.08 -1]
A =

   1.0000   1.0000
   2.0500  -1.0000
   3.0600   1.0000
  -1.0200   2.0000
   4.0800  -1.0000

octave:8> b3=[1.98;0.95;3.98;0.92;2.90]
b =

  1.98000
  0.95000
  3.98000
  0.92000
  2.90000

octave:9> A3\b3
ans =

  0.96310
  0.98854
```

Macierz odwrotną do macierzy **A** wyznaczymy za pomocą `inv`,

```
octave:17> inv(A)
ans =
```

```
  -0.25000   0.16667   0.50000
   0.25000   0.16667  -0.50000
   0.25000  -0.16667   0.50000

octave:18> A*inv(A)
ans =

  1  0  0
  0  1  0
  0  0  1
```

Dysponując macierzą odwrotną, również możemy rozwiązać układ równań liniowych:

```
octave:19> x=inv(A)*b
x =

   1.25000
  -0.25000
   0.75000
```

Jednak nie jest to zalecane rozwiązanie ze względu na wydajność i stabilność numeryczną.

Przykład W tabeli 4.1 porównane są dwie metody rozwiązania układu równań $\mathbf{H}\vec{x} = \vec{b}$, gdzie $\mathbf{H}$ to macierz Hilberta,

$$h_{ij} = \frac{1}{i+j-1},\ i,j = 1,\ldots,10, \tag{4.227}$$

natomiast $\vec{b}$ zadane jest wzorem

$$\vec{b} = \mathbf{H}\vec{x}_0, \quad \vec{x}_0 = (1,1,1,1,1,1,1,1,1,1)^{\mathrm{T}}. \tag{4.228}$$

Pierwsza metoda to mnożenie lewostronne,

```
octave:58> H=hilb(10);
octave:59> b=H*ones(10,1)
b =

  2.928968253968254
  2.019877344877345
  1.603210678210678
  1.346800421800422
  1.168228993228993
  1.034895659895660
  0.930728993228993
  0.846695379783615
  0.777250935339171
  0.718771403175428

octave:60> x1=H\b
x1 =

  0.999999999696065
  1.000000025148027
```

	H\b	inv(H)*b
$\|\mathbf{H}\vec{x} - \vec{b}\|$	0	$2,4461 * 10^{-5}$
szybkość	1	$2,39$

Tabela 4.1: Porównanie dwóch metod rozwiązywania układów równań w GNU Octave.

```
  0.999999481719599
  1.000004590192485
  0.999978568454343
  1.000057870920076
  0.999906487963904
  1.000089185046396
  0.999953715568736
  1.000010075438048
```

Natomiast w drugiej korzystamy ze wzoru $\vec{x} = \mathbf{H}^{-1}\vec{b}$,

```
octave:61> x2=inv(H)*b
x2 =

  1.000000000918760
  1.000000027248461
  0.999999719671905
  1.000005646608770
  0.999990645796061
  1.000089749693871
  0.999837309122086
  1.000069409608841
  0.999898672103882
  1.000009082257748
```

Otrzymane wyniki rzeczywiście przemawiają na korzyść metody z mnożeniem lewostronnym, która nie tylko jest szybsza[6], ale daje również dokładniejsze rozwiązanie.

Jeżeli macierz **A** jest macierzą prostokątną, poleceniem `pinv` wyznaczymy macierz pseudoodwrotną do niej,

```
octave:20> A2I=pinv(A2)
ans =

  0.20000
  0.40000

octave:21> A3I=pinv(A3)
ans =

 Columns 1 through 4:

    5.0222e-02    5.0968e-02    1.1857e-01    2.4883e-04
```

[6]Pomiar wydajności obu metod wykonany został przy pomocy polecenia `speed` (patrz par. 3.5).

```
    1.5080e-01    -9.8815e-02     1.8591e-01     2.5013e-01

 Column 5:

    1.1832e-01
   -6.4214e-02
```

Za jej pomocą również możemy rozwiązać niedookreślony lub nadokreślony układ równań,

```
octave:25> A2I*b2
ans =

  0.60000
  1.20000

octave:26> A3I*b3
ans =

  0.96310
  0.98854
```

Funkcja `pinv` wykorzystuje w obliczeniach rozkład macierzy na wartości osobliwe. Gdybyśmy byli zainteresowani samym rozkładem, mamy do dyspozycji polecenie `svd`,

```
octave:29> [U,Sig,V]=svd(A3)
U =

  -0.143595   0.424739  -0.715664  -0.233907  -0.481757
  -0.380824  -0.241398   0.099777   0.675095  -0.575318
  -0.497792   0.548570   0.599072  -0.277974  -0.122955
   0.232071   0.667941  -0.061520   0.636334   0.302146
  -0.729863  -0.119371  -0.339409   0.085692   0.574899

Sig =

  5.73851  0.00000
  0.00000  2.70600
  0.00000  0.00000
  0.00000  0.00000
  0.00000  0.00000

V =

  -0.98668   0.16266
   0.16266   0.98668
```

Algorytm SVD wykorzystywany jest również do wyliczenia wskaźnika uwarunkowania w normie $\|\cdot\|_2$,

```
octave:31> cond(A3)
ans = 2.1207
```

A samą normę macierzy lub wektora znajdziemy, korzystając z polecenia `norm`,

```
octave:32> norm(A3)
```

```
ans = 5.7385
octave:33> norm(A3,2)
ans = 5.7385
octave:34> norm(A3,1)
ans = 11.210
octave:35> norm(A3,Inf)
ans = 5.0800
octave:36> norm(A3,"fro")
ans = 6.3445
```

Do wyznaczenia rozkładów Cholesky'ego, QR i LU zadanej macierzy służą odpowiednio `chol`, `qr` i `lu`,

```
octave:37> [Q,R,P] = qr(A)
Q =

  -0.55470   0.71319  -0.42857
  -0.83205  -0.47546   0.28571
  -0.00000   0.51508   0.85714

R =

  -3.60555  -1.10940  -2.49615
   0.00000   1.94145  -0.91129
   0.00000   0.00000   1.71429

P =

  0  0  1
  1  0  0
  0  1  0

octave:38> [L,U,P] = lu(A)
L =

   1.00000   0.00000   0.00000
   0.00000   1.00000   0.00000
   0.33333  -0.50000   1.00000

U =

  3  3  0
  0  2  2
  0  0  2

P =

  0  1  0
  1  0  0
  0  0  1
```

GNU Octave oferuje również funkcję `det` do wyliczenia wyznacznika macierzy kwadratowej:

```
octave:39> det(A)
ans = -12.000
```

Rozdział 5

Równania nieliniowe

Częstym zadaniem w obliczeniach numerycznych jest rozwiązanie równania nieliniowego (bądź układu takich równań), tzn. znalezienie takiej wartości x, dla której spełniony jest warunek

$$f(x) = 0, \tag{5.1}$$

gdzie $f(x)$ jest pewną funkcją x. Przy tym, $f(x)$ niekoniecznie musi być zadana wzorem matematycznym. Równie dobrze może to być skomplikowany algorytm numeryczny. Ważne jest tylko, aby można było obliczyć wartość f dla zadanego x.

Znalezienie rozwiązania równania nieliniowego jest zazwyczaj dużo trudniejsze niż układu równań liniowych. Dla wielu równań rozwiązanie analityczne albo nie istnieje (równania przestępne, równania algebraiczne rzędu wyższego niż 4), albo jest tak skomplikowane, że zupełnie nie nadaje się do użycia w praktycznych obliczeniach. Na szczęście przybliżone rozwiązania, których możemy poszukać numerycznie, w większości wypadków zupełnie wystarczają.

Praktycznie wszystkie metody poszukiwania pierwiastków równań nieliniowych opierają się na iteracyjnym poprawianiu początkowego przybliżenia szukanego pierwiastka. W niniejszym rozdziale poznamy wybrane z nich.

5.1 Równania z jedną niewiadomą

5.1.1 Twierdzenie o punkcie stałym

Wiele algorytmów rozwiązywania równań nieliniowych należy do dość ogólnej klasy metod iteracyjnych, których działanie gwarantowane jest twierdzeniem o punkcie stałym.

Twierdzenie 5.1.1 ***(o punkcie stałym)*** *Niech $g(x)$ i jej pochodna $g'(x)$ będą funkcjami ciągłymi na pewnym przedziale $I = [\tilde{x} - r, \tilde{x} + r]$ wokół punktu $\tilde{x}$ takiego, że*

$$g(\tilde{x}) = \tilde{x}. \tag{5.2}$$

Wówczas, jeżeli

$$|g'(x)| \leqslant \alpha < 1, \tag{5.3}$$

gdzie α to pewna liczba dodatnia, to iteracja

$$x_{k+1} = g(x_k) \tag{5.4}$$

startująca z dowolnego $x_0 \in I$ dąży do punktu stałego $\tilde{x}$ przekształcenia g.

Dowód w [35].

O ile tylko równanie (5.1) da się przekształcić do postaci

$$x = g(x) \tag{5.5}$$

tak, aby funkcja $g(x)$ spełniała warunki twierdzenia, można spróbować potraktować je jako praktyczny przepis na znalezienie przybliżonego rozwiązania równania. Trudność polega na tym, że istnieje zwykle kilka różnych możliwości przekształcenia go do postaci (5.5). W myśl twierdzenia ze wszystkich możliwości powinniśmy wybrać te, dla których

$$|g'(x)| < 1 \tag{5.6}$$

wewnątrz przedziału I. Jednak bez znajomości chociaż zgrubnego oszacowania rozwiązania określenie przedziału I, a tym samym sprawdzenie warunku (5.6) może okazać się niemożliwe.

Przykład Aby mimo to zademonstrować działanie metody opartej na twierdzeniu o punkcie stałym, rozważmy równanie [35]

$$f(x) = x^2 - 2 = 0. \tag{5.7}$$

„Zgadujemy", że rozwiązanie powinno leżeć w przedziale $I = (1; 1, 5)$ i przekształcamy równanie (5.7) do postaci

$$x = \frac{2}{x}. \tag{5.8}$$

Innymi słowy, $g(x) = 2/x$. Po wyliczeniu pochodnej funkcji $g(x)$ okaże się, że warunek

$$|g'(x)| = \frac{2}{x^2} < 1 \tag{5.9}$$

nie jest spełniony dla wszystkich $x \in I$. W tej sytuacji procedura iteracyjna

$$x_{k+1} = \frac{2}{x_k} \tag{5.10}$$

raczej nie zadziała. I rzeczywiście, już po kilku iteracjach widać, że otrzymaliśmy naprzemienny ciąg wartości

$$x_0 = 1, \quad x_1 = 2, \quad x_2 = 1, \quad x_3 = 2, \ldots, \tag{5.11}$$

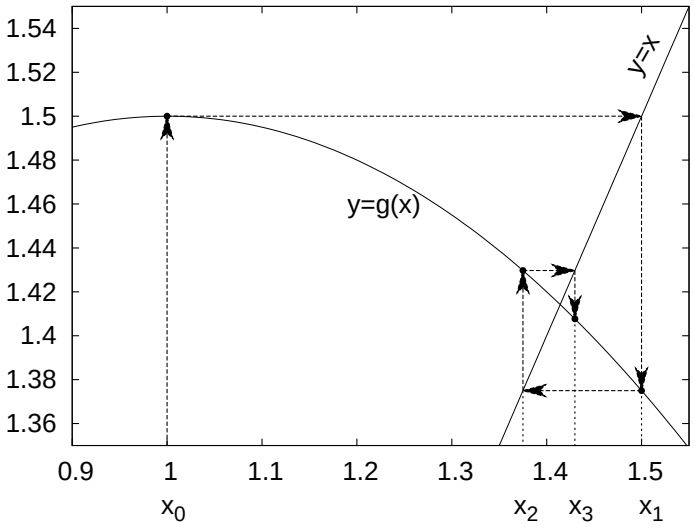
który nigdy nie osiągnie poszukiwanego rozwiązania.

Równanie (5.7) możemy jednak również zapisać w postaci

$$x = -\frac{1}{2}\left\{(x-1)^2 - 3\right\}. \tag{5.12}$$

W tym wypadku funkcja $g(x)$ spełnia warunek zbieżności

$$|g'(x)| = |x - 1| \leqslant 0.5 < 1, \quad \forall x \epsilon I. \tag{5.13}$$



Rysunek 5.1: Poszukiwanie punktu stałego równania $x^2 - 2 = 0$ w oparciu o twierdzenie o punkcie stałym i szereg iteracyjny (5.14).

Można więc użyć iteracji

$$x_{k+1} = -\frac{1}{2}\left\{(x_k - 1)^2 - 3\right\} \tag{5.14}$$

do znalezienia rozwiązania równania (5.7). Szereg iteracyjny

$$x_0 = 1; \quad x_1 = 1,5; \quad x_2 = 1,375; \quad x_3 = 1,4297; \quad x_4 = 1,4077\ldots, \tag{5.15}$$

rzeczywiście dąży do rozwiązania $\sqrt{2} = 1,414\ldots$. Przebieg samej iteracji przedstawiony jest na rys. 5.1.

Aby ostatecznie przekonać się o tym, że wybór $g(x)$ stanowi klucz do sukcesu, przekształcimy równanie (5.7) do postaci

$$x = \frac{1}{2}\left(x + \frac{2}{x}\right). \tag{5.16}$$

Nie dość, że pochodna funkcji $g(x)$ spełnia warunek zbieżności,

$$|g'(x)| = \frac{1}{2}|1 - \frac{2}{x^2}| \leqslant \frac{1}{2} < 1, \quad \forall x \epsilon I, \tag{5.17}$$

to dodatkowo przyjmuje wartość zero dla $x^2 = 2$ stanowiącego rozwiązanie równania. Ta własność powoduje, że szereg iteracyjny zbiega szczególnie szybko do punktu stałego. Rzeczywiście, już po trzech krokach otrzymamy bardzo dobre przybliżenie rozwiązania:

$$x_0 = 1; \quad x_1 = 1,5; \quad x_2 = 1,4167; \quad x_3 = 1,4142; \quad x_4 = 1,4142;\ldots, \tag{5.18}$$

Omawianą tu metodę bardzo łatwo zaimplementować. Poniższy kod stanowi przykład takiej implementacji w GNU Octave (za [35]),

```
function [x,err,xx] = fixedpoint(g,x0,tol,maxit)
% fixedpoint.m
% Szereg iteracyjny rozwiazujacy rownanie x=g(x)
% na podstawie twierdzenia o punkcie stalym
% Wejscie:
%    g     - funkcja
%    x0    - poczatkowe przyblizenie rozwiazania
%    tol   - dokladnosc (|x(n + 1) - x(n)|<tol zatrzymuje
%              procedure)
%    maxit - maksymalna liczba iteracji
% Wyjscie:
%    x   - przyblizone rozwiazanie (ostatni punkt
%           wyliczony przez procedure)
%    err - oszacowanie bledu
%    xx  - wektor przechowujacy wartosci x
%            z kazdego kroku
  if nargin < 4, maxit = 100; endif
  if nargin < 3, tol = 1e-6; endif
  xx(1) = x0;
  for k = 2:maxit
    xx(k) = feval(g,xx(k - 1));
    err = abs(xx(k) - xx(k - 1));
    if err < tol, break; endif
  endfor
  x = xx(k);
  if k == maxit
    fprintf("Slaba zbieznosc lub jej brak!\n")
  endif
endfunction
```

Działanie programu ilustruje następujący przykład:

```
octave:1> h=inline('0.5*(x+2.0/x)','x');
octave:2> [x,err,xx] = fixedpoint(h,1)
x = 1.4142
err =  1.5949e-12
xx =

  1.0000  1.5000  1.4167  1.4142  1.4142  1.4142
```

5.1.2 Lokalizacja miejsc zerowych

Dużą rolę w rozwiązywaniu równań nieliniowych dowolną metodą odgrywa wybór wartości startowej (lub przedziału, w którym ona się znajduje), czyli zgrubnego oszacowania wartości poszukiwanego pierwiastka. Źle wybrana wartość może spowodować, że metoda iteracyjna w ogóle nie będzie zbieżna lub, co również możliwe, że znajdzie „złe” rozwiązanie, czyli pierwiastek inny od szukanego.

Najprostszym i chyba najlepszym sposobem znalezienia wartości startowej jest sporządzenie wykresu funkcji. Nawet niezbyt dokładny wykres wykonany na podstawie kilku charakterystycznych punktów pozwala najczęściej wybrać rozsądne przybliżenie początkowe poszukiwanego pierwiastka.

Jeżeli metoda wymaga od nas przedziału, w którym znajduje się rozwiązanie, a nie tylko wartości początkowej, to powinniśmy wybrać tzw. przedział izolacji pierwiastka, tzn. przedział, w którym jest on jedynym rozwiązaniem rozpatrywanego równania. Wiele metod zawodzi mianowicie, kiedy podaje im się na starcie przedział zawierający więcej pierwiastków.

Wykres funkcji jako metoda lokalizacji rozwiązań sprawdza się znakomicie przy rozwiązaniu jednego (lub kilku równań), o ile tylko stanowi to cel sam w sobie. Niestety, gdy mamy wiele różnych równań do rozwiązania, sporządzenie wykresu dla każdego z nich i odczytanie z nich odpowiednich przedziałów będzie bardzo czasochłonne. Podobnie rzecz będzie się miała w sytuacji, kiedy rozwiązanie równania nieliniowego stanowi tylko krok pośredni skomplikowanych (na ogół) obliczeń, w większości przeprowadzanych automatycznie na komputerze[1]. Trudno wówczas wymagać, aby operator przerywał działanie programu w odpowiednim momencie, rysował wykres interesującej go funkcji, odczytał przybliżone położenie jej pierwiastków, wprowadzał odpowiednie dane do programu i ponownie go uruchamiał. W takiej sytuacji możemy skorzystać z metody automatycznego oddzielania pierwiastków (ang. *incremental search*) [56]. Opiera się ona na prostej obserwacji. Jeżeli mianowicie $f(a)f(b) < 0$, to ciągła funkcja $f(x)$ musi mieć w przedziale (a, b) przynajmniej jeden pierwiastek. Jeżeli dodatkowo przedział ten będzie mały, istnieje duże prawdopodobieństwo, że będzie on przedziałem izolacji danego pierwiastka. Stąd wynika, że do lokalizacji pierwiastków wystarczy zbadać zmiany znaku w ciągu wartości funkcji wyliczonych dla dyskretnego zbioru punktów

$$x_i = x_0 + i\Delta x, \quad n = 0, 1, 2, 3, \ldots \tag{5.19}$$

odległych od siebie o pewien niewielki krok Δ.

Przykład Chcemy zlokalizować pierwiastek równania

$$x^2 - 2 = 0. \tag{5.20}$$

Wartości funkcji $f(x) = x^2 - 2$ obliczone dla różnych x z krokiem $\Delta = 0.2$ zestawione są w tabeli 5.1. Ze zmiany znaku wartości funkcji wnioskujemy, że pierwiastek leży w przedziale $(1,4; 1,6)$.

Niewątpliwą zaletą powyższego algorytmu jest jego prostota. Bardzo łatwo go zaimplementować, o czym świadczy poniższy przykład:

```
function rts = rootsearch(f,a,b,dx)
% rootsearch.m
% Lokalizacja miejsc zerowych metoda izolacji
% pierwiastkow (ang. incremental search)
% Wejscie:
%    f     - funkcja zdefiniowana przez uzytkownika
%    a,b   - przedzial, w ktorym szukamy pierwiastkow
%    dx    - krok, z jakim "skanujemy" przedzial (a,b)
% Wyjscie:
%    rts   - tzw. cell array z przedzialami izolacji
%            pierwiastkow, zawiera element [inf inf],
```

[1] Istnieją projekty, w których obliczenia potrafią trwać nawet miesiące na komputerach o mocach przekraczających znacznie możliwości zwykłego peceta.

x	$f(x)$
0.00000	-2.000000
0.20000	-1.960000
0.40000	-1.840000
0.60000	-1.640000
0.80000	-1.360000
1.00000	-1.000000
1.20000	-0.560000
1.40000	-0.040000
1.60000	0.560000
1.80000	1.240000
2.00000	2.000000

Tabela 5.1: Metoda lokalizacji pierwiastków w praktyce.

```
%            jesli w podanym przedziale (a,b) nie ma
%            zadnego pierwiastka
  k=0;
  rts{1} = [inf inf];
  x=a:dx:b;
  y=feval(f,x);
  for m=1:length(x)-1
    if y(m)*y(m+1)<0.0
      k=k+1;
      rts{k}=[x(m) x(m+1)];
    endif
  endfor
endfunction
```

Sprawdźmy, jak funkcja `rootsearch` działa w praktyce:

```
octave:1> function y=f(x)
> y=x.**2-2;
> endfunction
octave:2> rootsearch('f',0,2,0.2)
ans =

{
  [1,1] =

    1.4000  1.6000

}
octave:3> function y=f(x)
> y=x.**3-10*x.**2 +5;
> endfunction
octave:4> rootsearch('f',-5,15,0.02)
ans =

{
  [1,1] =
```

```
    -0.70000   -0.68000

  [1,2] =

    0.72000   0.74000

  [1,3] =

    9.9400   9.9600

}
octave:5> rootsearch('f',-10,-5,0.02)
ans =

{
  [1,1] =

              Inf            Inf

}
```

W większości sytuacji metoda zadziała poprawnie. Należy jednak zdawać sobie sprawę z kilku potencjalnych problemów [56]:

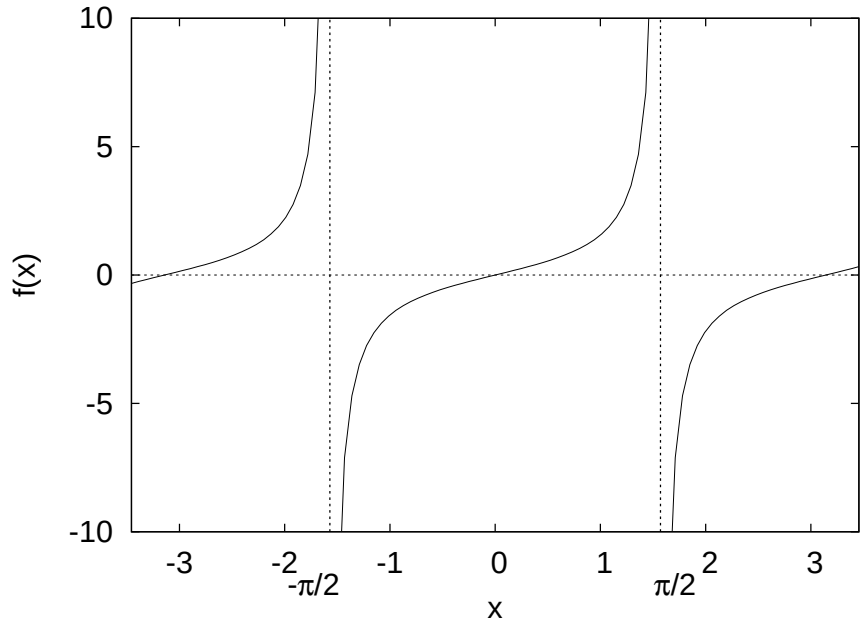
- jeśli krok Δx jest większy, niż odległość między dwoma sąsiednimi pierwiastkami, możemy je przeoczyć;
- pierwiastek o parzystej krotności nie zostanie znaleziony;
- niektóre osobliwości mogą zostać potraktowane jako pierwiastki.

Przykład Rozważmy funkcję

$$f(x) = \tan x. \tag{5.21}$$

Korzystając z algorytmu automatycznej lokalizacji pierwiastków otrzymamy

```
octave:84> rootsearch('tan',-1.1*pi,1.1*pi,0.02)
ans =

{
  [1,1] =

    -3.1558   -3.1358

  [1,2] =

    -1.5758   -1.5558

  [1,3] =

    -0.0157519    0.0042481

  [1,4] =

    1.5642   1.5842
```



Rysunek 5.2: Przykład funkcji, dla której metoda automatycznego wyszukiwania pierwiastków nie zadziała poprawnie.

```
  [1,5] =

    3.1242   3.1442

}
```

Z powyższych przedziałów izolacji tylko pierwszy, trzeci i piąty zawierają rzeczywiste pierwiastki. W dwóch pozostałych funkcja wprawdzie zmienia znak, ale ma tam osobliwość i nie przecina osi X, co widać na rysunku 5.2.

5.1.3 Metoda bisekcji

Jeżeli w metodzie lokalizacji pierwiastków z poprzedniego paragrafu używaliśmy małego kroku, może się okazać, że uzyskane w ten sposób przybliżenie pierwiastków jest wystarczające. Najczęściej będziemy jednak chcieli to przybliżenie poprawić, najlepiej w jakiś systematyczny sposób. Do dyspozycji mamy kilka metod, z których najprostszą jest chyba metoda bisekcji, czyli połowienia przedziału izolacji [15, 35, 50].

Sama nazwa wskazuje już, jak będzie działał algorytm. Dla miejsca zerowego α ciągłej funkcji $f(x)$ leżącego w przedziale (a_1, b_1) metoda wygeneruje ciąg przedziałów

$$(a_1, b_1) \supset (a_2, b_2) \supset (a_3, b_3) \supset \ldots, \tag{5.22}$$

z których każdy zawiera α i dodatkowo, każdy jest o połowę krótszy od poprzedniego. Mając do dyspozycji przedział $I_{k-1} = (a_{k-1}, b_{k-1})$ kolejny wyznaczamy

według następującego przepisu:

(a) Znajdujemy środek m_k przedziału I_{k-1},

$$m_k = \frac{1}{2}\left(a_{k-1} + b_{k-1}\right). \tag{5.23}$$

(b) Jeżeli $f(m_k) = 0$, znaleźliśmy pierwiastek. W przeciwnym razie

$$(a_k, b_k) = \begin{cases} (m_k, b_{k-1}), & \text{jeśli } f(m_k)f(b_{k-1}) < 0, \\ (a_{k-1}, m_k), & \text{jeśli } f(a_{k-1})f(m_k) < 0. \end{cases} \tag{5.24}$$

Zwróćmy uwagę, że do wyboru nowego przedziału korzystamy z tej samej własności, która w poprzednim paragrafie posłużyła nam do lokalizacji pierwiastka: jeżeli w przedziale (a, b) znajduje się miejsce zerowe ciągłej funkcji $f(x)$, to $f(a)f(b) < 0$.

Pewnego komentarza wymaga wzór (5.23) na wyliczanie punktu środkowego przedziału. Otóż w obliczeniach prowadzonych ze skończoną dokładnością w arytmetyce dziesiętnej nierówności

$$a_{k-1} \leqslant \frac{1}{2}\left(a_{k-1} + b_{k-1}\right) \leqslant b_{k-1} \tag{5.25}$$

mogą nie być spełnione dla wszystkich liczb zmiennoprzecinkowych a_{k-1} i b_{k-1} [57]. Dla ustalenia uwagi załóżmy, że używamy arytmetyki z sześcioma liczbami dziesiętnymi. Niech $a_{k-1} = 0,742531$ i $b_{k-1} = 0,742533$. Stąd wynika $a_{k-1} + b_{k-1} = 1.48506$ (po zaokrągleniu) i $\frac{1}{2}(a_{k-1} + b_{k-1}) = 0,742530$. Innymi słowy $\frac{1}{2}(a_{k-1} + b_{k-1}) < a_{k-1}$! Dlatego bezpieczniej jest używać wzoru

$$m_k = a_{k-1} + \frac{1}{2}\left(b_{k-1} - a_{k-1}\right), \tag{5.26}$$

który gwarantuje spełnienie nierówności $a_{k-1} \leqslant m_k \leqslant b_{k-1}$ w arytmetyce o dowolnej podstawie.

Proces bisekcji przedstawiony jest na rys. 5.3. O ile wcześniej nie natrafimy na pierwiastek, po n krokach otrzymamy przedział o długości

$$\frac{1}{2^n}(b - a), \tag{5.27}$$

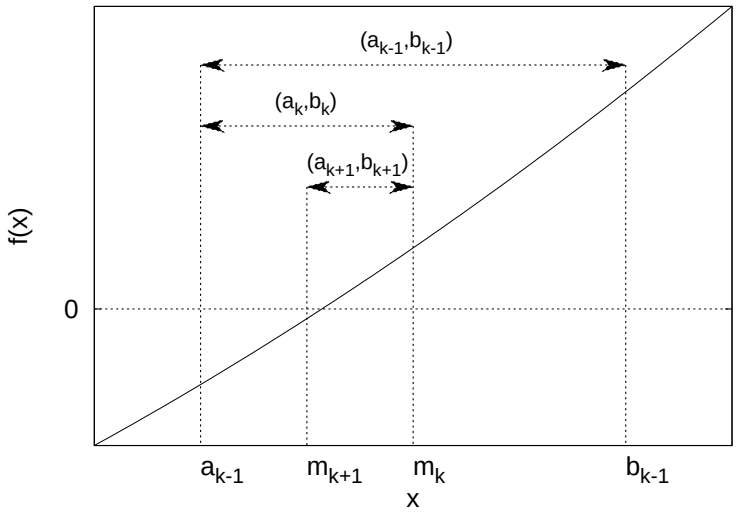
w którym zawarty jest pierwiastek równania. Jako wartość przybliżoną pierwiastka przyjmujemy

$$\alpha = m_{n+1} \pm d_n, \quad d_n = \frac{1}{2^{n+1}}(b - a). \tag{5.28}$$

Zaletą metody jest jej prostota oraz pewność, że jeżeli będziemy kontynuować podział przedziałów odpowiednio długo, otrzymamy wynik z żądaną dokładnością. Jednak zbieżność bisekcji jest wolna. W każdym kroku iteracji zyskujemy jedną dokładną cyfrę dwójkową. Ponieważ

$$10^{-1} \simeq 2^{-3,3},$$

więc jedną cyfrę dziesiętną uzyskamy średnio co 3,3 kroków.



Rysunek 5.3: Poszukiwanie pierwiastków równania $f(x) = 0$ metodą bisekcji.

Przykład W tabeli 5.2 zestawione są wyniki zastosowania bisekcji do równania

$$x^2 - 2 = 0 \tag{5.29}$$

i przedziału $I_0 = (1; 1, 5)$.

Poniższy kod stanowi przykład implementacji algorytmu bisekcji w GNU Octave (za [35]):

```
function [x,err,xx] = bisec(f,a,b,tol,maxit)
% bisec.m
% Rozwiazanie rownania f(x)=0 metoda bisekcji
% Wejscie:
%    f     - funkcja
%    a,b   - pierwszy przedzial izolacji
%    tol   - dokladnosc
%    maxit - maksymalna liczba iteracji
% Wyjscie:
%    x   - przyblizone rozwiazanie (ostatni punkt
%          wyliczony przez procedure)
%    err - polowa szerokosci ostatniego przedzialu
%    xx  - wektor przechowujacy wszystkie
%          przyblizenia rozwiazania
  tolf=eps; fa = feval(f,a); fb = feval(f,b);
  if fa*fb > 0
    error("Bledny przedzial izolacji pierwiastka!");
  endif
  for k = 1: maxit
    xx(k) = a+(b-a)/2;
```

n	a_{n-1}	b_{n-1}	m_n	$f(m_n)$
1	1,0000	1,5000	1,2500	-0,43750000
2	1,2500	1,5000	1,3750	-0,10938000
3	1,3750	1,5000	1,4375	0,06640600
4	1,3750	1,4375	1,4062	-0,02260200
5	1,4062	1,4375	1,4219	0,02180000
6	1,4062	1,4219	1,4141	-0,00032119
7	1,4141	1,4219	1,4180	0,01072400
8	1,4141	1,4180	1,4160	0,00505600
9	1,4141	1,4160	1,4150	0,00222500
10	1,4141	1,4150	1,4146	0,00109320
11	1,1441	1,4146	1,4143	0,00024449
12	1,4141	1,4143	1,4142	0,00003836

Tabela 5.2: Metoda bisekcji zastosowana do równania $x^2 - 2 = 0$.

```
    fx = feval(f,xx(k)); err = (b-a)/2;
    if abs(fx) < tolf | abs(err)<tol
      break;
    elseif
      fx*fa > 0, a = xx(k); fa = fx;
    else
      b = xx(k);
    endif
  endfor
  x = xx(k);
  if k == maxit
    fprintf("Najlepsze x po  %d iteracjach\n",maxit);
  endif
endfunction
```

Jego wykorzystanie nie powinno sprawiać kłopotów:

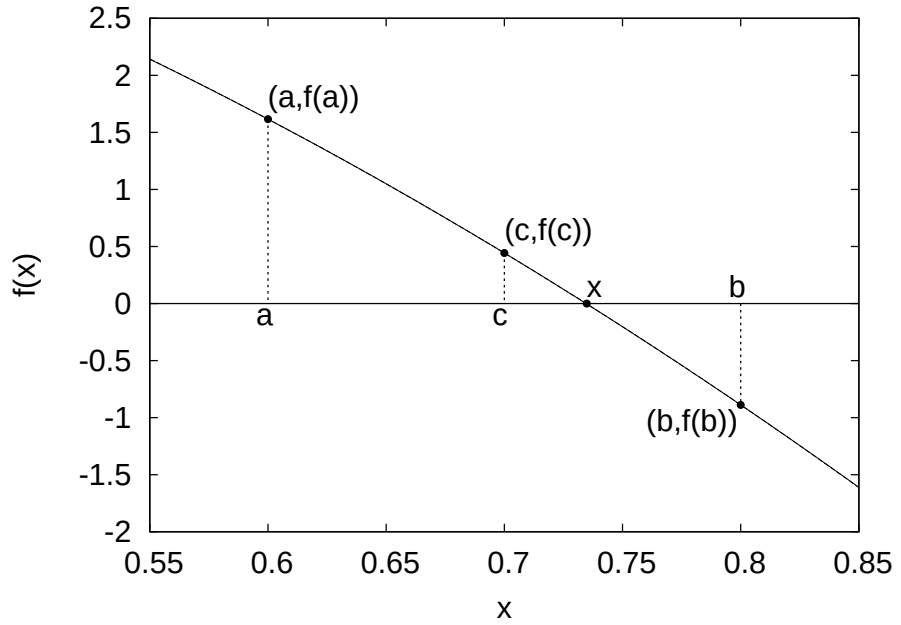
```
octave:1> function y=f(x)
> y=x.**2-2;
> endfunction
octave:2> bisec('f',1,1.5,1e-6,100)
ans = 1.4142
```

Naturalnym uogólnieniem bisekcji jest metoda wielopodziału przedziału izolacji (a, b) pierwiastka α. Jak sama nazwa wskazuje, w jednym kroku dzielimy ten przedział na k podprzedziałów $I_i = [x_i, x_{i+1}]$, gdzie

$$x_i = a + i\left(\frac{b-a}{k}\right), \quad i = 0, 1, 2, \ldots, k. \tag{5.30}$$

Wybór nowego przedziału izolacji pierwiastka odbywa się dokładnie tak samo, jak poprzednio. Szukając w ten sposób pierwiastka z dokładnością ϵ musimy wykonać

$$n_k = \frac{\log_2\left(\frac{b-a}{2\epsilon}\right)}{\log_2 k} \tag{5.31}$$



Rysunek 5.4: Schemat metody Brenta. Prowadzimy parabolę przez $(a, f(a))$, $(c, f(c))$ i $(b, f(b))$ i szukamy punktu przecięcia z osią X.

podziałów. Z metody wielopodziału warto skorzystać w sytuacji, kiedy w przedziale początkowym jest kilka pierwiastków i mamy możliwość równoległego operowania na większej liczbie podprzedziałów.

5.1.4 Metoda Brenta

Metoda Brenta [15, 56, 58] łączy w sobie niezawodność bisekcji (par. 5.1.3) z odwrotną interpolacją kwadratową (patrz rozdział 6). Jej schemat przedstawiony jest na rys. 5.4. Dzielimy wyjściowy przedział (a, b) izolacji pierwiastka na połowę. Podobnie, jak w metodzie bisekcji określamy, w którym z przedziałów $\left(a, c = \frac{a+b}{2}\right)$ i (c, b) leży poszukiwany pierwiastek. Nie kontynuujemy jednak połowienia, tylko przez punkty $(a, f(a))$, $(c, f(c))$ i $(b, f(b))$ prowadzimy parabolę i szukamy punktu przecięcia z osią X. Znalezienie wzoru paraboli przechodzącej przez trzy dane punkty nie powinno sprawić trudności[2]

$$\begin{aligned} x &= \frac{[y-f(b)][y-f(c)]}{[f(a)-f(b)][f(a)-f(c)]}a + \frac{[y-f(a)][y-f(c)]}{[f(b)-f(a)][f(b)-f(c)}b \\ &\quad + \frac{[y-f(a)][y-f(b)]}{[f(c)-f(a)][f(c)-f(b)]}c. \end{aligned} \tag{5.32}$$

[2]W przypadku metody Brenta mówi się o odwrotnej interpolacji kwadratowej, ponieważ odwrócone jest znaczenie x i y, tzn. szukamy $x = f(y)$.

Kładąc $y = 0$ znajdziemy nowe przybliżenie poszukiwanego pierwiastka

$$x = -\frac{af(b)f(c)[f(b) - f(c)] + bf(c)f(a)[f(c) - f(a)] + cf(a)f(b)[f(a) - f(b)]}{[f(a) - f(b)][f(b) - f(c)][f(c) - f(a)]}. \tag{5.33}$$

Aby ustrzec się przed niespodziankami, przybliżenie to przyjmujemy tylko pod warunkiem, że leży ono w nowym przedziale izolacji. W przeciwnym razie wynik interpolacji porzucamy i przeprowadzamy następny krok bisekcji. Procedurę tę powtarzamy aż do uzyskania żądanej dokładności.

Przykład Skorzystamy teraz z metody Brenta, aby znaleźć pierwiastek funkcji $f(x) = x^3 - 10x^2 + 5$ znajdujący się początkowo w przedziale $(0,6; 0,8)$.

W punktach startowych mamy

$$\begin{aligned} a &= 0,6, \quad f(a) = 1,616, \\ b &= 0,8, \quad f(b) = -0,888. \end{aligned}$$

Połowimy przedział izolacji pierwiastka:

$$c = 0,7, \quad f(c) = 0,443.$$

Stąd wynika, że nowym przedziałem izolacji pierwiastka jest $(c, b) = (0,7; 0,8)$.

Przez punkty $(a, f(a))$, $(c, f(c))$ i $(b, f(b))$ prowadzimy parabolę. Ze wzoru (5.33) znajdujemy punkt przecięcia z osią X,

$$x = 0,73487.$$

Ponieważ leży on w nowym przedziale izolacji pierwiastka, akceptujemy wynik. W ten sposób pierwszy krok metody Brenta został ukończony. W drugim kroku wartości c, x i b będziemy traktować jako nowe wartości a, c i b,

$$\begin{aligned} c = 0,7 &\rightarrow a \\ x = 0,73487 &\rightarrow c \\ b = 0,8 &\rightarrow b \end{aligned}$$

Nowym przedziałem izolacji będzie $(a, c) = (0,7; 0,73487)$. Interpolacja kwadratowa prowadzi do

$$x = 0,73460.$$

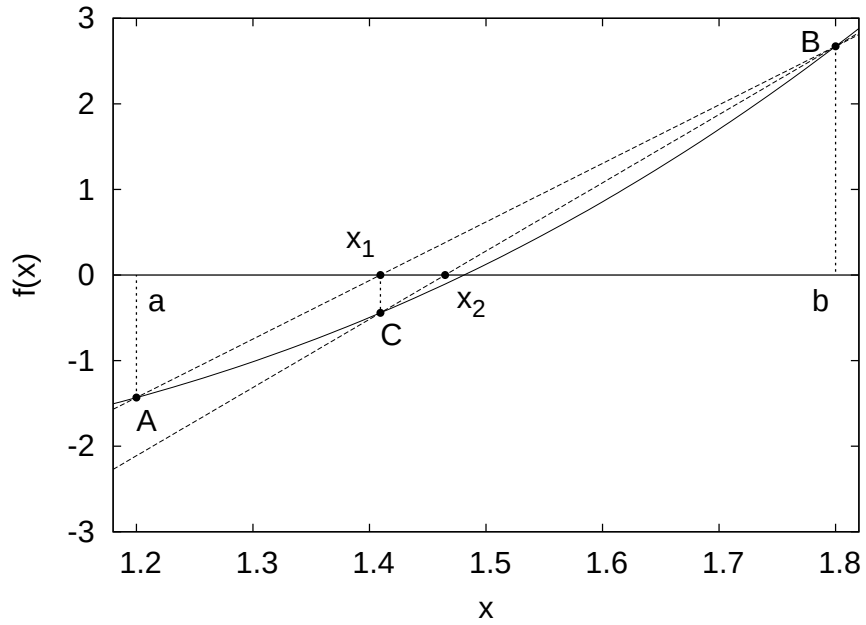
Ponownie akceptujemy wynik, ponieważ leży on w przedziale izolacji. W ten sposób, po dwóch krokach, otrzymaliśmy rozwiązanie z pięcioma poprawnymi cyframi dziesiętnymi.

5.1.5 Regula Falsi

Przyjmijmy ponownie, że równanie $f(x) = 0$ ma w przedziale (a, b) pojedynczy pierwiastek α. Założymy ponadto, że funkcja $f(x)$ jest klasy C^2 na przedziale $\langle a, b\rangle$ oraz że jej pierwsza i druga pochodna mają stały znak na tym przedziale[3] [15, 50, 59]. Dla ustalenia uwagi rozważymy przypadek $f'(x) > 0$ i $f''(x) > 0$ dla $x \in \langle a, b\rangle$. W pozostałych przypadkach będziemy postępować analogicznie.

Przez punkty $A = (a, f(a))$ i $B = (b, f(b))$ prowadzimy cięciwę (rys. 5.5). Jej równanie ma postać

[3]Założenia te pozwolą oszacować błąd metody i określić punkt stały iteracji



Rysunek 5.5: Schemat algorytmu regula falsi.

$$y - f(a) = \frac{f(b) - f(a)}{b - a}(x - a). \tag{5.34}$$

Odciętą x_1 punktu, w którym cięciwa AB przecina oś X, przyjmujemy za pierwsze przybliżenie szukanego pierwiastka:

$$-f(a) = \frac{f(b) - f(a)}{b - a}(x_1 - a) \quad \Rightarrow x_1 = a - \frac{f(a)}{f(b) - f(a)}(b - a). \tag{5.35}$$

Jeżeli $f(x_1) = 0$, to znaleźliśmy nasz pierwiastek. Jeżeli $f(x_1) \neq 0$ i przybliżenie nie jest wystarczające, to przez punkt $C = (x_1, f(x_1))$ oraz jeden z punktów A i B (wybieramy ten, w którym funkcja jest przeciwnego znaku niż w C) prowadzimy następną cięciwę. Odcięta punktu, w którym cięciwa przetnie oś X, będzie kolejnym przybliżeniem pierwiastka. W ten sposób otrzymamy ciąg

$$x_1, x_2, \ldots, x_k, \ldots$$

kolejnych przybliżeń pierwiastka,

$$\begin{aligned} x_0 &= a, \\ x_{k+1} &= x_k - \frac{f(x_k)}{f(b) - f(x_k)}(b - x_k), \quad k = 1, 2, \ldots \end{aligned} \tag{5.36}$$

Można wykazać, że powyższy ciąg jest rosnący i ograniczony z góry, musi być więc zbieżny. Możemy zatem przejść w równaniu rekurencyjnym do granicy $k \to \infty$,

$$g = g - \frac{f(g)}{f(b) - f(g)}(b - g), \tag{5.37}$$

gdzie

$$g = \lim_{k\to\infty} x_k, \quad a < g < b. \tag{5.38}$$

Stąd wynika $f(g) = 0$, czyli ciąg kolejnych przybliżeń x_1, x_2, ..., jest rzeczywiście zbieżny do pierwiastka równania $f(x) = 0$.

Korzystając z twierdzenia Lagrange'a o przyrostach,

$$f(x_n) - f(\alpha) = f'(c)(x_n - \alpha), \quad x_n < c < \alpha, \tag{5.39}$$

możemy oszacować błąd n-tego przybliżenia ($f(\alpha) = 0$)

$$|x_n - \alpha| \leqslant \frac{f(x_n)}{m}, \quad m = \inf_{x\in\langle a,b\rangle} |f'(x)|. \tag{5.40}$$

Błąd ten możemy ocenić również znając dwa kolejne przybliżenia. Ze wzoru (5.36) mamy mianowicie

$$-f(x_k) = \frac{f(x_k) - f(b)}{x_k - b}(x_{k+1} - x_k). \tag{5.41}$$

Ponieważ $f(\alpha) = 0$, więc

$$f(\alpha) - f(x_k) = \frac{f(x_k) - f(b)}{x_k - b}(x_{k+1} - x_k). \tag{5.42}$$

Z twierdzenia Lagrange'a otrzymujemy

$$(\alpha - x_k)f'(\xi_k) = (x_{k+1} - x_k)f'(\bar{x}_k), \tag{5.43}$$

przy czym $\xi_k \in (x_k, \alpha)$, $\bar{x}_k \in (x_k, b)$. Obustronne dodanie do powyższego równania wyrazu $-x_{k+1}f'(\xi_k)$ prowadzi do

$$|\alpha - x_{k+1}| = \frac{|x_k - x_{k+1}| \cdot |f'(\xi_k) - f'(\bar{x}_k)|}{|f'(\xi_k)|} \leqslant \frac{M - m}{m}|x_{k+1} - x_k|, \tag{5.44}$$

gdzie

$$\begin{aligned} m &= \inf_{x\in\langle a,b\rangle} |f'(x)|, \\ M &= \sup_{x\in\langle a,b\rangle} |f'(x)|. \end{aligned} \tag{5.45}$$

Oszacowanie (5.44) jest na ogół bardzo pesymistyczne, a ponadto wymaga znajomości m i M. Dlatego dla przybliżeń w niewielkim otoczeniu pierwiastka α korzystamy z

$$|\alpha - x_{k+1}| \sim \left|\frac{f(x_{k+1})}{f'(x_{k+1})}\right| \sim \left|\frac{x_{k+1} - x_k}{f(x_{k+1}) - f(x_k)}\right| \cdot |f(x_{k+1})|. \tag{5.46}$$

Regula falsi jest metodą zbieżną dla dowolnej funkcji ciągłej na przedziale $\langle a, b\rangle$, o ile tylko spełniony jest warunek $f(a)f(b) < 0$ i pierwsza pochodna tej funkcji jest ograniczona i różna od zera w otoczeniu pierwiastka. Jeżeli druga pochodna nie zmienia znaku w rozpatrywanym przedziale, to ten punkt, w którym

$$ff'' > 0,$$

jest punktem stałym iteracji. Podobnie jak bisekcja, również ta metoda jest stosunkowo wolno zbieżna.

Większość omawianych tu algorytmów rozwiązywania równań nieliniowych jest stosunkowo łatwo zaimplementować samemu. Poniższy kod stanowi przykład takiej implementacji metody regula falsi w GNU Octave:

```
function [x,err,fx]=regula(f,a,b,delta,epsilon,maxit)
% regula.m
% Rozwiazanie rownania f(x)=0 metoda regula falsi
% Wejscie:
%    f       - funkcja
%    a,b     - pierwszy przedzial izolacji
%    delta   - dokladnosc rozwiazania
%    epsilon - dokladnosc f(x)
%    maxit   - maksymalna liczba iteracji
% Wyjscie:
%    x       - pierwiastek
%    err     - oszacowanie bledu
%    fx      - f(x)
fa = feval(f,a); fb = feval(f,b);
if fa*fb>0
  error("Bledny przedzial izolacji pierwiastka!");
endif
for k=1:maxit
  dx=fb*(b-a)/(fb-fa);
  x=b-dx;
  ac=x-a;
  fx=feval(f,x);
  if fx==0
    break
  elseif fb*fx>0
    b=x;
    fb=fx;
  else
    a=x;
    fa=fx;
  endif
  dx=min(abs(dx),ac);
  if abs(dx)<delta
    break
  endif
  if abs(fx)<epsilon
    break
  endif
endfor
err=abs(b-a)/2;
```

Przykład Chcemy znaleźć dodatni pierwiastek równania

$$x^3 + x^2 - 3x - 3 = 0.$$

Z wykresu funkcji $f(x) = x^3 + x^2 - 3x - 3$ wynika, że pierwiastek dodatni leży

x	$f(x)$
$a = 1$	-4
$b = 2$	3
$x_1 = 1,57142$	-1,36449
$x_2 = 1,70540$	-0,24784
$x_3 = 1,72788$	-0,03936
$x_4 = 1,73140$	-0,00615

Tabela 5.3: Regula falsi w przypadku funkcji $f(x) = x^3 + x^2 - 3x - 3$.

w przedziale $(1, 2)$. Ponadto,

$$\begin{aligned} f'(x) &= 3x^2 + 2x - 3, \\ f''(x) &= 6x + 2, \end{aligned}$$

zatem obie pochodne są dodatnie w tym przedziale. W tabeli 5.3 przedstawione są wyniki kolejnych iteracji.

5.1.6 Metoda siecznych

Metodę regula falsi możemy ulepszyć rezygnując z założenia, aby funkcja miała w punktach wytyczających następną cięciwę różne znaki. Kolejne przybliżenie pierwiastka będziemy w tym wypadku będziemy wyznaczać z poprzednich przybliżeń,

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}. \tag{5.47}$$

Otrzymaliśmy w ten sposób metodę siecznych, która na ogół dąży do pierwiastka znacznie szybciej niż regula falsi [15,50]. Zdarzają się jednak przypadki, dla których nie jest ona zbieżna (np. gdy początkowe przybliżenia nie leżą dostatecznie blisko pierwiastka). Ponadto, gdy różnica $(x_k - x_{k-1})$ jest tego samego rzędu, co oszacowanie błędu, następne przybliżenie może już być nie do przyjęcia. Jeżeli w trakcie obliczeń odległości między kolejnymi przybliżeniami zaczynają wzrastać, należy je przerwać i przeprowadzić ponowną lokalizację pierwiastka, zawężając przedział izolacji.

Przykład Rozważmy ponownie równanie

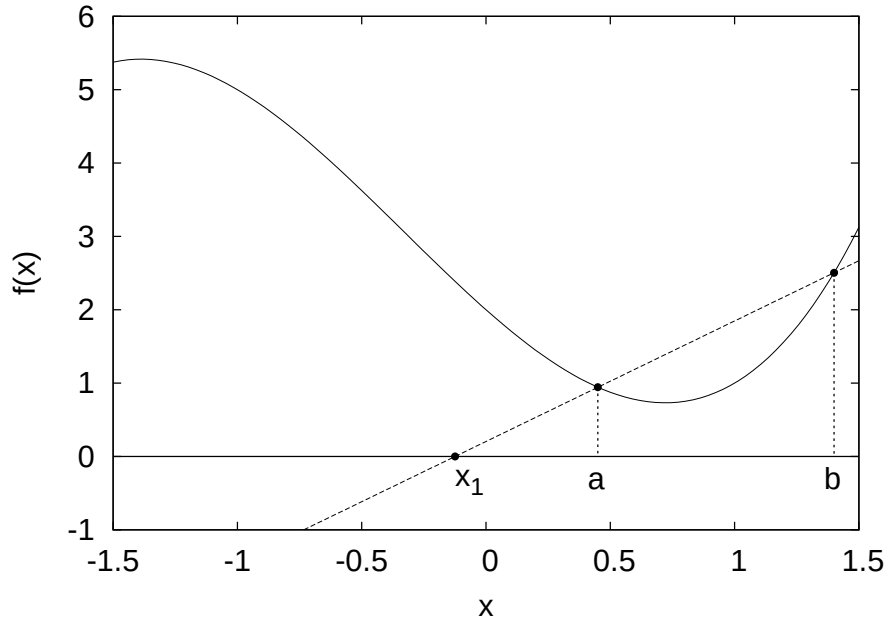
$$x^3 + x^2 - 3x - 3 = 0.$$

W tabeli 5.4. zestawione są wyniki obliczeń pierwiastka dodatniego metodą siecznych. Jak widać, przynajmniej w tym przykładzie metoda siecznych szybciej zbiega do poszukiwanego pierwiastka niż omawiana wcześniej regula falsi.

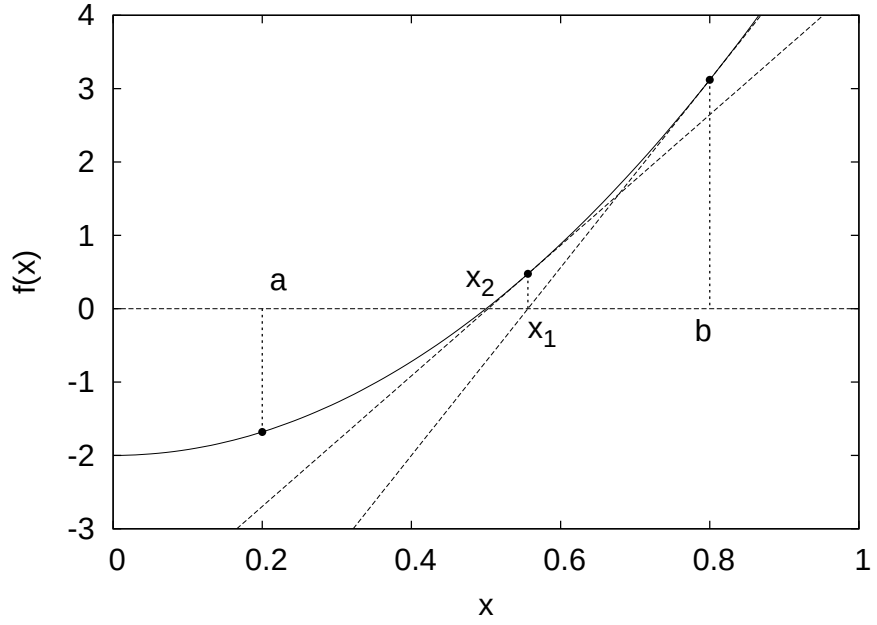
Należy podkreślić, że z wymagania, aby w punktach wyznaczających cięciwę funkcja miała różne znaki, wolno nam zrezygnować jedynie w dalszych krokach iteracji. W pierwszym kroku warunek ten ciągle jest potrzebny, aby uniknąć wykrycia rozwiązań, które w rzeczywistości nie istnieją (rys. 5.6).

x	$f(x)$
$a = 1$	-4
$b = 2$	3
$x_1 = 1,57142$	-1,36449
$x_2 = 1,70540$	-0,24784
$x_3 = 1,73513$	0,02920
$x_4 = 1,73199$	0,000576

Tabela 5.4: Metoda siecznych dla funkcji $f(x) = x^3 + x^2 - 3x - 3$.



Rysunek 5.6: Przykład złego użycia metody siecznych.



Rysunek 5.7: Metoda Newtona.

5.1.7 Metoda Newtona

Jeżeli funkcja $f(x)$ jest ciągła na przedziale $\langle a, b\rangle$, spełnia warunek $f(a)f(b) < 0$ oraz $f'(x)$ i $f''(x)$ mają stały znak w rozpatrywanym przedziale, to równanie

$$f(x) = 0 \tag{5.48}$$

można również rozwiązać metodą Newtona (metodą stycznych) [15,50,59]. Dzięki wykorzystaniu dodatkowej informacji zawartej w pochodnych funkcji metoda generuje na ogół ciąg przybliżeń zbieżny szybciej, niż w przypadku bisekcji czy regula falsi.

Schemat metody przedstawiony jest na rysunku 5.7. Od końca przedziału, w którym funkcja $f(x)$ ma ten sam znak co jej druga pochodna, prowadzimy styczną do wykresu funkcji. Za pierwsze przybliżenie pierwiastka, x_1, bierzemy punkt przecięcia się tej stycznej z osią X. Z punktu $(x_1, f(x_1))$ prowadzimy następną styczną i określamy kolejne przybliżenie. Postępujemy w ten sposób aż do uzyskania żądanej dokładności.

Z równania stycznej otrzymamy

$$f(x_n) + (x_{n+1} - x_n)f'(x_n) = 0. \tag{5.49}$$

Stąd łatwo już otrzymać iteracyjny wzór Newtona na kolejne przybliżenie pierwiastka równania $f(x) = 0$:

$$x_{n+1} = x_n + h_n, \quad h_n = -\frac{f(x_n)}{f'(x_n)}. \tag{5.50}$$

Powróćmy jeszcze na chwilę do przykładu pokazanego na rysunku 5.7. W tym przypadku obie pochodne są dodatnie na rozpatrywanym przedziale. Ze

wzoru (5.50) wynika

$$x_1 = b - \frac{f(b)}{f'(b)}. \tag{5.51}$$

Ponadto, z rozwinięcia w szereg Taylora otrzymujemy

$$f(\alpha) = f(b) + f'(b)(\alpha - b) + \frac{1}{2}f''(c)(\alpha - b)^2, \tag{5.52}$$

gdzie $c \in \langle a, b\rangle$. Jeśli α ma być pierwiastkiem równania, to z powyższego wzoru wynika

$$\alpha = b - \frac{f(b)}{f'(b)} - \frac{1}{2}\frac{f''(c)}{f'(b)}(\alpha - b)^2, \tag{5.53}$$

czyli

$$\alpha - x_1 = -\frac{1}{2}\frac{f''(c)}{f'(b)}(\alpha - b)^2 < 0 \tag{5.54}$$

(bo pochodne są dodatnie). Stąd wynika

$$x_1 > \alpha.$$

Ponieważ zachodzi również $x_1 - b < 0$, więc

$$x_1 < b.$$

Zatem punkt x_1 leży rzeczywiście bliżej szukanego pierwiastka, niż punkt startowy b. W podobny sposób możemy pokazać, że ciąg generowany wzorem (5.50) jest malejącym ciągiem ograniczonym z dołu poprzez α, jest więc ciągiem zbieżnym (do pewnej liczby g). Przechodząc w (5.50) obustronnie do granicy $n \to \infty$, mamy

$$g = g - \frac{f(g)}{f'(g)} \quad \Rightarrow \quad f(g) = 0 \quad \Rightarrow \quad g = \alpha. \tag{5.55}$$

Błąd n–tego przybliżenia szacujemy korzystając z twierdzenia Lagrange'a o przyrostach, podobnie jak dla wcześniej prezentowanych metod. Ponieważ

$$f(x_n) - f(\alpha) = f'(c)(x_n - \alpha), \tag{5.56}$$

gdzie c leży między punktami x_n i α, więc

$$|x_n - \alpha| \leqslant \left|\frac{f(x_n)}{m}\right|, \quad m = \inf_{x\in\langle a,b\rangle} |f'(x)|. \tag{5.57}$$

Ze wzoru Taylora wynika

$$\begin{aligned} f(x_n) &\equiv f(x_{n-1} + (x_n - x_{n-1})) \\ &= f(x_{n-1}) + f'(x_{n-1})(x_n - x_{n-1}) + \frac{1}{2}f''(\xi_{n-1})(x_n - x_{n-1})^2, \end{aligned} \tag{5.58}$$

co po uwzględnieniu wzoru (5.50) prowadzi do

$$f(x_{n-1}) + f'(x_{n-1})(x_n - x_{n-1}) = 0. \tag{5.59}$$

x	$f(x)$	x	$f(x)$
$x_0 = 2$	3	$x_0 = 1$	-4
$x_1 = 1,76923$	0,36048	$x_1 = 3$	24
$x_2 = 1,73292$	0,00823	$x_2 = 2,2$	5,88800
$x_3 = 1,73205$	-0,000008	$x_3 = 1,83015$	0,98899
		$x_4 = 1,7578$	0,24782
		$x_5 = 1,73195$	-0,00095

Tabela 5.5: Metoda Newtona zastosowana do równania $x^3 + x^2 - 3x - 3 = 0$ dla dwóch różnych punktów startowych

Stąd wynika

$$|f(x_n)| \leqslant \frac{1}{2} M (x_n - x_{n-1})^2, \quad M = \sup_{x \in \langle a,b \rangle} |f''(x)|, \tag{5.60}$$

czyli również

$$|\alpha - x_n| \leqslant \frac{1}{2} \frac{M}{m} (x_n - x_{n-1})^2 = \frac{M}{2m} \left[\frac{f(x_n)}{f'(x_n)} \right]^2. \tag{5.61}$$

Podobnie, jak w przypadku metody regula falsi, dla x_n dostatecznie bliskich α możemy korzystać z oszacowania

$$|\alpha - x_n| \approx \left| \frac{f(x_{n-1})}{f'(x_{n-1})} \right|. \tag{5.62}$$

Metoda Newtona jest szybko zbieżna do szukanego pierwiastka, jeżeli tylko punkt startowy leży wystarczająco blisko niego. Aby się o tym przekonać, rozważmy następujący przykład.

Przykład Szukamy ponownie dodatniego pierwiastka równania

$$x^3 + x^2 - 3x - 3 = 0,$$

tym razem stosując metodę stycznych dla dwóch punktów startowych, $x_0 = 2$ i $x_0 = 1$. Wyniki zebrane są w Tabeli 5.5. Widzimy, że rzeczywiście, w przypadku punktu startowego $x_0 = 1$ proces iteracyjny jest zbieżny dużo wolniej.

Warunki, dla których metoda Newtona jest zawsze zbieżna, określa następujące twierdzenie:

Twierdzenie 5.1.2 *Jeżeli mamy przedział $\langle a, b \rangle$ taki, że:*

(i) $f(a)$ i $f(b)$ mają przeciwne znaki,

(ii) f'' jest ciągła i nie zmienia znaku na $\langle a, b \rangle$,

(iii) styczne do krzywej $y = f(x)$ poprowadzone w punktach o odciętych a i b przecinają oś OX wewnątrz przedziału $\langle a, b \rangle$,

wówczas równanie $f(x) = 0$ ma dokładnie jeden pierwiastek α w przedziale $\langle a, b \rangle$ i metoda Newtona jest zbieżna do α dla dowolnego punktu startowego $x_0 \in \langle a, b \rangle$.

Dowód w [57].

Przykład Metodę Newtona można zastosować np. do obliczania pierwiastka kwadratowego z liczby dodatniej c. Jest on bowiem rozwiązaniem równania

$$x^2 - c = 0.$$

Na podstawie wzoru (5.50) otrzymujemy (dla $x_n \neq 0$):

$$x_{n+1} = x_n - \frac{x_n^2 - c}{2x_n} = \frac{1}{2}\left(x_n + \frac{c}{x_n}\right). \tag{5.63}$$

Warunki powyższego twierdzenia są spełnione na przedziale $\langle a, b\rangle$ takim, że $0 < a < c^{1/2}$ oraz $b > \frac{1}{2}\left(a + \frac{c}{a}\right)$.

Poniższy kod stanowi przykład prostej implementacji metody Newtona w GNU Octave:

```
function [x,err,k,fx]=newton(f,df,x,delta,epsilon,maxit)
% newton.m
% Rozwiazanie rownania f(x)=0 metoda Newtona
% Wejscie:
%    f       - funkcja
%    df      - jej pochodna
%    x       - poczatkowe przyblizenie pierwiastka
%    delta   - dokladnosc rozwiazania
%    epsilon - dokladnosc f(x)
%    maxit   - maksymalna liczba iteracji
% Wyjscie:
%    x       - pierwiastek
%    err     - oszacowanie bledu
%    k       - ilosc iteracji
%    fx      - f(x)
for k=1:maxit
  x1=x-feval(f,x)/feval(df,x);
  err=abs(x1-x);
  x=x1;
  fx=feval(f,x);
  if (err<delta) | (abs(fx)<epsilon)
    break
  endif
endfor
```

5.1.8 Metoda iteracyjna Eulera

Niech x_n będzie aktualnym przybliżeniem poszukiwanego pierwiastka,

$$f(x_n + h) = 0, \tag{5.64}$$

gdzie h jest pewną małą liczbą. Rozwijając funkcję $f(x)$ w szereg Taylora wokół punktu x_n i pomijając wyrazy rzędu wyższego niż drugi otrzymamy

$$f(x_n) + hf'(x_n) + \frac{h^2}{2}f''(x_n) = 0, \quad h = x - x_n. \tag{5.65}$$

Metoda	Rząd
bisekcji	1
regula falsi	1
siecznych	$\frac{1}{2}(1+\sqrt{5}) \simeq 1,62$
Brenta	$\simeq 1,8$
Newtona	2
Eulera	3

Tabela 5.6: Rzędy poznanych metod w przypadku pierwiastków pojedynczych.

Jeśli $[f'(x_n)]^2 \geqslant 2f(x_n)f''(x_n)$, to powyższy trójmian kwadratowy ma pierwiastki rzeczywiste

$$h_n = -\frac{f'(x_n)}{f''(x_n)}\left(1 \pm \sqrt{1 - 2\frac{f(x_n)f''(x_n)}{[f'(x_n)]^2}}\right). \tag{5.66}$$

Biorąc mniejszy z nich, znajdziemy

$$x_{n+1} = x_n - u(x_n)\frac{2}{1+\sqrt{1-2t(x_n)}}, \tag{5.67}$$

gdzie

$$u(x) = \frac{f(x)}{f'(x)}, \quad t(x) = \frac{f(x)f''(x)}{[f'(x)]^2}. \tag{5.68}$$

Powyższy wzór to metoda iteracyjna Eulera [57]. W przypadku $|t| \ll 1$ możemy jeszcze skorzystać z przybliżenia

$$\frac{2}{1-\sqrt{2t(x_n)}} \simeq 1 + \frac{1}{2}t(x_n). \tag{5.69}$$

Wówczas

$$x_{n+1} = x_n - u(x_n)\left(1 + \frac{1}{2}t(x_n)\right). \tag{5.70}$$

5.1.9 Rząd metod

Definicja Mówimy, że metoda jest rzędu p, jeżeli istnieje stała K taka, że dla dwu kolejnych przybliżeń x_k i x_{k+1} zachodzi nierówność

$$|x_{k+1} - \alpha| \leqslant K|x_k - \alpha|^p. \tag{5.71}$$

Rząd metody oznacza szybkość jej zbieżności. W przypadku poznanych metod poszukiwania pojedynczych pierwiastków rzeczywistych ich rzędy zebrane są w tabeli 5.6 [50, 57].

5.1.10 Pierwiastki wielokrotne

W naszych rozważaniach do tej pory zakładaliśmy, że pierwiastek α jest pierwiastkiem jednokrotnym równania $f(x) = 0$. Spróbujmy ocenić przydatność omówionych w poprzednich paragrafach metod do poszukiwania pierwiastków wielokrotnych. Przy tym:

Definicja Liczbę α nazywamy r–krotnym pierwiastkiem równania $f(x) = 0$, jeżeli

$$0 < |g(\alpha)| < \infty, \quad g(x) = \frac{f(x)}{(x-\alpha)^r}. \tag{5.72}$$

Metoda bisekcji, regula falsi i metoda siecznych nadają się jedynie do poszukiwania pierwiastków o krotności nieparzystej, gdy spełniony jest warunek $f(a)f(b) < 0$. Przy tym w przypadku tej ostatniej metody obniża się rząd metody (szybkość zbieżności). Natomiast metoda Newtona może być stosowana dla krotności zarówno parzystych, jak i nieparzystych, jeśli tylko istnieje odpowiednie lewo- lub prawostronne sąsiedztwo szukanego pierwiastka, w którym znak $f'(x)$ i $f''(x)$ pozostaje stały. Jednak również tutaj obniża się rząd metody [50, 59].

Powyższe metody modyfikuje się często w celu poprawienia zbieżności. Jeżeli np. krotność pierwiastka jest znana, to metodę Newtona można zmodyfikować w sposób następujący:

$$x_{n+1} = x_n + rh_n, \quad h_n = -\frac{f(x_n)}{f'(x_n)}. \tag{5.73}$$

Jeżeli krotność nie jest znana, możemy postąpić inaczej. Załóżmy, że $f(x)$ ma r–tą pochodną ciągłą w otoczeniu pierwiastka α o krotności r. Wówczas

$$f^{(i)}(\alpha) = 0, \quad i < r \tag{5.74}$$

i ze wzoru Taylora wynika

$$\begin{aligned} f(x) &= \frac{1}{r!}(x-\alpha)^r f^{(r)}(\xi), \\ f'(x) &= \frac{1}{(r-1)!}(x-\alpha)^{r-1} f^{(r)}(\xi'), \end{aligned} \tag{5.75}$$

przy czym ξ i ξ' leżą w przedziale między x i α. Niech

$$u(x) = \frac{f(x)}{f'(x)}. \tag{5.76}$$

Zachodzi

$$\lim_{x\to\alpha} \frac{u(x)}{x-\alpha} = \frac{1}{r}, \tag{5.77}$$

czyli równanie $u(x) = 0$ ma pierwiastek pojedynczy α. Równanie $u(x) = 0$ można już rozwiązać wszystkimi omówionymi metodami. Np. metoda Newtona daje w tym przypadku

$$x_{n+1} = x_n - \frac{u(x_n)}{u'(x_n)}, \quad u'(x_n) = 1 - \frac{f''(x_n)}{f'(x_n)}u(x_n), \tag{5.78}$$

natomiast wzór siecznych prowadzi do

$$x_{n+1} = x_n - u(x_n)\frac{x_n - x_{n-1}}{u(x_n) - u(x_{n-1})}. \tag{5.79}$$

5.1.11 Przyspieszanie zbieżności

Zbieżność dowolnej metody rzędu 1 można przyspieszyć przy pomocy techniki nazywanej procesem Δ^2 Aitkena [59].

Definicja Dla ciągu $\{x_n\}_{n=0}^{\infty}$ różnicą skończoną w przód nazywamy

$$\Delta x_n = x_{n+1} - x_n, \quad n \geqslant 0. \tag{5.80}$$

Różnice wyższego rzędu określone są wzorem rekurencyjnym:

$$\Delta^k x_n = \Delta^{k-1}(\Delta x_n), \quad k \geqslant 2. \tag{5.81}$$

Twierdzenie 5.1.3 *Niech ciąg $\{x_n\}_{n=0}^{\infty}$ zbiega liniowo do granicy α i niech $\alpha - x_n \neq 0$ dla wszystkich $n \geqslant 0$. Jeśli istnieje liczba rzeczywista A taka, że $|A| < 1$ i*

$$\lim_{n\to\infty} \frac{\alpha - x_{n+1}}{\alpha - x_n} = A, \tag{5.82}$$

wówczas ciąg $\{y_n\}_{n=0}^{\infty}$ zdefiniowany jako

$$y_n = x_n - \frac{(\Delta x_n)^2}{\Delta^2 x_n} = x_n - \frac{(x_{n+1} - x_n)^2}{x_{n+2} - 2x_{n+1} + x_n} \tag{5.83}$$

zbiega do α szybciej niż $\{x_n\}_{n=0}^{\infty}$, tzn.

$$\lim_{n\to\infty} \frac{\alpha - y_n}{\alpha - x_n} = 0. \tag{5.84}$$

Dowód w [59].

Kiedy proces Aitkena połączony jest z metodą iteracyjną opartą na twierdzeniu o punkcie stałym, mówimy o algorytmie Steffensena. Używa się go często do poprawienia zbieżności metody Newtona w przypadku pierwiastków wielokrotnych.

5.2 Równania algebraiczne

Specjalnym rodzajem równań nieliniowych są równania wielomianowe (algebraiczne)

$$w(z) \equiv a_0 z^n + a_1 z^{n-1} + \ldots + a_{n-1} z + a_n = 0. \tag{5.85}$$

Zasadnicze twierdzenie algebry mówi, że równanie (5.85) ma dokładnie n pierwiastków, które mogą być rzeczywiste lub zespolone [60]. Jeśli współczynniki $a_0, a_1, \ldots, a_n$ są rzeczywiste, to ewentualne pierwiastki zespolone występują w parach sprzężonych.

Pierwiastków rzeczywistych wielomianów z rzeczywistymi współczynnikami możemy szukać opisanymi wcześniej metodami. Jeśli zależy nam również na znalezieniu pierwiastków zespolonych, lepiej w tym celu używać metod dedykowanych wielomianom.

5.2.1 Metoda Laguerre'a

Bardzo popularną metodą poszukiwania pierwiastków równań wielomianowych jest metoda Laguerre'a [15, 56, 57]. Jej dużą zaletą jest to, że pozwala wyliczyć pierwiastki rzeczywiste i zespolone, pojedyncze i wielokrotne. Wymaga jednak arytmetyki zespolonej.

Aby znaleźć pierwiastek wielomianu $w_n(z)$ stopnia n, przybliżamy go wielomianem

$$r(z) = a(z - p_1)(z - p_2)^{n-1}. \tag{5.86}$$

Parametry a, p_1 i p_2 dobieramy tak, aby

$$w_n(z_k) = r(z_k), w_n'(z_k) = r'(z_k), \quad w_n''(z_k) = r''(z_k). \tag{5.87}$$

Jeśli z_k jest pewnym przybliżeniem pojedynczego pierwiastka α, wówczas parametr p_1 będzie jego następnym oszacowaniem. Zauważmy, że dla

$$w(z) = (z - \alpha_1)(z - \alpha_2)\dots(z - \alpha_n) \tag{5.88}$$

zachodzi

$$S_1 \equiv \frac{w'(z)}{w(z)} = \sum_{i=1}^{n} \frac{1}{z - \alpha_i}. \tag{5.89}$$

Różniczkując obustronnie otrzymamy

$$-\frac{\mathrm{d}S_1(z)}{\mathrm{d}z} \equiv S_2(z) = \left(\frac{w'(z)}{w(z)}\right)^2 - \frac{w''(z)}{w(z)} = \sum_{i=1}^{n} \frac{1}{(z - \alpha_i)^2} \tag{5.90}$$

Warunek (5.87) prowadzi do

$$\begin{aligned} S_1(z_k) &= \frac{1}{z_k - p_1} + \frac{(n-1)}{z_k - p_2}, \\ S_2(z_k) &= \frac{1}{(z_k - p_1)^2} + \frac{(n-1)}{(z_k - p_2)^2}. \end{aligned} \tag{5.91}$$

Stąd ($p_1 = z_{k+1}$)

$$z_{k+1} = z_k - \frac{nw(z_k)}{w'(z_k) \pm \sqrt{H(z_k)}}, \tag{5.92}$$

gdzie

$$H(z_k) = (n-1)^2 \left[w'(z_k)\right]^2 - n(n-1)w(z_k)w''(z_k). \tag{5.93}$$

Znak we wzorze (5.92) powinniśmy wybrać tak, aby poprawka $|z_{k+1} - z_k|$ była jak najmniejsza.

Dla wielomianów, które mają tylko pierwiastki rzeczywiste, metoda Laguerre'a jest globalnie zbieżna [57]. W przypadku wielomianów z pierwiastkami zespolonymi zbieżności dla dowolnego punktu startowego nie można zagwarantować, ale w praktyce w większości przypadków metoda działa dobrze. W szczególności dla punktu startowego $z_0 = 0$ metoda pozwoli zazwyczaj znaleźć pierwiastek o najmniejszym module. Rząd metody Laguerre'a wynosi 3 dla pierwiastków pojedynczych i 1 dla wielokrotnych.

5.2.2 Macierz towarzysząca (ang. *companion matrix*)

Jak wiadomo, wartości własne macierzy $\mathbf{A}$ to pierwiastki wielomianu charakterystycznego

$$w(x) = \det[\mathbf{A} - x\mathbf{I}]. \tag{5.94}$$

W rozdziale 9 dowiemy się wprawdzie, że znalezienie pierwiastków tego wielomianu wprost nie jest najlepszym sposobem na wyznaczenie wartości własnych, ale istnieją inne wydajne i stabilne metody dedykowane zagadnieniom własnym. Jeżeli dysponujemy implementacją jednej z takich metod (a można je znaleźć w większości bibliotek numerycznych), to problem można odwrócić i poszukać takiej macierzy, której wartości własne byłyby pierwiastkami interesującego nas wielomianu, a następnie znaleźć te wartości [15].

Macierz taką możemy zbudować dla każdego wielomianu. Wystarczy bowiem, że weźmiemy

$$\mathbf{A} = \begin{pmatrix} -\frac{a_{n-1}}{a_n} & -\frac{a_{n-2}}{a_n} & \dots & -\frac{a_1}{a_n} & -\frac{a_0}{a_n} \\ 1 & 0 & \dots & 0 & 1 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \end{pmatrix} \tag{5.95}$$

Łatwo sprawdzić, że wielomian charakterystyczny macierzy $\mathbf{A}$ ma postać

$$x^n + \frac{a_{n-1}}{a_n}x^{n-1} + \frac{a_{n-2}}{a_n}x^{n-2} + \dots + \frac{a_1}{a_n}x + \frac{a_0}{a_n} = 0, \tag{5.96}$$

a więc rzeczywiście jest równoważny interesującemu nas wielomianowi.

5.2.3 Liczba pierwiastków rzeczywistych

W przypadku wielomianów o współczynnikach rzeczywistych często jesteśmy zainteresowani ich rzeczywistymi pierwiastkami. Aby oszacować ich liczbę, dla wielomianu $w(x)$ stopnia n utwórzmy ciąg:

$$w(x), w'(x), \dots, w^{(n)}(x). \tag{5.97}$$

Oznaczmy przez $M(x_0)$ liczbę zmian znaku w ciągu (5.97) w punkcie x_0. Zachodzi następujące twierdzenie:

Twierdzenie 5.2.1 *(Fouriera) Jeżeli $w(x)$ jest wielomianem stopnia n określonym w przedziale (a, b) oraz $w(a)w(b) \neq 0$, to liczba zer wielomianu w tym przedziale wynosi*

$$M(a) - M(b), \tag{5.98}$$

lub jest od tej liczby mniejsza o liczbę parzystą.

Dowód w [61].

Przykład Niech

$$w(x) = x^3 - 2x^2 - 5x + 5.$$

x	$-\infty$	∞	0	1	3
w	$-$	$+$	$+$	$-$	$+$
w'	$+$	$+$	$-$	$-$	$+$
w''	$-$	$+$	$-$	$+$	$+$
w'''	$+$	$+$	$+$	$+$	$+$
$M(x)$	3	0	2	1	0

Tabela 5.7: Zmiana znaków w przykładowym ciągu pochodnych.

Tworzymy ciąg pochodnych:

$$\begin{aligned} w'(x) &= 3x^2 - 4x - 5, \\ w''(x) &= 6x - 4, \\ w'''(x) &= 6. \end{aligned}$$

Zmiany znaków w tym ciągu dla wybranych punktów przedstawione są w tabeli 5.7 Ponieważ

$$M(-\infty) - M(\infty) = 3,$$

na mocy twierdzenia Fouriera wnioskujemy, że równanie $w(x) = 0$ ma jeden lub trzy pierwiastki rzeczywiste. Ponadto, ponieważ

$$\begin{aligned} M(-\infty) - M(0) &= 1, \\ M(0) - M(1) &= 1, \\ M(1) - M(3) &= 1, \end{aligned}$$

stwierdzamy, że równanie ma trzy pierwiastki, po jednym w każdym z przedziałów $(-\infty, 0)$, $(0, 1)$ i $(1, 3)$.

Twierdzenie Fouriera nie jest jedynym sposobem pozwalającym określić liczbę pierwiastków rzeczywistych wielomianu. Możemy również badać zmiany znaków w tzw. ciągu Sturma [62], jednak wymaga to dość uciążliwych obliczeń. Z drugiej jednak strony metoda pozwala określić liczbę pierwiastków dokładnie.

5.3 Układy równań nieliniowych

Poszukiwać teraz będziemy rozwiązań $\vec{\alpha} \in \mathbb{R}^n$ równania

$$F(\vec{x}) = 0, \tag{5.99}$$

przy czym teraz

$$F : \mathbb{R}^n \to \mathbb{R}^n. \tag{5.100}$$

W szczególnym przypadku, kiedy odwzorowanie F jest afiniczne[4], np.

$$F(\vec{x}) = \mathbf{A}\vec{x} + \vec{b},$$

[4]tzn. jest odwzorowaniem geometrycznym, które zachowuje współliniowość punktów (proste przekształca zawsze na proste)

uzyskujemy układ równań liniowych, który potrafimy już rozwiązać (patrz rozdział 4). Jeżeli F jest odwzorowaniem nieliniowym, rozwiązanie układu $F(\vec{x}) = 0$ znacznie się komplikuje. Ponadto brak ogólnego kryterium istnienia rozwiązania. Dlatego w naszych rozważaniach będziemy a priori zakładać, że rozwiązanie istnieje i omówimy pokrótce wybrane metody jego wyznaczenia [50].

5.3.1 Ogólne metody iteracyjne

Skonstruujmy ciąg punktów

$$\vec{x}^{(0)}, \quad \vec{x}^{(1)}, \quad \vec{x}^{(2)}, \quad \ldots \tag{5.101}$$

zbieżny do rozwiązania $\vec{\alpha}$ układu $F(\vec{x}) = 0$. Jeżeli można wskazać odwzorowanie G takie, że

$$\vec{x}^{(i)} = G(\vec{x}^{(i-1)}, \ldots, \vec{x}^{(i-p)}), \tag{5.102}$$

to metodę iteracyjną nazywamy metodą stacjonarną p–punktową. Jeżeli natomiast G ulega modyfikacjom podczas iteracji, to mówimy o metodzie niestacjonarnej (zmiennego operatora).

Definicja Niech $G: \ D \subset \mathbb{R}^n \times \ldots \times \mathbb{R}^n \to \mathbb{R}^n$. Punkt $\vec{\alpha}$ nazywamy punktem przyciągania metody iteracyjnej, jeżeli istnieje takie otoczenie $U_{\vec{\alpha}}$ tego punktu, że obierając punkty $\vec{x}^{(-p+1)}, \ldots, \vec{x}^{(0)}$ z tego otoczenia uzyskamy ciąg punktów $\vec{x}^{(1)}, \vec{x}^{(2)}, \ldots \in D$, zbieżny do $\vec{\alpha}$.

Definicja Odwzorowanie $G: \ \mathbb{R}^n \to \mathbb{R}^n$ nazywamy różniczkowalnym w sensie Frécheta w punkcie $\vec{x}$, jeżeli istnieje taka macierz $A: \ \mathbb{R}^n \to \mathbb{R}^n$, że

$$\lim_{\vec{h} \to 0} \frac{\|G(\vec{x} + \vec{h}) - G(\vec{x}) - A\vec{h}\|}{\|\vec{h}\|} = 0 \tag{5.103}$$

przy dowolnym sposobie wyboru wektorów $\vec{h} \to 0$. Macierz A nazywamy pochodną Frécheta odwzorowania G w punkcie $\vec{x}$ i oznaczamy ją przez $G'(\vec{x})$.

Twierdzenie 5.3.1 *Jeżeli pochodna Frécheta odwzorowania $G: \ \mathbb{R}^n \to \mathbb{R}^n$ w punkcie $\vec{\alpha}$ ma promień spektralny $\varphi(G'(\vec{\alpha})) = \beta < 1$ oraz $G(\vec{\alpha}) = \vec{\alpha}$, to $\vec{\alpha}$ jest punktem przyciągania metody iteracyjnej $\vec{x}^{(i+1)} = G(\vec{x}^{(i)})$.*

Dowód w [50].

Powyższe twierdzenie pozwala uzasadnić lokalną zbieżność wielu metod iteracyjnych.

5.3.2 Metoda Newtona

Podamy teraz uogólnienie metody stycznych na przypadek większej liczby równań.

Twierdzenie 5.3.2 *Niech funkcja $F(\vec{x})$ będzie różniczkowalna w sensie Frécheta w pewnym otoczeniu $K(\vec{\alpha}, r)$ punktu $\vec{\alpha}$, w którym $F(\vec{\alpha}) = 0$. Załóżmy, że pochodna $F'(\vec{x})$ jest ciągła w punkcie $\vec{\alpha}$, a pochodna $F'(\vec{\alpha})$ jest nieosobliwa. Wówczas punkt $\vec{\alpha}$ jest punktem przyciągania metody iteracyjnej Newtona,*

$$\vec{x}^{(i+1)} = \vec{x}^{(i)} - [F'(\vec{x}^{(i)})]^{-1} F(\vec{x}^{(i)}). \tag{5.104}$$

Ponadto, jeżeli ciągłość pochodnej w punkcie $\vec{\alpha}$ jest typu Höldera, tzn.

$$\|F'(\vec{x}) - F'(\vec{\alpha})\| \leqslant H\|\vec{x} - \vec{\alpha}\|^p, \quad \vec{x} \in K(\vec{\alpha}, r), \quad p \in (0, 1\rangle, \tag{5.105}$$

to

$$\|\vec{x}^{(i+1)} - \vec{\alpha}\| \leqslant C\|\vec{x}^{(i)} - \vec{\alpha}\|^{1+p}, \tag{5.106}$$

gdzie $C = 4H\|[F'(\vec{\alpha})]^{-1}\|$.

Metoda Newtona w przypadku układów równań nieliniowych wymaga silnych założeń dotyczących funkcji (ciągła różniczkowalność, nieosobliwość pochodnej w punkcie $\vec{\alpha}$), jednak ze względu na szybką zbieżność lokalną jest metodą szeroko stosowaną.

5.3.3 Metoda siecznych

Gdy liczba zmiennych n jest bardzo duża, obliczanie pochodnych $F'(\vec{x})$ odwzorowania F we wzorze Newtona (5.104) staje się uciążliwe. Wówczas warto jest zastosować uogólnienie metody siecznych na przypadek wielowymiarowy [50].

Uogólnienie to sprowadza się do przybliżenia odwzorowania F odwzorowaniem afinicznym w taki sposób, aby

$$F(\vec{y}^{(j)}) \simeq \mathbf{A}\vec{y}^{(j)} + \vec{b}, \quad j = 0, 1, \ldots, n, \tag{5.107}$$

a następnie do przyjęcia za przybliżenie rozwiązania $\vec{\alpha}$ rozwiązania liniowego układu równań $\mathbf{A}\vec{x} + \vec{b} = 0$.

Niech $\Delta\mathbf{Y}$ będzie macierzą o kolumnach $\vec{y}^{(1)} - \vec{y}^{(0)}$, $\vec{y}^{(2)} - \vec{y}^{(2)}$, $\ldots$, $\vec{y}^{(n)} - \vec{y}^{(n-1)}$, a $\Delta\mathbf{F}$ macierzą o kolumnach $F(\vec{y}^{(1)}) - F(\vec{y}^{(0)})$, $F(\vec{y}^{(2)}) - F(\vec{y}^{(2)})$, $\ldots$, $F(\vec{y}^{(n)}) - F(\vec{y}^{(n-1)})$. Przybliżenie (5.107) prowadzi do

$$\Delta\mathbf{F} = \mathbf{A}\Delta\mathbf{Y}. \tag{5.108}$$

Ponieważ rozwiązaniem układu równań $\mathbf{A}\vec{x} + \vec{b} = 0$ będzie

$$\vec{x} = -\mathbf{A}^{-1}\vec{b} = -\mathbf{A}^{-1}(F(\vec{y}) - \mathbf{A}\vec{y}) = \vec{y} - \mathbf{A}^{-1}F(\vec{y}), \tag{5.109}$$

więc rozwiązanie układu równań $F(\vec{x}) = 0$ metodą siecznych może przebiegać następująco:

1. wybieramy punkty $\vec{x}^{(-n)}$, $\vec{x}^{(-n+1)}$, $\ldots$, $\vec{x}^{(0)}$. Przyjmujemy $i = 0$,
2. obliczamy $\Delta\mathbf{F}^{(i)}$ oraz $\Delta\mathbf{Y}^{(i)}$ przyjmując

$$\vec{y}^{(0)} = \vec{x}^{(i-n)}, \quad \vec{y}^{(1)} = \vec{x}^{(i-n+1)}, \quad \ldots, \quad \vec{y}^{(n)} = \vec{x}^{(i)}, \tag{5.110}$$

3. wyznaczamy $\vec{x}^{(i+1)}$ ze wzoru

$$\vec{x}^{(i+1)} = \vec{x}^{(i)} - \Delta\mathbf{Y}^{(i)}[\Delta\mathbf{F}^{(i)}]^{-1}F(\vec{x}^{(i)}), \tag{5.111}$$

4. zwiększamy i o jeden i wracamy do kroku 2.

Jest to tzw. $(n+1)$–punktowa metoda siecznych. Można ją stosować, o ile tylko macierz $\Delta\mathbf{F}^{(i)}$ nie jest osobliwa. Jeżeli jest, musimy inaczej wybierać punkty $\vec{y}^{(j)}$, np. według wzoru

$$\begin{aligned} \vec{y}^{(n)} &= \vec{x}^{(i)} \\ \vec{y}^{(n-j)} &= \vec{x}^{(i)} + h\vec{e}^{(j)}, \quad j = 1, 2, \ldots, n, \end{aligned} \tag{5.112}$$

gdzie h jest pewną stałą, a $\vec{e}^{(j)}$ to wektory przestrzeni $\mathbb{R}^n$. W niektórych przypadkach może się jednak okazać, że nie jest możliwe takie dobranie punktów $\vec{y}^{(j)}$, aby macierz była nieosobliwa.

5.4 Funkcje biblioteczne - przykłady zastosowań

Mimo, że podstawowe algorytmy numerycznego rozwiązywania równań nieliniowych są stosunkowo łatwe do zaimplementowania na własną rękę, w bibliotekach numerycznych znajdziemy duży wybór gotowych funkcji przeznaczonych do tego celu.

5.4.1 C/C++

Narzędzia do poszukiwania miejsc zerowych funkcji jednej zmiennej w bibliotece GSL [12] zadeklarowane są w pliku nagłówkowym `gsl_roots.h`. Znajdziemy tam większość z algorytmów omówionych w tym rozdziale, m.in. metodę połowienia przedziału, falsi, Brenta i Newtona.

Procedura rozwiązywania równania na pierwszy rzut oka może się wydać nieco skomplikowaną. Użytkownik musi przede wszystkim zdefiniować funkcję określającą równanie i przekształcić ją w `gsl_function`. Następnie należy zainicjalizować obiekt typu

`gsl_root_fsolver`

lub

`gsl_root_fdfsolver`

dla wybranej metody. Tego pierwszego używamy w przypadku algorytmów, które nie wymagają wyliczania pierwszej pochodnej rozważanej funkcji. Pojedynczy krok iteracji możemy przeprowadzić za pomocą

`gsl_root_fsolver_iterate`

lub

`gsl_root_fdfsolver_iterate`

w zależności od metody, a zbieżność sprawdzimy, korzystając m.in. z

`gsl_root_test_interval`

Proszę zwrócić uwagę, że funkcje z biblioteki GSL pozwalają jedynie wykonać pewne czynności w jednym kroku iteracji. Natomiast przebieg procesu iteracyjnego leży w gestii programisty.

Przykład Chcemy rozwiązać równanie

$$x^2 - 5 = 0 \tag{5.113}$$

metodą Brenta z wykorzystaniem biblioteki GSL. Posłuży do tego poniższy program:

```
#include <iostream>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_roots.h>

double f(double x, void * params)
{
      return x*x-5;
}

int main()
{
  int status;
  int iter = 0, max_iter = 100;
  double r = 0;
  double x_lo = 0.0, x_hi = 5.0;
  gsl_function F;
  const gsl_root_fsolver_type *T;
  gsl_root_fsolver *s;

  F.function = &f;
  F.params = 0;

  T = gsl_root_fsolver_brent;
  s = gsl_root_fsolver_alloc (T);
  gsl_root_fsolver_set (s, &F, x_lo, x_hi);

  std::cout << "Metoda: " << gsl_root_fsolver_name (s) << "\n";
  std::cout << "iter\t"  << "r\t" << "dr\n";

do
{
  iter++;
  status = gsl_root_fsolver_iterate (s);
  r = gsl_root_fsolver_root (s);
  x_lo = gsl_root_fsolver_x_lower (s);
  x_hi = gsl_root_fsolver_x_upper (s);
  status = gsl_root_test_interval (x_lo,x_hi,0,0.001);
  if (status == GSL_SUCCESS)
        std::cout << "Zbiezna:\n";
  std::cout << iter << "\t" << r << "\t"
                <<  x_hi-x_lo << "\n";
}
while (status == GSL_CONTINUE && iter < max_iter);
return status;
}
```

Program skompilujemy poleceniem

```
g++ brent.cpp -lgsl -lgslcblas
```

Wynik jego działania jest następujący

```
Metoda: brent
```

```
iter     r        dr
1        1        4
2        3        2
3        2        1
4        2.2      0.8
5        2.23663  0.03663
Zbiezna:
6        2.23606  0.000566628
```

Jedną z niewątpliwych zalet biblioteki GSL jest łatwość, z jaką jeden algorytm można zastąpić innym. Aby rozwiązać równanie (5.113) metodą bisekcji, a nie jak wcześniej - Brenta, wystarczy w powyższym programie zmienić jedną linię z

```
T = gsl_root_fsolver_brent;
```

na

```
T = gsl_root_fsolver_bisection;
```

Reszta programu pozostaje bez zmian. Wynik jest następujący:

```
Metoda: bisection
iter     r        dr
1        1.25     2.5
2        1.875    1.25
3        2.1875   0.625
4        2.34375  0.3125
5        2.26562  0.15625
6        2.22656  0.078125
7        2.24609  0.0390625
8        2.23633  0.0195312
9        2.23145  0.00976562
10       2.23389  0.00488281
11       2.23511  0.00244141
Zbiezna:
12       2.23572  0.0012207
```

Przy okazji przekonaliśmy się, że zbieżność bisekcji jest dużo gorsza niż metody Brenta.

Biblioteka GSL zawiera również funkcje do rozwiązywania układów równań nieliniowych. Ich deklaracje znajdują się w pliku `gsl_multiroots.h`. Używa się ich podobnie, jak metod do poszukiwania miejsc zerowych funkcji jednej zmiennej.

5.4.2 Fortran

Jeżeli planujemy rozwiązywać równania nieliniowe w Fortranie, możemy skorzystać w tym celu z darmowej biblioteki Minpack [63]. Będziemy ją najprawdopodobniej musieli zainstalować na własną rękę. Użytkownicy systemów uniksowych znajdą odpowiednie instrukcje na stronie J. Burkardta [64].

Po zainstalowaniu Minpacka możemy m.in. skorzystać z funkcji `hybrd`, która stanowi implementację hybrydowej metody Powella [65], lub z jej nieco uproszczonej wersji `hybrd1`[5]:

[5] W rzeczywistości `hybrd1` stanowi jedynie nakładkę na `hybrd`. W `hybrd1` definiowane są wartości domyślne większości z 24(!) argumentów, z którymi należy wywołać procedurę `hybrd`.

```
hybrd1(fcn,n,x,fvec,tol,info)
```

Procedura ta rozwiązuje układ n nieliniowych ze względu na n zmiennych, zdefiniowany w funkcji `fnc`. Wektor `x` jest punktem startowym na wejściu i oszacowaniem pierwiastka na wyjściu. Argument `fvec` to wektor wartości funkcji dla znalezionego oszacowania. Parametr `tol` określa maksymalny błąd względny, jaki jesteśmy gotowi popełnić przy obliczeniu pierwiastka, a w zmiennej `info` zakodowany jest komunikat błędu.

Przykład Rozwiążmy ponownie równanie (5.113), tym razem korzystając z Fortrana i procedury `hybrd1`. Posłuży do tego następujący program:

```
program roots
  implicit none
  integer, parameter :: n=1
  real(kind = 8) :: fvec
  integer :: info
  real(kind=8) :: tol
  real(kind=8) :: x
  external fun

  x = 0.0D+00
  tol = 0.00001D+00

  call hybrd1(fun,n,x,fvec,tol,info)

  write(6,'(a,11x,a)') 'x','dx'
  write(6,'(f10.8,x,e16.8)') x, fvec
end


subroutine fun (n,x,fvec,iflag)
  implicit none
  integer :: n
  real(kind=8) :: fvec
  integer :: iflag
  real(kind=8) :: x
  fvec = x**2 - 5.0D0
  return
end
```

Skompilujemy go poleceniem:

```
gfortran roots.f90 -L/home/szwabin/lib -lminpack
```

Katalog, w którym znajduje się biblioteka, zależy oczywiście od sposobu, w jaki ją zainstalowaliśmy. Po uruchomieniu programu otrzymamy:

```
x           dx
2.23606798  -0.15942941E-11
```

5.4.3 Python

Korzystając z SciPy [16], narzędzia do rozwiązywania równań nieliniowych znajdziemy w module `optimize`. Funkcja `fsolve` to nakładka na procedurę `hybrd`

z biblioteki Minpack [63] (patrz par. 5.4.2). Dodatkowo mamy do dyspozycji kilka funkcji przeznaczonych do poszukiwania miejsc zerowych funkcji jednej zmiennej, m.in. `bisect` (bisekcja), `brentq` (metoda Brenta), `newton` (metoda Newtona).

Przykład Naszym celem jest rozwiązanie układu równań

$$\begin{aligned} -2x^2 + 3xy + 4\sin(y) &= 6, \\ 3x^2 - 3xy^2 + 3\cos(x) &= -4, \end{aligned} \tag{5.114}$$

w Pythonie z wykorzystaniem funkcji `fsolve`:

```
>>> from scipy import *
>>> from optimize import fsolve
>>> def func(x):
...    out = [-2*x[0]**2+3*x[0]*x[1]+4*sin(x[1])-6]
...    out.append(3*x[0]**2-2*x[0]*x[1]**2+3*cos(x[0])+4)
...    return out
...
>>> fsolve(func,x0)
array([ 0.57982909,  2.54620921])
>>> func(_)
[-1.5020162891232758e-10, 1.0563683261466394e-10]
```

5.4.4 GNU Octave

Do rozwiązywania równań nieliniowych i ich układów służy w GNU Octave funkcja `fsolve`

```
[x,info,msg] = fsolve(fcn,x0)
```

która na wejściu oczekuje nazwy `fnc` funkcji definiującej równania oraz punktu startowego `x0`. Podobnie, jak `fsolve` w Pythonie, funkcja wykorzystuje procedurę `hybrd` z biblioteki Minpack [63].

Przykład Rozwiążmy ponownie układ (5.114):

```
octave:1> function y = f(x)
> y(1) = -2*x(1)^2+3*x(1)*x(2)+4*sin(x(2))-6;
> y(2) = 3*x(1)^2-2*x(1)*x(2)^2+3*cos(x(1))+4;
> endfunction
octave:3> x0 = [1;2];
octave:4> [x,info] = fsolve("f",x0)
x =

  0.57983
  2.54621

info = 1
```

Przy pomocy funkcji `perror` sprawdzimy, co oznacza `info=1`:

```
octave:5> perror("fsolve",1)
solution converged to requested tolerance
```

A zatem w tym wypadku założona dokładność została osiągnięta. Powyższy komunikat możemy również odczytać bezpośrednio, przy wywołaniu funkcji `fsolve`:

```
octave:6> [x,info,msg] = fsolve("f",x0)
x =

  0.57983
  2.54621

info = 1
msg = solution converged within specified tolerance
```

GNU Octave oferuje również szereg funkcji do manipulowania wielomianami. Wśród nich znajdziemy m.in. procedurę `compan`, która utworzy macierz towarzyszącą wielomianu na podstawie wektora jego współczynników.

Przykład Szukamy wszystkich miejsc zerowych wielomianu

$$w(x) = \frac{147}{60}x^6 - 6x^5 + \frac{15}{2}x^4 - \frac{20}{3}x^3 + \frac{15}{4}x^2 - \frac{6}{5}x + \frac{1}{6} \tag{5.115}$$

korzystając z poznanej metody macierzowej:

```
octave:1> wiel = [147/60 -6 15/2 -20/3 15/4 -6/5 1/6];
octave:2> pierwiastki = eig(compan(wiel))
pierwiastki =

  0.14527 + 0.85107i
  0.14527 - 0.85107i
  1.00000 + 0.00000i
  0.37615 + 0.28847i
  0.37615 - 0.28847i
  0.40612 + 0.00000i
```

Sprawdźmy otrzymany wynik:

```
octave:6> polyval(wiel,pierwiastki(4))
ans = -5.5511e-17 + 3.3204e-19i
```

Rzeczywiście wartość wielomianu w tym punkcie można traktować jako 0.

Rozdział 6

Interpolacja i aproksymacja

6.1 Interpolacja

6.1.1 Funkcje biblioteczne - przykłady zastosowania

C/C++

Biblioteka GSL [12] oferuje wiele przydatnych funkcji do interpolowania danych dyskretnych. Ich definicje znajdziemy w plikach nagłówkowych `gsl_interp.h` i `gsl_spline.h`.

Funkcja interpolująca dla danego zbioru punków przechowywana jest w obiekcie `gsl_interp`. Obiekt ten dla określonego typu interpolacji tworzymy za pomocą

```
gsl_interp * gsl_interp_alloc(const gsl_interp_type * T
        size_t size)
```

Do inicjalizacji obiektu `gsl_interp` służy polecenie

```
int gsl_interp.init(gsl_interp * interp, const double xa[],
        const double ya[],size_t size)
```

gdzie `xa` i `ya` to tablice o rozmiarze `size`, w których przechowywane są odcięte i rzędne węzłów interpolacji. Do dyspozycji mamy kilka algorytmów, m.in.:

- interpolację przedziałami liniową, `gsl_interp_linear`,
- interpolację wielomianową, `gsl_interp_polynomial`,
- naturalne funkcje sklejane, `gsl_interp_cspline`.

Po znalezieniu funkcji interpolującej do obliczenia jej wartości w dowolnym punkcie x służy

```
double gsl_interp_eval(const gsl_interp * interp,
        const double xa[],      const double ya[], double x,
        gsl_interp_accel * acc)
```

gdzie `gsl_interp_accel` to pewnego rodzaju iterator przyspieszający obliczenia. Podobnie, do obliczenia pierwszej i drugiej pochodnej funkcji interpolującej możemy skorzystać z

rok	1850	1950	1960	1970	1985	2000	2025
ludność (mld)	1,402	2,486	3,014	3,683	4,842	6,127	8,177

Tabela 6.1: Liczba ludności w latach 1850-2025.

```
double gsl_interp_eval_deriv (const gsl_interp * interp,
        const double xa[], const double ya[], double x,
        gsl_interp_accel * acc)
double gsl_interp_eval_deriv2 (const gsl_interp * interp,
        const double xa[], const double ya[], double x,
        gsl_interp_accel * acc)
```

Przykład W tabeli 6.1 podana jest liczba ludności na Ziemi w latach 1850-2025 (szacowanie):
Chcemy wyznaczyć wielomian interpolujący te dane. Posłuży do tego program

```
#include <iostream>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_interp.h>

int main()
{
   int n=7;
   double x,y;
   double lata[]={1850.0, 1950.0,1960.0,1970.0,1985.0,2000.0,
                  2025.0};
   double lud[]={1.402,2.486,3.014,3.683,4.842,6.127,8.177};

   std::cout <<  "# lata\t ludnosc\n";
   gsl_interp_accel *acc
           = gsl_interp_accel_alloc ();
   gsl_interp *poly
           = gsl_interp_alloc (gsl_interp_polynomial,n);
   gsl_interp_init (poly,lata,lud,n);

   for (int i=0;i<175;i++)
     {
        x=lata[0]+i;
        y=gsl_interp_eval(poly,lata,lud,x,acc);
        std::cout << x << "\t" << y << "\n";
     }
   gsl_interp_free (poly);
   gsl_interp_accel_free (acc);
}
```

Skompilujemy i wywołamy go w następujący sposób

```
g++ ludnosc_poly.cpp -lgsl -lgslcblas
./a.out >> ludnosc_poly.dat
```

Zamiast wielomianu możemy szukać funkcji sklejanych interpolujących nasze dane. W tym celu modyfikujemy powyższy program:

```
#include <iostream>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_spline.h>

int main()
{
   int n=7;
   double x,y;
   double lata[]={1850.0, 1950.0,1960.0,1970.0,1985.0,2000.0,
                  2025.0};
   double lud[]={1.402,2.486,3.014,3.683,4.842,6.127,8.177};

   std::cout <<  "# lata\t ludnosc\n";
   gsl_interp_accel *acc
           = gsl_interp_accel_alloc ();
   gsl_spline *spline
           = gsl_spline_alloc (gsl_interp_cspline,n);
   gsl_spline_init (spline,lata,lud,n);

   for (int i=0;i<175;i++)
     {
        x=lata[0]+i;
        y=gsl_spline_eval(spline,x, acc);
        std::cout << x << "\t" << y << "\n";
     }
   gsl_spline_free (spline);
   gsl_interp_accel_free (acc);
}
```

Kompilacja i wywołanie są podobne:

```
g++ ludnosc_spline.cpp -lgsl -lgslcblas
./a.out >> ludnosc_spline.dat
```

Obie funkcje interpolujące przedstawione są na wykresie 6.1. Jak widzimy, wielomian interpolujący ma w okolicach roku 1870 maksimum rzędu 8,5 miliona. Oczywiście w tamtych czasach liczba mieszkańców naszego globu była znacznie niższa. To maksimum to wspominany już efekt Rungego występujący przy interpolacji wielomianami zbyt wysokich rzędów (patrz par.**??**). Dlatego w tym przypadku lepiej jest korzystać z funkcji sklejanych.

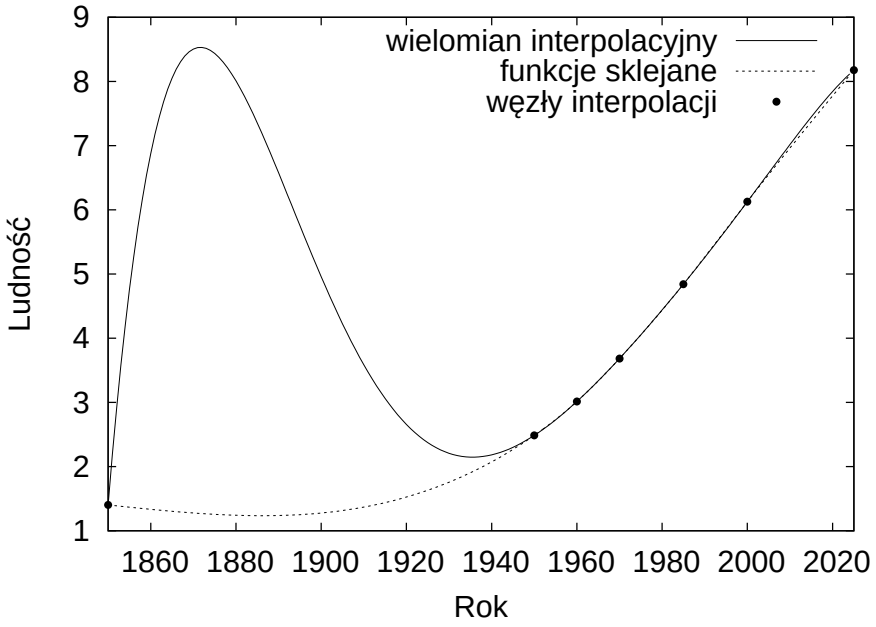
Fortran

Python

GNU Octave

6.2 Aproksymacja

6.2.1 Funkcje biblioteczne - przykłady zastosowania



Rysunek 6.1: Interpolacja danych z tabeli 6.1 wielomianem i funkcjami sklejanymi

Rozdział 7

Całkowanie numeryczne funkcji

Rozdział 8

Różniczkowanie numeryczne

Inaczej niż w teorii, w obliczeniach numerycznych różniczkowanie stanowi na ogół większe wyzwanie od całkowania. I nie chodzi tu wcale o stopień komplikacji algorytmów do wyliczania pochodnych, bo te akurat są względnie proste. Problem leży gdzie indziej. Aby go sobie uzmysłowić, przypomnijmy, że pochodna funkcji $f(x)$ zdefiniowana jest jako granica ilorazu różnicowego [66],

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{8.1}$$

Pierwsza trudność to przejście graniczne $h \to 0$. Ponieważ nie można go zrealizować na komputerze, naturalnym odruchem jest przybliżenie pochodnej za pomocą ilorazu różnicowego

$$f'(x) \simeq \frac{f(x+h) - f(x)}{h} \tag{8.2}$$

przy założeniu, że h jest dostatecznie małe. Intuicja podpowiada, że im mniejsze h, tym lepiej przybliżymy pochodną wzorem (8.2). Ale tutaj napotykamy na kolejny problem związany z ograniczeniami typowymi dla arytmetyki zmiennopozycyjnej na maszynach liczących (patrz rozdział 2). Po pierwsze, nie da się na komputerze przedstawić dowolnie małej liczby. Po drugie, jeżeli h będzie zbyt małe w porównaniu z x, to w argumencie $x + h$ może dojść do zaniedbania składnika (paragraf 2.3.6). Po trzecie, dla małych h wartości $f(x+h)$ i $f(x)$ mogą tak nieznacznie różnić się od siebie, że przy odejmowaniu dojdzie do utraty cyfr znaczących (par. 2.3.7).

Konflikt pomiędzy błędami zaokrągleń a obcięcia powoduje, że różniczkowanie numeryczne nie jest operacją dokładną. Nie uda nam się nigdy wyznaczyć pochodnej z taką samą precyzją, z jaką wyliczamy wartości różniczkowanej funkcji.

Funkcja może być zadana na dwa sposoby: albo mamy do dyspozycji algorytm (wzór matematyczny) pozwalający wyliczyć jej wartość w dowolnym punkcie, albo dysponujemy zbiorem dyskretnych punktów (x_i, y_i), $i = 0, 1, \ldots, n$. W obu przypadkach mamy do czynienia ze skończoną ilością punktów, z których musimy wyznaczyć pochodną.

8.1 Różnice skończone

8.1.1 Przybliżenia pierwszego i drugiego rzędu w h

Wzór (8.2) to przykład przybliżenia pochodnej za pomocą różnicy skończonej. Chociaż w jego przypadku wyszliśmy od matematycznej definicji pochodnej, wygodniejszym (i bardziej systematycznym) sposobem na znajdowanie odpowiednich przybliżeń są szeregi Taylora [66].

Rozważmy następujące rozwinięcia funkcji $f(x)$ w szereg Taylora wokół punktu x:

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \frac{f^{(3)}(x)}{3!}h^3 + \ldots, \tag{8.3}$$

$$f(x-h) = f(x) - f'(x)h + \frac{f''(x)}{2!}h^2 - \frac{f^{(3)}(x)}{3!}h^3 + \ldots, \tag{8.4}$$

$$f(x+2h) = f(x) + 2f'(x)h + 4\frac{f''(x)}{2!}h^2 + 8\frac{f^{(3)}(x)}{3!}h^3 + \ldots, \tag{8.5}$$

$$f(x-2h) = f(x) - 2f'(x)h + 4\frac{f''(x)}{2!}h^2 - 8\frac{f^{(3)}(x)}{3!}h^3 + \ldots \tag{8.6}$$

Przydatne okażą się również sumy i różnice powyższych szeregów:

$$f(x+h) + f(x-h) = 2f(x) + h^2 f''(x) + \frac{h^4}{12}f^{(4)}(x) + \ldots, \tag{8.7}$$

$$f(x+h) - f(x-h) = 2hf'(x) + h^2 f''(x) + \frac{h^3}{3}f^{(3)}(x) + \ldots, \tag{8.8}$$

$$f(x+2h) + f(x-2h) = 2f(x) + 4h^2 f''(x) + \frac{4h^4}{3}f^{(4)}(x) + \ldots, \tag{8.9}$$

$$f(x+2h) - f(x-2h) = 4hf'(x) + \frac{8h^3}{3}f^{(3)}(x) + \ldots \tag{8.10}$$

Rozwiązując równanie (8.3) względem $f'(x)$ otrzymamy znany już wzór (8.2),

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h), \tag{8.11}$$

z tą różnicą, że teraz dostaliśmy również informację o popełnionym błędzie obcięcia. Przybliżenie (8.11) nazywane bywa wzorem dwupunktowym „w przód". Podobny wzór, tym razem „w tył", możemy wyprowadzić z rozwinięcia (8.4):

$$f'(x) = \frac{f(x) - f(x-h)}{h} + O(h). \tag{8.12}$$

Zauważmy, że w obu przybliżeniach popełniamy błąd obcięcia, który maleje liniowo z h. Wynika stąd, że h musi być naprawdę bardzo małe, aby przybliżenie było w miarę dokładne.

Rozwiążmy teraz równanie (8.8) względem $f'(x)$. Otrzymamy tzw. wzór trójpunktowy (różnicę centralną),

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2), \tag{8.13}$$

którego błąd obcięcia maleje jak h^2. Innymi słowy, korzystając z różnicy centralnej nie musimy używać aż tak małego kroku, jak przy wzorach dwupunktowych, aby osiągnąć zbliżoną dokładność.

	$f(x+4h)$	$f(x+3h)$	$f(x+2h)$	$f(x+h)$	$f(x)$
$hf'(x)$				1	-1
$h^2f''(x)$			1	-2	1
$h^3f^{(3)}(x)$		1	-3	3	-1
$h^4f^{(4)}(x)$	1	-4	6	-4	1

Tabela 8.1: Współczynniki przybliżeń pochodnych różnicami skończonymi „w przód" (klasa $O(h)$).

	$f(x)$	$f(x-h)$	$f(x-2h)$	$f(x-3h)$	$f(x-4h)$
$hf'(x)$	1	-1			
$h^2f''(x)$	1	-2	1		
$h^3f^{(3)}(x)$	1	-3	3	-1	
$h^4f^{(4)}(x)$	1	-4	6	-4	1

Tabela 8.2: Współczynniki przybliżeń pochodnych różnicami skończonymi „w tył" (klasa $O(h)$).

W podobny sposób możemy szukać przybliżeń dla pochodnych wyższych rzędów. Na przykład z równań (8.3) i (8.5) wynika

$$f''(x) = \frac{f(x+2h) - 2f(x+h) + f(x)}{h^2} + O(h), \tag{8.14}$$

a z równania (8.7)

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2), \tag{8.15}$$

W tabelach 8.1 i 8.2 zestawione są przybliżenia pochodnych do czwartego rzędu włącznie różnicami w przód i w tył, natomiast w tabeli 8.3 - różnicami centralnymi [56].

8.1.2 „Generator" wzorów na pochodne

W przypadku wzorów na wyższe pochodne manipulowanie różnymi rozwinięciami funkcji w szereg Taylora może być nieco czasochłonne. Odpowiednie wzory można jednak wygenerować automatycznie [35].

Dla uproszczenia załóżmy, że naszym celem jest przybliżenie drugiej pochodnej wzorem, do którego wchodzą wartości

$$f(x+2h),\ f(x+h),\ f(x),\ f(x-h),\ f(x-2h)$$

zadanej funkcji $f(x)$. Będzie on zatem postaci

$$D^{(2)}_{c4}(x,h) = \frac{c_2 f(x+2h) + c_1 f(x+h) + c_0 f(x) + c_{-1} f(x-h) + c_{-2} f(x-2h)}{h^2}. \tag{8.16}$$

	$f(x+2h)$	$f(x+h)$	$f(x)$	$f(x-h)$	$f(x-2h)$
$hf'(x)$		1	0	-1	
$h^2f''(x)$		1	-2	1	
$h^3f^{(3)}(x)$	1	-2	0	2	-1
$h^4f^{(4)}(x)$	1	-4	6	-4	1

Tabela 8.3: Współczynniki przybliżeń pochodnych centralnymi różnicami skończonymi (klasa $O(h^2)$).

Po podstawieniu rozwinięć (8.3)-(8.6) do wzoru (8.16) otrzymamy

$$\begin{aligned} D_{c4}^{(2)}(x,h) &= \frac{1}{h^2}\left[(c_2+c_1+c_0+c_{-1}+c_{-2})f(x)+\right. \\ &\quad h(2c_2+c_1-c_{-1}-2c_{-2})f'(x)+ \\ &\quad h^2\left(\frac{2^2}{2}c_2+\frac{1}{2}c_1+\frac{1}{2}c_{-1}+\frac{2^2}{2}c_{-2}\right)f^{(2)}(x)+ \\ &\quad h^3\left(\frac{2^3}{3!}c_2+\frac{1}{3!}c_1-\frac{1}{3!}c_{-1}-\frac{2^3}{3!}c_{-2}\right)f^{(3)}(x)+ \\ &\quad \left. h^4\left(\frac{2^4}{4!}c_2+\frac{1}{4!}c_1+\frac{1}{4!}c_{-1}+\frac{2^4}{4!}c_{-2}\right)f^{(4)}(x)+\ldots\right]. \end{aligned} \tag{8.17}$$

Aby $D_{c4}^{(2)}(x,h)$ rzeczywiście przybliżało drugą pochodną, współczynniki przy $f(x)$, $f'(x)$, $f^{(3)}(x)$ i $f^{(4)}(x)$ muszą znikać. Zatem znalezienie wzoru na przybliżenie pochodnej sprowadza się do rozwiązania następującego układu równań:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 0 & -1 & -2 \\ \frac{2^2}{2!} & \frac{1}{2!} & 0 & \frac{1}{2!} & \frac{2^2}{2!} \\ \frac{2^3}{3!} & \frac{1}{3!} & 0 & -\frac{1}{3!} & -\frac{2^3}{3!} \\ \frac{2^4}{4!} & \frac{1}{4!} & 0 & \frac{1}{4!} & \frac{2^4}{4!} \end{pmatrix} \begin{pmatrix} c_2 \\ c_1 \\ c_0 \\ c_{-1} \\ c_{-2} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}. \tag{8.18}$$

Uogólnienie tej metody nie powinno sprawiać problemów (patrz [67]). Jej zaletą jest stosunkowo prosta implementacja [35]:

```
function [c,err,eoh,A,b] = diffapx(N,points)
  %diffapx.m - współczynniki przybliżenia pochodnej N-tego rzędu
  %różnicami skończonymi
  l = max(points);
  L = abs(points(1)-points(2))+1;
  if L<N+1
    error('Potrzebnych więcej punktów');
  endif
  for n= 1:L
    A(1,n) = 1;
    for m=2:(L+2)
      A(m,n) = A(m-1,n)*l/(m-1);
    endfor
    l=l-1;
```

```
  endfor
  b = zeros(L,1); b(N+1) = 1;
  c = (A(1:L,:)\b)'; %szukane współczynniki
  err = A(L+1,:)*c'; eoh = L-N; % szacowanie bledu
  if abs(err) < eps
    err=A(L+2,:)*c'; eoh = L-N+1;
  endif
  if(points(1)<points(2))
    c=fliplr(c);
  endif
endfunction
```

Funkcja `diffapx` przyjmuje dwa argumenty: `N` jest rzędem pochodnej, a wektor `points` określa, które wartości funkcji mają wchodzić do wzoru na pochodną numeryczną. Stosowana jest przy tym konwencja $f(x+i*h) = f_i$, a w wektorze podajemy tylko indeksy skrajnych wartości. W zależności od wywołania funkcja zwraca albo wektor współczynników `c`, albo dodatkowo jeszcze współczynnik (`err`) i rząd błędu (`eoh`) oraz macierz współczynników i wektor wyrazów wolnych z równania (8.18).

W celu sprawdzenia działania funkcji poszukajmy wzoru na czwartą pochodną, do którego wchodzą wartości $f(x+2h)$, $f(x+h)$, $f(x)$, $f(x-h)$ i $f(x-2h)$:

```
octave:4> diffapx(4,[2 -2])
ans =

   1.00000   -4.00000    6.00000   -4.00000    1.00000
```

Jak widzimy, wynik zgadza się z wartościami współczynników umieszczonymi w tabeli 8.3.

8.1.3 Błąd przybliżenia pierwszej pochodnej

Zgodnie z naszym oczekiwaniem błąd obcięcia popełniany w przybliżeniu pochodnej różnicami skończonymi maleje z h. Niestety, o czym wspominałem we wstępie do tego rozdziału, nie możemy wziąć h dowolnie małego, ponieważ im mniejsze h, tym większą rolę odgrywają błędy zaokrągleń. Przekonamy się o tym, rozważając następujący przykład.

Przykład W tabeli 8.4 przedstawione są wyniki numerycznego obliczenia pochodnej

$$\left.\frac{d\sin x}{dx}\right|_{x=1} (= 0.540302)$$

w różnych kroków h. Początkowo dokładność obliczeń rośnie w miarę zmniejszania h, przy czym zgodnie z oczekiwaniami zbieżność wzoru trójpunktowego jest szybsza niż dwupunktowych. Nie udaje się jednak poprawiać dokładności w "nieskończoność". Po osiągnięciu pewnej wartości granicznej h dokładność przybliżenia zaczyna się pogarszać. Spowodowane jest to właśnie skończoną precyzją obliczeń i błędami zaokrąglenia.

Aby oszacować całkowity błąd, jaki popełniamy zastępując pochodną różnicą skończoną, przypuśćmy, że wartości

$$f(x+2h),\ f(x+h),\ f(x),\ f(x-h),\ f(x-2h)$$

h	dwupunktowy (przód)		dwupunktowy (tył)		trójpunktowy	
	f'	$\lvert E(f')\rvert$	f'	$\lvert E(f')\rvert$	f'	$\lvert E(f')\rvert$
0,5	0,312048	0,228254	0,724091	0,183789	0,518069	0,022233
0,1	0,497364	0,042938	0,581440	0,041138	0,539402	0,000900
0,05	0,519046	0,021257	0,561109	0,020806	0,540077	0,000225
0,01	0,536090	0,004212	0,544500	0,004198	0,540295	0,000007
0,005	0,538206	0,002096	0,542402	0,002100	0,540304	0,000002
0,001	0,539958	0,000344	0,540674	0,000371	0,540316	0,000014
0,0005	0,540257	0,000046	0,540376	0,000073	0,540316	0,000014
0,0001	0,540614	0,000312	0,540018	0,000284	0,540316	0,000014
0,00005	0,540018	0,000284	0,540018	0,000284	0,540018	0,000284
0,00001	0,542402	0,002100	0,536442	0,003861	0,539422	0,000880

Tabela 8.4: Numeryczna pochodna funkcji $f(x) = \sin x$ w punkcie $x = 1$ dla różnych kroków h.

zaokrąglane są do [35]

$$\begin{aligned} y_2 &= f(x+2h) + e_2, \\ y_1 &= f(x+h) + e_1, \\ y_0 &= f(x) + e_0, \\ y_{-1} &= f(x-h) + e_{-1}, \\ y_{-2} &= f(x-2h) + e_{-2}, \end{aligned} \tag{8.19}$$

gdzie wartości błędów zaokrągleń e_i są mniejsze od pewnej dodatniej liczby ϵ,

$$|e_i| \leqslant \epsilon. \tag{8.20}$$

Wówczas

$$\begin{aligned} D_{f1}(x,h) &\equiv \frac{y_1 - y_0}{h} + O(h) = \frac{f(x+h) + e_1 - f(x) - e_0}{h} + O(h) \\ &= f'(x) + \frac{e_1 - e_0}{h} + \frac{K_1}{2}h, \end{aligned} \tag{8.21}$$

gdzie $K_1 = f''(x)$. Całkowity błąd przybliżenia wynosi zatem

$$|D_{f1}(x,h) - f'(x)| \leqslant \left|\frac{e_1 - e_0}{h}\right| + \frac{|K_1|}{2}h \leqslant \frac{2\epsilon}{h} + \frac{|K_1|}{2}h. \tag{8.22}$$

Powyższa nierówność to tzw. dylemat wyboru kroku. Mały krok oznacza mały błąd obcięcia i duży zaokrągleń. I odwrotnie - duży krok to zaniedbywalne błędy zaokrągleń i duże błędy obcięcia. Dylemat ten możemy rozwiązać wybierając h, dla którego zachodzi

$$\frac{\mathrm{d}}{\mathrm{d}h}\left(\frac{2\epsilon}{h} + \frac{|K_1|}{2}h\right) = -\frac{2\epsilon}{h^2} + \frac{K_1}{2} = 0. \tag{8.23}$$

Rozwiązując względem h otrzymamy

$$h_0 = 2\sqrt{\frac{\epsilon}{|K_1|}}. \tag{8.24}$$

W podobny sposób znajdziemy oszacowanie błędu dla różnicy centralnej:

$$\begin{aligned} D_{c2}(x,h) &\equiv \frac{y_1 - y_{-1}}{2h} + O(h^2) = f'(x) + \frac{e_1 - e_{-1}}{2h} + \frac{K_2}{6}h^2, \\ |D_{c2}(x,h) - f'(x)| &\leqslant \frac{\epsilon}{h} + \frac{|K_2|}{6}h^2, \end{aligned} \tag{8.25}$$

gdzie $K_2 = f'''(x)$. W tym wypadku optymalny krok będzie wynosił

$$h_0 = \sqrt[3]{\frac{3\epsilon}{|K_2|}}. \tag{8.26}$$

Analogicznie postępujemy dla pozostałych wzorów.

Tym sposobem dylemat wyboru kroku został rozwiązany. Niestety, tylko teoretycznie, ponieważ powyższe wyrażenia na optymalny krok są najczęściej bezużyteczne w praktyce. Zawierają one pochodne wyższych rzędów, o których zazwyczaj nie mamy żadnych informacji.

8.1.4 Ekstrapolacja Richardsona

Jak już zostało wspomniane, różnicę centralną dla pierwszej pochodnej otrzymamy ze wzoru (8.8),

$$D_{c2}(x,h) = \frac{f(x+h) - f(x-h)}{2h} = f'(x) + O(h^2). \tag{8.27}$$

Rozważmy teraz wyrażenie

$$2^2 D_{c2}(x,h) - D_{c2}(x,2h) = 4\frac{f(x+h) - f(x-h)}{2h} - \frac{f(x+2h) - f(x-2h)}{4h}. \tag{8.28}$$

Na mocy wzorów (8.8) i (8.10) mamy

$$D_{c4}(x,h) \equiv 2^2 D_{c2}(x,h) - D_{c2}(x,2h) = 3f'(x) + O(h^4). \tag{8.29}$$

Udało nam się zatem z dwóch przybliżeń klasy $O(h^2)$ otrzymać wzór o dokładności $O(h^4)$. Tę procedurę, zwaną ekstrapolacją Richardsona, można oczywiście kontynuować. Ogólnie dysponując przybliżeniami pewnego rzędu, lepsze przybliżenia otrzymamy ze wzorów [35]:

$$\begin{aligned} D_{f,n+1}(x,h) &= \frac{2^n D_{f,n}(x,h) - D_{f,n}(x,2h)}{2^n - 1}, \\ D_{b,n+1}(x,h) &= \frac{2^n D_{b,n}(x,h) - D_{b,n}(x,2h)}{2^n - 1}, \end{aligned} \tag{8.30}$$

$$D_{c,2(n+1)}(x,h) = \frac{2^n D_{c,2n}(x,h) - D_{c,2n}(x,2h)}{2^{2n} - 1}, \tag{8.31}$$

gdzie n to rząd przybliżenia, oraz

$$\begin{aligned} D_{f1}(x,h) &\equiv \frac{f(x+h) - f(x)}{h}, \\ D_{b1}(x,h) &\equiv \frac{f(x) - f(x-h)}{h}. \end{aligned} \tag{8.32}$$

h	$D_{f3}\big\vert_{x=\pi/3}$	$D_{f3} - \cos(\pi/3)$	$D_{f3} - D_{f2}$
10^{-1}	0.540604616151542	3.02310283401996e-04	-0.00128238312258866
10^{-2}	0.540302588847030	2.82978890542296e-07	-0.00001751610338996
10^{-3}	0.540302306148691	2.80551692988240e-10	-0.00000017960973453
10^{-4}	0.540302305870692	2.55184762210092e-12	-0.00000000179930149
10^{-5}	0.540302305888455	2.03154160161034e-11	-0.00000000000740152
10^{-6}	0.540302306006879	1.38739242316888e-10	0.00000000000000000
10^{-7}	0.540302309559593	3.69145292111739e-09	0.00000000222044605
10^{-8}	0.540302321401972	1.55338317764375e-08	0.00000000740148676
10^{-9}	0.540302617461445	3.11593305046820e-07	0.00000014802973658
10^{-10}	0.540302617461445	3.11593305046820e-07	-0.00000074014868312

Tabela 8.5: Przybliżenie pochodnej funkcji $\sin x$ w punkcie $x = \pi/3$ za pomocą różnic skończonych „w przód".

Ekstrapolację Richardsona można wykorzystać do praktycznego oszacowania optymalnego kroku h (patrz poprzedni paragraf).

Przykład Obliczymy pochodną numeryczną funkcji $f(x) = \sin x$ w punkcie $x = 1$ dla różnych wartości h za pomocą różnicy skończonej $D_{f3}(x, h)$. Wartość pochodnej w tym punkcie wynosi

$$f'(1) = \cos(1) = 0.540302305868140.$$

Wyniki obliczeń numerycznych dla pochodnej zestawione są w tabeli 8.5. Oprócz samej pochodnej, znajduje się tam również błąd przybliżenia oraz różnica między D_{f3} a D_{f2}. Najmniejszy błąd w wyznaczeniu pochodnej popełniony został dla $h = 10^{-4}$. Natomiast kryterium wyboru kroku w oparciu o ekstrapolację Richardsona daje 10^{-6}. Nie trafiliśmy wprawdzie w optymalny krok, ale oszacowana w ten sposób wartość daje rozsądne wyniki.

8.2 Interpolacja a różniczkowanie

Omawiane w poprzednich paragrafach wzory nadają się przede wszystkim do wyznaczania pochodnych numerycznych w sytuacji, kiedy dysponujemy algorytmem do obliczania wartości funkcji. Nie można ich raczej stosować dla funkcji zadanych tabelarycznie, zwłaszcza jeżeli danych jest mało i nie są od siebie równooddalone. Powodów jest kilka. Po pierwsze, chociaż możliwe jest znalezienie różnic skończonych przybliżających pochodną dla punktów nierówno od siebie odległych, dokładność takiego przybliżenia będzie dużo gorsza od poznanych do tej pory przybliżeń [67, 68]. Po drugie, jeśli danych jest mało, krok h jest z konieczności dość duży, co zwiększa niedokładność. Po trzecie, nawet jeśli danych jest dużo i są równoodległe, krok h może znacznie się różnić od wartości optymalnej. Gdybyśmy mimo to zdecydowali się przybliżać pochodną różnicami skończonymi, będziemy mieli kłopot z obliczeniem pochodnej dla punktów krańcowych. Stosuje się wówczas wzory niższych rzędów albo ekstrapoluje pochodną z wartości obliczonych wewnątrz przedziału, co znowu powiększa błąd przybliżenia.

Biorąc pod uwagę wszystkie wymienione powyżej problemy, często dużo lepszym rozwiązaniem jest znalezienie funkcji interpolującej dane, jeżeli jest ich mało, lub aproksymującej, jeżeli jest ich dużo (patrz rozdział 6), a następnie zróżniczkowanie tej funkcji[1].

8.3 Implementacje

Algorytmy do obliczania pochodnych są tak proste, że często implementujemy je samodzielnie. Mimo to warto wiedzieć, jakie gotowe narzędzia do liczenia pochodnych oferują nam popularne biblioteki.

C++

W bibliotece GSL [12] mamy do dyspozycji kilka funkcji obliczających pochodne. Wszystkie zadeklarowane są w pliku nagłówkowym `gsl_deriv.h`. Na przykład za pomocą

```
int gsl_deriv_central(const gsl_function * f, double x,
                      double h, double * result, double * abserr)
```

obliczymy pochodną numeryczną funkcji f w punkcie x z krokiem h przy wykorzystaniu różnic centralnych. Pochodna zwracana jest w zmiennej `result`, a szacowanie jej błędu bezwzględnego - w zmiennej `abserr`. Funkcja używa podanej wartości h do wyznaczenia optymalnego kroku w oparciu o zachowanie błędów obcięcia i zaokrągleń. Pochodna wyliczana jest ze wzoru pięciopunktowego, a błąd szacowany na podstawie różnicy między wzorem pięcio- i trzypunktowym.

Przykład Poniższy program wyliczy pochodną funkcji $f(x) = \sqrt(x)$ w punkcie $x = 2$.

```
#include <iostream>
#include <gsl/gsl_deriv.h>

double f(double x, void * params)
{
   return sqrt(x);
}

int main()
{
   gsl_function F;
   double result, abserr;

   F.function = &f;
   F.params = 0;

   gsl_deriv_central(&F,2.0,1e-8,&result,&abserr);

   std::cout<< "Wynik analityczny: " << 0.5/sqrt(2.0) << "\n";
   std::cout<< "Wynik numeryczny:  " << result << "\n";
   std::cout<< "Szacowanie błędu:  " << abserr << "\n";
}
```

[1] Pomijając dylemat doboru kroku, różniczkowanie funkcji interpolującej dane równoodległe równoważne jest stosowaniu wzorów na różnice skończone.

Po skompilowaniu i uruchomieniu programu, na ekranie zobaczymy następujący wynik

```
Wynik analityczny: 0.353553
Wynik numeryczny:  0.353553
Szacowanie błędu:  2.03214e-07
```

Fortran

W „Numerical Recipes" [15] znajdziemy funkcję `dfridr` do obliczania pochodnej metodą ekstrapolacyjną zaproponowaną przez Riddersa [69]. Do jej uruchomienia potrzebne będą moduły `nrtype` i `nrutil`. Odpowiednie pliki ściągniemy ze strony głównej „Numerical Recipes" [70].

Poniższy program ilustruje wykorzystanie funkcji `dfridr`:

```
program pochodna
  use nrtype
  use iface
  implicit none
  real(sp) :: x=1.0,h=1.0,err
  real(sp) :: df

  df=dfridr(funkcja,x,h,err)
  write(6,'(a21,f16.8)') "Pochodna numeryczna: ",df
  write(6,'(a21,f16.8)') "Szacowanie błędu:    ",err
  write(6,'(a21,f16.8)') "Pochodna 'dokładna': ",cos(x)
end program pochodna

function funkcja(x)
  use nrtype
  implicit none
  real(sp), intent(in) :: x
  real(sp) :: funkcja
  funkcja=sin(x)
end function
```

Jak widać, `dfridr` oczekuje czterech argumentów. `Func` to nazwa funkcji, której pochodnej w punkcie `x` szukamy. Algorytm poszukiwania pochodnej wystartuje z początkowym krokiem `h`. Nie musi być on bardzo mały. Lepiej jest wybrać `h` tak, aby reprezentowało charakterystyczną długość (wyrażoną w jednostkach `x`), na której funkcja zmienia się znacząco. Czwarty argument to zmienna, poprzez którą `dfridr` zwróci oszacowanie błędu wyliczenia pochodnej.

Słowa komentarza wymaga jeszcze moduł `iface`, który pojawił się na powyższym wydruku. Zawiera on interfejsy funkcji `dfridr` i `funkcja`. Przeniosłem je do innego pliku celem zwiększenia czytelności kodu.

Program skompilujemy poleceniem:

```
ifort nrtype.f90 nrutil.f90 iface.f90 pochodna.f90 dfridr.f90
```

Po jego uruchomieniu otrzymamy

```
Pochodna numeryczna:       0.54030228
Szacowanie błędu:          0.00000006
Pochodna 'dokładna':       0.54030228
```

Użytkownicy komercyjnej biblioteki NAG [71] mają do dyspozycji funkcję `D04AAF`, która wylicza pochodne (do 14 rzędu) funkcji jednej zmiennej.

Ciekawe rozwiązanie oferuje ADIFOR [72], narzędzie do automatycznego różniczkowania programów w Fortranie 77[2]. Jeżeli w jakimś pliku źródłowym zdefiniowane są funkcje , których pochodnych szukamy, ADIFOR automatycznie wygeneruje kod potrzebny do ich wyliczenia. Warto wspomnieć, że ADIFOR jest dostępny za darmo dla celów niekomercyjnych i edukacyjnych.

Python

Jeżeli chcemy skorzystać z gotowych funkcji do obliczania pochodnych w Pythonie, mamy do dyspozycji kilka możliwości.

W bibliotece ScientificPython K. Hinsena [17] zawarte są dwa moduły do różniczkowania funkcji. `Scientific.Functions.Derivatives` pozwala na wyliczenie pochodnej dowolnego rzędu funkcji wielu zmiennych. `Scientific.Functions.FirstDerivatives` ogranicza się tylko do pierwszych pochodnych, jest za to dużo szybszy. Oba działają bez zarzutu, chociaż trzeba się oswoić z mało intuicyjną składnią, np.:

```
>>> from Numeric import sin
>>> from Scientific.Functions.Derivatives import DerivVar
>>> print sin(DerivVar(2))
(0.90929742682568171, [-0.41614683654714241])
```

Pierwsza z uzyskanych liczb to wartość funkcji $\sin x$ w punkcie $x = 2$, druga - jej pochodna w tym punkcie. Jeżeli różniczkujemy funkcję wielu zmiennych, `DerivVar` musi być wywołane z drugim argumentem całkowitym określającym numer zmiennej (liczony od 0):

```
>>> from Numeric import sqrt
>>> x=DerivVar(7,0)
>>> y=DerivVar(42,1)
>>> z=DerivVar(2,2)
>>> print (sqrt(x*x+y*y+z*z))
(42.626282971894227, [0.16421793109700586,
0.9853075865820351, 0.046919408884858814])
```

W tym przykładzie otrzymaliśmy wartość funkcji $f(x) = \sqrt{x^2 + y^2 + z^2}$ oraz jej pochodne cząstkowe w punkcie $(7, 42, 2)$.

Pochodne wyższych rzędów wymagają trzeciego argumentu w `DerivVar`, np.:

```
>>> print sin(DerivVar(2,0,3))
(0.90929742682568171, [-0.41614683654714241],
[[-0.90929742682568171]], [[[0.41614683654714241]]])
```

Inna możliwość to projekt PyGSL [18] stanowiący interfejs do biblioteki GSL i funkcji w niej zawartych.

GNU Octave

GNU Octave daje do dyspozycji dwa narzędzia przydatne w różniczkowaniu numerycznym. Pierwszym jest funkcja `diff()`. Jeżeli `x` jest n-elementowym wektorem, wówczas `diff(x)` zwraca wektor różnic `x(2)-x(1)`, ..., `x(n)-x(n-1)`.

[2]Dostępny jest również ADIC, czyli wersja tego narzędzia dla ANSI C.

Dlatego funkcję tę możemy wykorzystać np. do przybliżenia pochodnej wzorem dwupunktowym w przód. Załóżmy, że plik `xy.dat` zawiera następujące dane:

```
#xy.dat
1       2
2       4
3       6
4       8
5       10
6       12
```

Wówczas do obliczenia różnic skończonych w przód wystarczy

```
octave:1> load xy.dat
octave:2> dydx = diff(xy(:,2))./diff(xy(1,:))
dydx =

  2
  2
  2
  2
  2
```

Drugim narzędziem jest funkcja `polyder`, za pomocą której możemy obliczyć pochodną wielomianu. Funkcja ta będzie szczególnie przydatna w sytuacji, gdy będziemy najpierw interpolować dane dyskretne, a następnie różniczkować funkcję interpolowaną.

Rozważmy problem znalezienia pochodnej funkcji $f(x) = \sin x$ w punkcie $x = \pi/4$, jeżeli zadana jest ona dyskretnym zbiorem punktów

$$\left\{(0, \sin 0,)\,,\ \left(\frac{\pi}{8}, \sin\frac{\pi}{8},\right),\ \left(\frac{\pi}{4}, \sin\frac{\pi}{4},\right),\ \left(\frac{3\pi}{8}, \sin\frac{3\pi}{8},\right),\ \left(\frac{\pi}{2}, \sin\frac{\pi}{2},\right)\right\}.$$

Ponieważ punktów jest mało, znajdziemy najpierw wielomian interpolujący przy pomocy funkcji `polyfit`, a następnie obliczymy jego pochodną:

```
octave:1> x0 = pi/4;
octave:2> dfx0 = cos(x0)
dfx0 = 0.70711
octave:3> x = [0:4]*pi/8;
octave:4> y = sin(x);
octave:5> p = polyfit(x,y,4)
p =

   0.02871  -0.20359   0.01995   0.99632   0.00000

octave:6> dpx = polyder(p)
dpx =

   0.114857  -0.610756   0.039903   0.996317

octave:7> dfx=polyval(dpx,x0)
dfx = 0.70656
octave:8> dfx-dfx0
ans =  -5.5034e-04
```

Rozdział 9

Zagadnienia własne

Rozdział 10

Równania różniczkowe zwyczajne

Dodatek A

GNU Octave

Dodatek B

SciPy

Moduł SciPy to zestaw bibliotek numerycznych przeznaczonych dla Pythona. Pod Linuksem zainstalujemy go albo z pakietów dołączanych do dystrybucji, albo ze źródeł. Użytkownicy Windowsa znajdą binarny instalator na stronie głównej modułu [16].

Możliwości Pythona rozszerzonego o SciPy są imponujące, porównywalne z możliwościami takich środowisk numerycznych jak Matlab (`http://www.mathworks.com/`) czy GNU Octave [19]. Jednak w odróżnieniu od tych środowisk nie jesteśmy ograniczeni do pracy z interpreterem czy ze skryptami o czysto obliczeniowym przeznaczeniu. Dzięki uniwersalności Pythona możliwe jest tworzenie różnorodnych aplikacji korzystających z algorytmów numerycznych zaimplementowanych w SciPy.

Moduł podzielony jest na kilka podmodułów. Ich listę i krótki opis znajdziemy w tabeli B.1. Aby skorzystać z jakiejś funkcji znajdującej się w dowolnym z tych podmodułów, musimy najpierw zaimportować Scipy,

```
>>> import scipy
```

Jednym ze znaków rozpoznawczych Pythona jest doskonała dokumentacja dostępna z poziomu interpretera. Nie inaczej jest pod tym względem z modułem SciPy - dokumentację interesującej nas funkcji wywołamy poleceniem `scipy.info`,

```
 >>> scipy.info(scipy.linalg.pinv)
 pinv(a, cond=None)

pinv(a, cond=None) -> a_pinv

Compute generalized inverse of A using
least-squares solver.
```

W każdej chwili możemy również podglądnąć implementację używanej funkcji,

```
>>> scipy.source(scipy.linalg.pinv)
In file:
/usr/lib/python2.4/site-packages/scipy/linalg/basic.py

def pinv(a, cond=None):
    """ pinv(a, cond=None) -> a_pinv
```

Moduł	Opis
cluster	algorytmy klastrowania
cow	wsparcie dla programowania równoległego
fftpack	szybka transformata Fouriera
fftw	j.w. (wymaga biblioteki FFTW)
ga	algorytmy genetyczne
gplt	wizualizacja danych (wymaga Gnuplota)
integrate	całkowanie numeryczne
interpolate	interpolacja
io	operacje wejścia/wyjścia
linalg	algebra liniowa
optimize	optymalizacja, poszukiwanie miejsc zerowych
plt	wizualizacja danych (wymaga wxPythona)
signal	analiza sygnałów
special	funkcje specjalne
stats	funkcje i rozkłady statystyczne
xplt	wizualizacja danych (za pomocą gista)

Tabela B.1: Lista podmodułów w pakiecie SciPy.

```
    Compute generalized inverse of A using
    least-squares solver.
    """
    a = asarray_chkfinite(a)
    t = a.typecode()
    b = scipy_base.identity(a.shape[0],t)
    return lstsq(a, b, cond=cond)[0]
```

Dzięki temu nie tylko łatwiej jest nam zrozumieć sposób jej działania. To również bardzo dobra metoda uczenia się zaawansowanego programowania w Pythonie.

Dodatek C

Biblioteka GSL

Dodatek D

Gnuplot

W metodach numerycznych obok wygenerowania wyniku równie ważne jest jego odpowiednie przedstawienie. Ludzki mózg tylko w ograniczonym zakresie jest w stanie przyswajać i przetwarzać informacje zakodowane w długich ciągach liczb, natomiast znakomicie radzi sobie z obrazami. Dlatego, o ile to tylko możliwe, rozsądniej jest prezentować dane w formie graficznej niż tabelarycznej. Dzięki odpowiedniej wizualizacji łatwiej jest szukać związków między danymi, na podstawie których można później wyciągać wnioski na temat badanych zjawisk.

Istnieje wiele programów do wizualizacji danych. Jednym z popularnych narzędzi tego typu jest Gnuplot [23]. Pozwala tworzyć wykresy dwu- i trójwymiarowe, a przy tym dostępny jest na większość platform. Program działa w trybie tekstowym[1]. Można z nim pracować interaktywnie, w wierszu poleceń, lub w trybie wsadowym przy pomocy odpowiednich skryptów.

Oprócz przedstawienia wykresów na ekranie, Gnuplot oferuje możliwość zapisania ich do plików w różnych formatach graficznych, m.in. EPS, PNG i JPEG.

Głównym źródłem informacji na temat Gnuplota jest dokumentacja na stronie głównej programu [23]. Warto odwiedzić również stronę „Gnuplot tips" [73], na której można znaleźć kilka ciekawych przykładów jego użycia.

D.1 Pierwsze kroki

D.1.1 Uruchamianie i zatrzymywanie programu, sesje

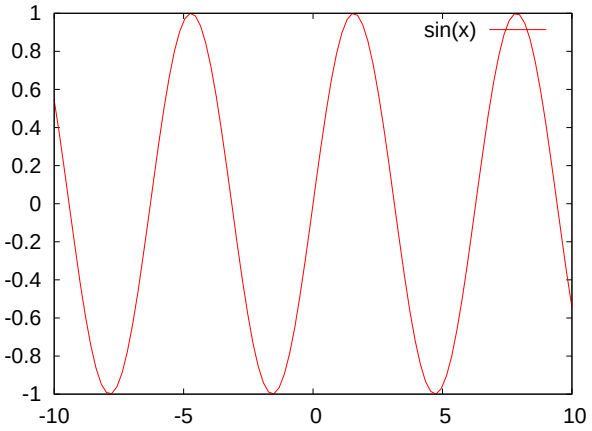
Po zainstalowaniu programu uruchamiamy go w konsoli poleceniem `gnuplot`. O wystartowaniu interpretera poinformowani zostaniemy zmienionym znakiem zachęty:

```
gnuplot>
```

Jego działanie kończymy poleceniami `exit` lub `quit`. Po wpisaniu `help` uzyskamy dostęp do plików pomocy dostarczonych z programem.

Wszystkie polecenia, jakie wydamy w interpreterze od czasu jego uruchomienia, tworzą sesję. Możemy je zapisać do pliku o dowolnej nazwie za pomocą `save`:

[1] Istnieje wprawdzie kilka nakładek graficznych na Gnuplota, jednak nie będziemy się nimi tutaj zajmować.



Rysunek D.1: Pierwszy wykres w Gnuplocie.

```
gnuplot> save "mojasesja.plt"
```

Po następnym uruchomieniu programu odtworzymy tak zapisaną sesję w następujący sposób:

```
gnuplot> load "mojasesja.plt"
```

D.1.2 Proste wykresy dwuwymiarowe

Aby sporządzić wykres funkcji $\sin x$, wpisujemy po prostu[2]

```
gnuplot> plot sin(x)
```

Wynik przedstawiony jest na rys. D.1.

Jeśli interesuje nas jakiś konkretny przedział wartości zmiennej niezależnej, możemy go podać jako argument funkcji plot,

```
gnuplot> plot [0:5] sin(x)
```

lub zmienić poleceniem `set`,

```
gnuplot> set xrange [0:5]
gnuplot> replot
```

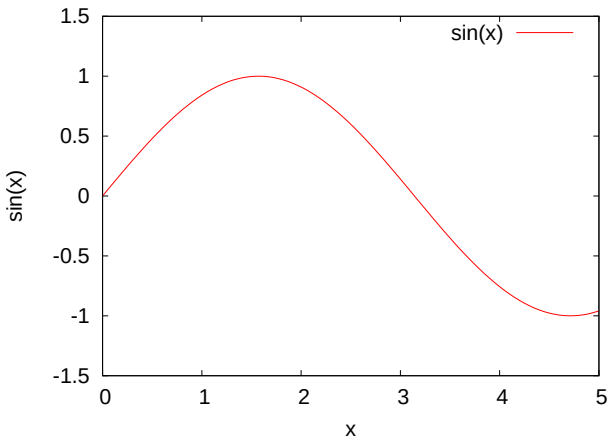
Wykonanie `replot` bez żadnych argumentów powtarza ostatnie wywołanie polecenia `plot`.

Aby dodać opisy osi na wykresie, również użyjemy `set` z odpowiednimi słowami kluczowymi:

```
gnuplot> set xlabel "x"
gnuplot> set ylabel "y"
gnuplot> set xrange [0:5]
gnuplot> set yrange [-1.5:1.5]
gnuplot> plot sin(x)
```

Uzyskany w ten sposób wykres przedstawiony jest na rysunku D.2.

[2]Oprócz $\sin x$ użytkownik Gnuplota ma do dyspozycji wiele popularnych funkcji matematycznych.



Rysunek D.2: Drugi wykres w Gnuplocie.

D.1.3 Zapisywanie wykresów do pliku

Aby zapisać wykres do pliku, musimy najpierw zmienić typ terminala, a następnie określić nazwę pliku:

```
gnuplot> set terminal postscript eps
gnuplot> set output "sin.eps"
gnuplot> plot sin(x)
```

W tym wypadku zamiast na pokazany na ekranie, wykres funkcji $\sin x$ zostanie zapisany do pliku `sin.eps`. Listę dostępnych terminali wyświetlimy poleceniem `set terminal`. Jest ona naprawdę imponująca, choć w nowszych wersjach Gnuplota (zwłaszcza tych rozprowadzanych razem z niektórymi dystrybucjami Linuksa) dziwić może trochę brak możliwości zapisu plików w formacie PDF. Jest to związane z mało przyjazną dla Wolnego Oprogramowania licencją biblioteki PDFlib, którą Gnuplot wykorzystuje przy konwersji wykresów. Jeżeli więc zależy nam na wykresach w tym formacie, mamy dwa wyjścia. Albo korzystamy z terminala `postscript`, tworzymy pliki w formacie EPS i przekształcamy je do PDF za pomocą programu `epstopdf`. Albo instalujemy brakującą bibliotekę własnoręcznie (`http://www.pdflib.org`) i kompilujemy źródła Gnuplota, odpowiednio je z nią linkując.

D.1.4 Wizualizacja danych dyskretnych

Do tej pory mieliśmy do czynienia jedynie z funkcjami zadanymi za pomocą wzorów matematycznych. Gnuplot pozwala również na wizualizację danych dyskretnych wprowadzanych albo ze standardowego wejścia, albo, co dużo wygodniejsze, wczytywanych z pliku.

Niech naszym zadaniem będzie przedstawienie na wykresie jakości dwóch różnych przybliżeń Pade [15, 74] funkcji e^{-x} dla różnych wartości x. Poniższy program pozwoli wygenerować nam potrzebne dane numeryczne:

```
//pade.cpp
//aproksymacja Pade funkcji exp(-x)
#include <iostream>
```

```
#include <cmath>

int main()
{
  double y,z1,z2;
  double d = 0.1;
  double x = 0.0;
  for(int i=0;i<50;i++)
    {
      x += d;
      y  = exp(-x);
      z1 = (6.0 - 2.0*x)/(6.0 + 4.0*x + x*x);
      z2 = (6.0 - 4.0*x + x*x)/(6.0 + 2.0*x);
      std::cout << x << "\t" << y << "\t" << z1 << "\t" << z2 << "\n";
     }
}
```

Wynikowy plik będzie miał postać:

```
#pade.dat
#aproksymacja Pade funckji exp(-x)
#x      exp(-x)          z1               z2
0.1     0.904837         0.904836         0.904839
0.2     0.818731         0.818713         0.81875
0.3     0.740818         0.740741         0.740909
0.4     0.67032          0.670103         0.670588
0.5     0.606531         0.606061         0.607143
0.6     0.548812         0.547945         0.55
0.7     0.496585         0.495156         0.498649
0.8     0.449329         0.447154         0.452632
0.9     0.40657          0.403458         0.411538
1       0.367879         0.363636          0.375
.
.
.
```

Gnuplot ignoruje każdą linię, która zaczyna się od znaku #. Dzięki temu możemy wstawiać komentarze do plików z danymi. To bardzo użyteczne, zwłaszcza gdy mamy wiele takich plików.

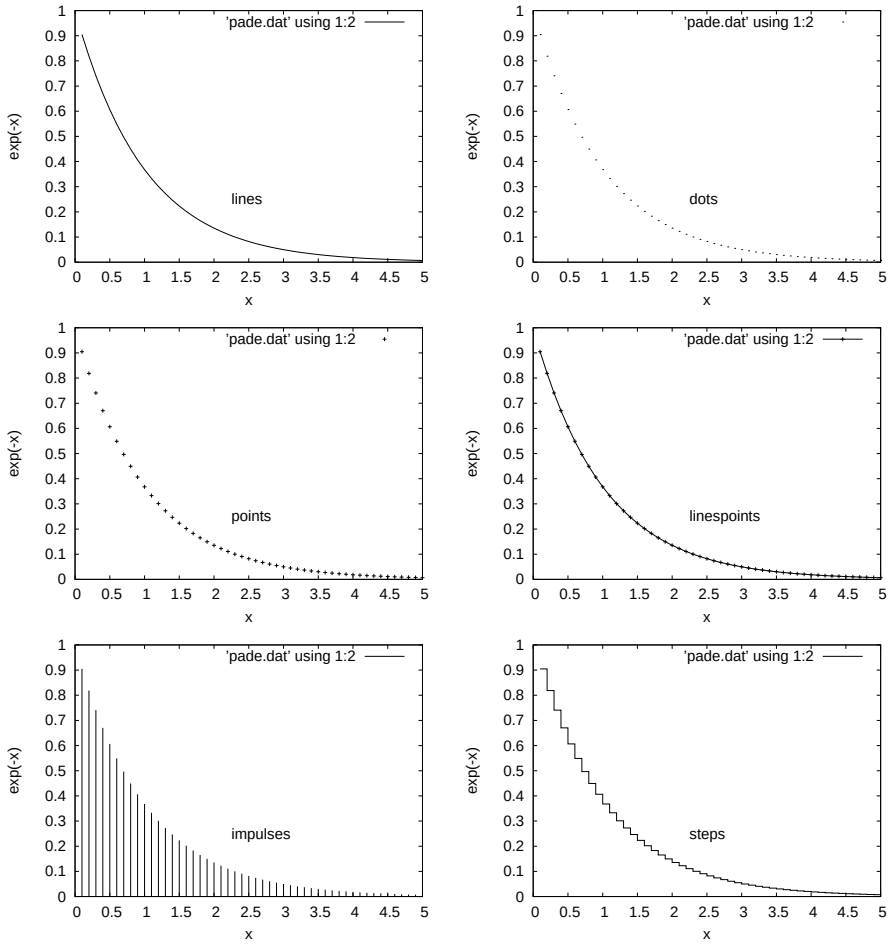
Wykres na podstawie dwóch pierwszych kolumn z danymi sporządzimy w następujący sposób

```
gnuplot> plot "pade.dat" using 1:2 with lines
```

Wyboru kolumn dokonaliśmy poleceniem `using`. Natomiast `with` określa styl wykresu. Oprócz `lines`, który łączy punkty linią, mamy do dyspozycji kilka innych, m.in. `dots`, `points`, `linespoints`, `impulses` i `steps`. Różnice między nimi zaprezentowane są na rys. D.3

Aby na jednym wykresie umieścić kilka krzywych, dla każdej z nich musimy powtórzyć specyfikację danych. Odpowiednie polecenie może okazać się bardzo długie. Wówczas dla poprawienia czytelności możemy użyć znaku kontynuacji linii \:

```
gnuplot> plot "pade.dat" using 1:2 with lines lt 1, \
              "pade.dat" using 1:3 with lines lt 8, \
              "pade.dat" using 1:4 with lines lt 6
```



Rysunek D.3: Popularne style wykresów w Gnuplocie.

Odpowiedni wykres przedstawiony jest na rysunku D.4.

D.1.5 Legenda, tytuł, etykiety

Gnuplot automatycznie tworzy legendę wykresu. Wpisy do niej możemy zmodyfikować za pomocą słowa kluczowego `title`:

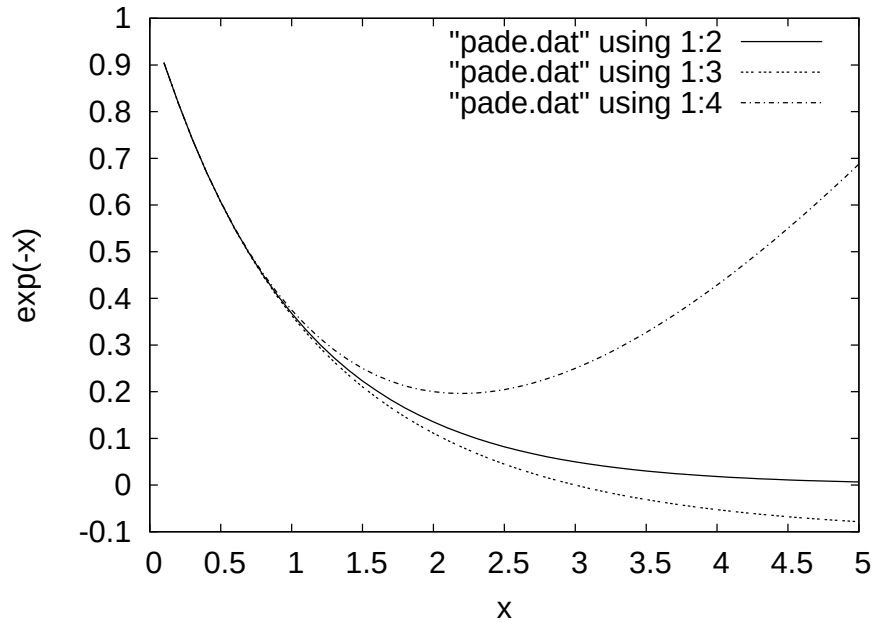
```
gnuplot> plot "pade.dat" using 1:2 title "analityczne" with \
      lines, \
      "pade.dat" using 1:3 title "L=1, M=2" with lines, \
      "pade.dat" using 1:4 title "L=2, M=1" with lines
```

Domyślnie legenda pojawi się w prawym górnym rogu wykresu. Położenie oczywiście również można zmienić:

```
gnuplot> set key left bottom
```

Do ustawień domyślnych powrócimy poprzez

```
gnuplot> set key default
```



Rysunek D.4: Kilka funkcji na jednym wykresie.

Wyłączamy legendę poleceniem:

```
gnuplot> unset key
```

Jeżeli uważamy, że legenda nie wystarczy do pełnego opisu wykresu, mamy jeszcze możliwość nadania mu tytułu

```
gnuplot> set title "Aproksymacja Pade"
```

lub wstawienia dodatkowych opisów, tzw. etykiet. Na przykład po wpisaniu

```
gnuplot> set label "a" at graph 0.5,0.5 font "Symbol"
```

na środku wykresu pojawi się grecka litera α. Pamiętajmy przy tym, że zmiany dokonane za pomocą `set` już po wykonaniu polecenia `plot` staną się widoczne dopiero po powtórzeniu tej komendy lub wpisaniu `replot`.

D.1.6 Proste wykresy trójwymiarowe

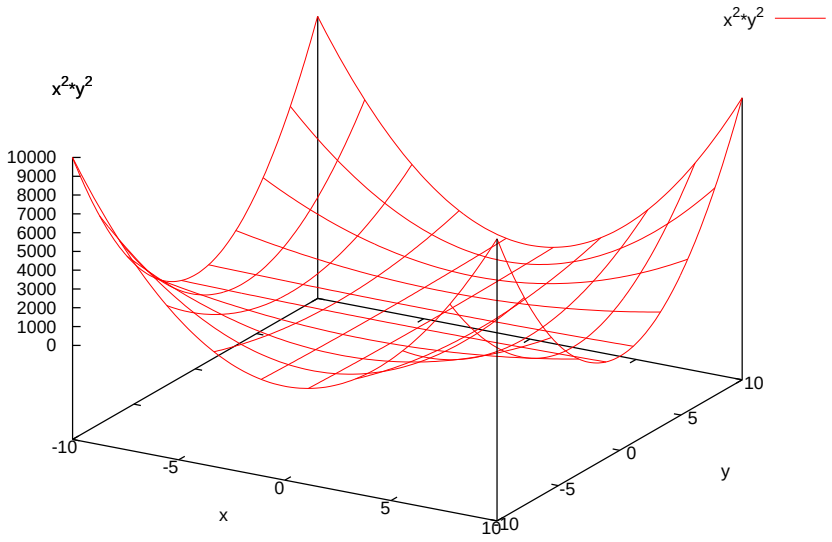
Do rysowania wykresów trójwymiarowych służy polecenie `splot`,

```
gnuplot> splot (x**2)*(y**2)
```

Jak widać na rys. D.5, zero na osi Z domyślnie znajduje się nad płaszczyzną XY. Zmienimy to za pomocą

```
gnuplot> set ticslevel 0
```

Rozmiar siatki, dla której węzłów wyliczane są wartości wykreślanej funkcji, przechowywany jest w zmiennej `isosample`. Domyślnie rozmiar ten wynosi 10, czasami jednak zachodzi potrzeba zwiększenia go:



Rysunek D.5: Wykres trójwymiarowy w Gnuplocie.

```
gnuplot> set isosample 40
```

Podobnie, jak w przypadku `plot`, możliwe są wykresy na podstawie danych wczytanych z pliku:

```
gnuplot> splot "potencjal.dat" using 2:3:4 title "potencjał"\
         with lines
```

D.1.7 Zmienne i funkcje użytkownika

Gnuplota można używać jako rozbudowany kalkulator:

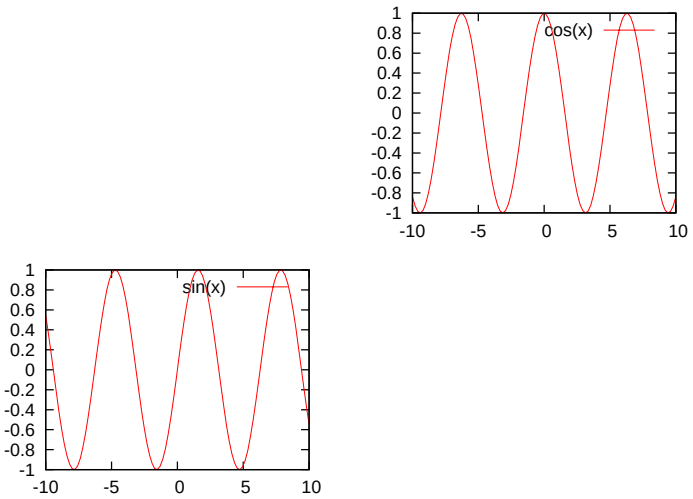
```
gnuplot> a = 1+2*sqrt(3)
gnuplot> print log(a)
1.4960679806764
gnuplot> plot a*sin(x)
```

Przy tym, obliczenia na liczbach rzeczywistych prowadzone są w podwójnej precyzji. Warto również wiedzieć, że we wbudowanej zmiennej `pi` przechowywana jest liczba π:

```
gnuplot> print pi
3.14159265358979
gnuplot> plot [-pi:pi] sin(x)
```

Bardzo użyteczną własnością jest możliwość definiowania własnych funkcji:

```
gnuplot> f(x) = sin(x)*cos(x)
gnuplot> print f(0)
0.0
gnuplot> plot f(x)
```



Rysunek D.6: Gnuplot w trybie wielowykresowym.

Będziemy z niej korzystać przy aproksymacji danych eksperymentalnych (patrz paragraf D.2.3).

D.2 Zaawansowane możliwości Gnuplota

D.2.1 Tryb wielowykresowy

Jeżeli chcemy przedstawić na ekranie (lub w pliku) kilka wykresów obok siebie lub jeden w drugim, musimy przełączyć Gnuplota w tryb wielowykresowy,

```
gnuplot> set multiplot
```

a następnie dla każdego wykresu zdefiniować jego położenie (`origin`) i rozmiar (`size`):

```
gnuplot> set multiplot
gnuplot> set size 0.4,0.4
gnuplot> set origin 0.1,0.1
gnuplot> plot sin(x)
gnuplot> set size 0.4,0.4
gnuplot> set origin 0.5,0.5
gnuplot> plot cos(x)
```

Wynik przedstawiony jest na rys. D.6. Z trybu wychodzimy poleceniem

```
gnuplot> unset multiplot
```

D.2.2 Manipulowanie danymi

Znane już słowo kluczowe `using` nie tylko definiuje określa kolumny w pliku, z których Gnuplot ma wczytywać dane, ale również umożliwia różne manipulacje na nich. Rozważmy prosty przykład. Niech plik z danymi ma postać:

```
#dane.dat
1
2
3
4
5
6
7
8
9
10
```

Aby narysować wykres funkcji $y = x^2$ dla wartości x zapisanych w tym pliku, wystarczy polecenie

```
gnuplot> plot "dane.dat" using 1:($1**2)
```

gdzie `**` to znany z Fortrana operator potęgowania. Jak widać w powyższym przykładzie, do danych wczytywanych z i-tej kolumny Gnuplot pozwala się dostać za pomocą $\$i$.

Patrząc na rys. D.4, już na pierwszy rzut oka widzimy, że rozważane przybliżenia Pade funkcji $y = e^{-x}$. dają rozsądne wyniki tylko w pewnym zakresie wartości zmiennej x. Na podstawie danych, którymi już dysponujemy, możemy sporządzić wykres błędu względnego (patrz paragraf 2.3.2) tych przybliżeń:

```
gnuplot> plot "pade.dat" using 1:(abs($3-$2)/$3)  with lines, \
     "pade.dat" using 1:(abs($4-$2)/$4)  with lines
```

D.2.3 Aproksymacja średniokwadratowa

Częstym zadaniem (nie tylko) w fizyce jest znalezienie funkcji matematycznej najlepiej pasującej do danych pochodzących z pomiaru lub obliczeń. Załóżmy, że podobną operację mamy przeprowadzić dla danych, których wykres przedstawiony jest na rys. D.7. Spróbujemy dopasować do nich funkcję Gaussa

```
gnuplot> f(x)=1/(s*sqrt(2*pi))*exp(-((x-m)**2/(2*s**2)))
```

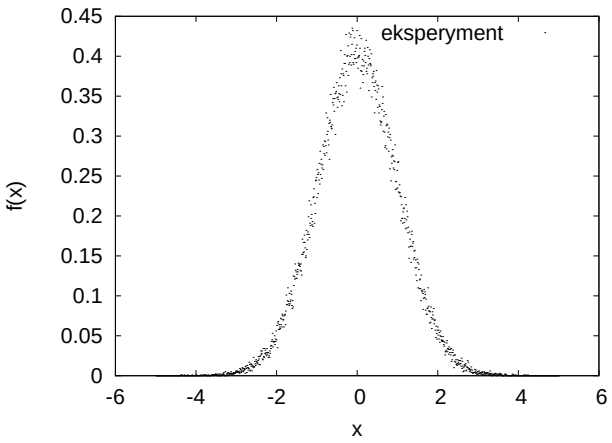
gdzie s i m to szukane parametry. Ich wartości obliczymy w następujący sposób:

```
gnuplot> fit f(x) 'normal.dat' via s,m
```

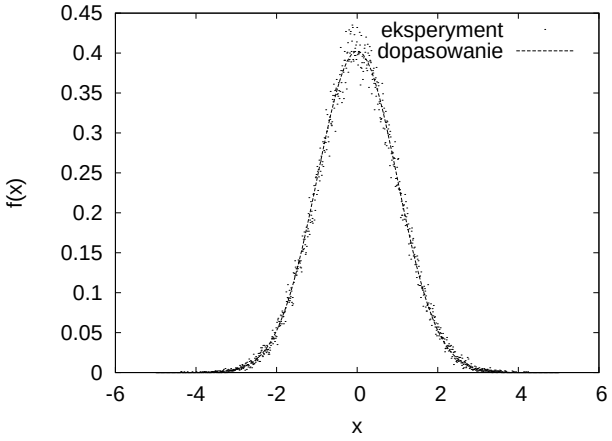
Wynik pojawi się na ekranie,

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================

s               = 0.991271         +/- 0.002066     (0.2084%)
m               = 0.00130067       +/- 0.002529     (194.4%)
```

i jednocześnie zostanie zapisany do pliku `fit.log`. Nie pozostaje nam już nic innego, jak sprawdzić dopasowanie na wykresie (patrz rys. D.8).



Rysunek D.7: Dane, które chcemy przybliżyć funkcją matematyczną.



Rysunek D.8: Dane eksperymentalne i dopasowana krzywa.

D.2.4 Symulacje w czasie rzeczywistym

W Gnuplocie zdefiniowanych jest kilka nazw specjalnych. Jedna z nich, `'-'`, oznacza standardowe wejście i pozwala na podanie danych razem z poleceniem `plot`, np.:

```
gnuplot> plot '-'
1 1
2 4
3 9
4 16
e
```

Przy tym, znak `e` sygnalizuje koniec bloku danych. Dzięki tej własności możemy użyć Gnuplota do tworzenia wizualizacji w czasie rzeczywistym.

Dla ustalenia uwagi rozważmy błądzenie losowe cząstki na siatce kwadratowej z periodycznymi warunkami brzegowymi. Silnikiem naszej symulacji będzie

prosty program w Fortranie,

```
!symulacja.f90
!prosta symulacja z wykorzystaniem
!mozliwosci Gnuplota
!JS, 22.08.2006

program symulacja
  implicit none
  real:: pos(2),xx,yy
  logical:: ok

  inquire(file='fifo',exist=ok)
  if(.not.ok) stop 'Nazwany potok nie istnieje!'

  pos = 0.d0

  do
    write(6,*) "set nokey"
    write(6,*) "set xr [-1:1]"
    write(6,*) "set yr [-1:1]"
    write(6,*) "plot '-' w p"
    write(6,*) pos

    call random_number(xx)
    xx = xx - 0.5d0
    call random_number(yy)
    yy = yy - 0.5d0
    pos(1) = pos(1)+0.1d0*xx
    pos(2) = pos(2)+0.1d0*yy
    where(pos(:)<-1.d0) pos(:)  =  1.d0
    where(pos(:)> 1.d0) pos(:)  = -1.d0
    write(6,*) "e"
  end do
end program symulacja
```

który wypisuje na standardowym wyjściu (w pętli nieskończonej) kilka poleceń Gnuplota oraz wyliczone położenie cząstki. Program możemy skompilować kompilatorem Intela

```
ifort symulacja.f90
```

lub gfortran,

```
gfortran symulacja.f90
```

Oprócz programu generującego dane potrzebować jeszcze będziemy nazwanego potoku (ang. *named pipe*) [75]. Potoki wykorzystywane są w systemach operacyjnych do komunikacji między różnymi programami. Potok tworzymy w katalogu, w którym znajdują się binaria naszego programu:

```
mkfifo FIFOfile
```

Uruchamiamy symulację, przekierowując standardowe wyjście do utworzonego potoku:

```
./a.out > FIFOfile &
```

Następnie startujemy Gnuplota w trybie wczytywania z potoku:

```
gnuplot < FIFOfile
```

i wizualizacja jest gotowa.

D.3 Gnuplot w praktyce

Omówione do tej pory możliwości Gnuplota są imponujące, ale początkujący użytkownik, zwłaszcza ten wychowany na systemach operacyjnych z rodziny MS Windows, może zacząć zastanawiać się, po co uczyć się tych wszystkich poleceń, skoro w wielu innych programach z „porządnym" (czytaj: graficznym) interfejsem użytkownika można ten sam wykres sporządzić na ogół bez żadnego przygotowania w mniej lub bardziej intuicyjny sposób. I rzeczywiście, są to wątpliwości uzasadnione, zwłaszcza w sytuacji, kiedy sporządzenie wykresu jakiejś funkcji jest zadaniem jednorazowym bądź bardzo sporadycznym.

Zalety Gnuplota (ogólnie programów pracujących w trybie tekstowym) ujawniają się najczęściej wówczas, gdy mamy do wykonania dużo podobnych wykresów i chcielibyśmy jakoś zautomatyzować proces obróbki danych i dostrajania wyglądu wykresu. W paragrafie tym postaram się pokazać, jak takie rzeczy robić w praktyce.

D.3.1 Skrypty

Skrypt w Gnuplocie to plik tekstowy, do którego wpisujemy polecenia programu tak, jakbyśmy wykonywali je w interpreterze. Poznana w paragrafie D.1.1 komenda `save` nie robi nic innego, jak właśnie przekształca aktualną sesję interaktywną w skrypt.

Dla ustalenia uwagi rozważmy następujący przykład[3]:

```
#sym.gpl
set terminal postscript eps color enhanced 'Helvetica' 24
set encoding iso_8859_2
set output "sym1.eps"
set xr [0:500]
set xlabel "{/Symbol t}"
set ylabel "Gęstość populacji"
plot "sym1.dat" using 1:4 title 'drap' with lines, \
     "" using 1:2 title 'of1' with lines, \
     "" using 1:3 title 'of2' with lines
```

W skrypcie tym pojawiło się kilka nowych elementów, ale ich znaczenie powinno być jasne. Plik `sym1.dat` to wynik pewnej symulacji:

```
#sym1.dat
#czas      x         y         z
0         0.1      0.1      0.01
0.05      0.103164          0.103216          0.0104162
0.1       0.10635           0.10646           0.0108668
0.15      0.109549          0.109723          0.0113549
0.2       0.112751          0.112995          0.0118836
0.25      0.115944          0.116264          0.0124567
```

[3]Skrypty Gnuplota mogą mieć dowolną nazwę. Rozszerzenie `gpl` nie jest wymagane.

```
0.3     0.119116        0.11952         0.013078
0.35    0.122253        0.122747        0.0137519
0.4     0.125338        0.125931        0.014483
0.45    0.128356        0.129055        0.015276
0.5     0.131287        0.132102        0.0161365
.
.
.
```

Skrypt uruchamiamy poleceniem

```
gnuplot [-persist] sym.gpl
```

Argument `-persist` przyda się, jeżeli zamiast do pliku, chcielibyśmy przedstawić wykres na ekranie. Wówczas okno z wykresem nie zniknie po zakończeniu działania skryptu.

Użytkownicy Basha mogą skrócić wywołanie skryptu, dopisując do niego (w pierwszej linii)

```
#!/usr/bin/gnuplot -persist
```

i nadając mu atrybut wykonywalności

```
chmod u+x sym.gpl
```

Wówczas wystarczy wpisać w konsoli

```
./sym.gpl
```

D.3.2 Automatyzacja zadań

Często zdarza się, że w wyniku symulacji otrzymujemy nie jeden plik z danymi, ale wiele podobnych (np. `sym1.dat`, `sym2.dat` itd.). Dzięki uniksowemu narzędziu `sed` i skryptowi, który sporządziliśmy w poprzednim paragrafie, niewielkim nakładem sił możemy stworzyć wykres dla danych z dowolnego pliku, np.:

```
sed "s/sym1/sym2/g" sym.gpl | gnuplot
```

lub od razu dla wszystkich:

```
$ for i in sym*.dat ; do
 > sed "s/sym1/‘echo ${i%.dat}‘/g" sym.gpl | gnuplot
 > done
```

Innym sposobem będzie wykorzystanie właściwości powłoki. Zmodyfikujmy nieco skrypt `sym.gpl`:

```
#!/bin/sh
#plot.sh
gnuplot <<KOD
set terminal postscript eps color enhanced 'Helvetica' 24
set encoding iso_8859_2
set output "$1.eps"
set xr [0:500]
set xlabel "{/Symbol t}"
set ylabel "Gęstość populacji"
plot "$1.dat" using 1:4 title 'drap' with lines, \
     "" using 1:2 title 'of1' with lines, \
     "" using 1:3 title 'of2' with lines
KOD
```

Wykorzystaliśmy tu tzw. **here documents**, które pozwalają na przekierowywanie bloków kodu na standardowe wejścia różnych programów, w tym wypadku Gnuplota. Po nadaniu skryptowi atrybutu wykonalności wykres dla danych z pliku `sym3.dat` sporządzimy wywołaniem

```
./plot.sh sym3
```

Nazwa, która podamy jako argument wywołania, zostanie wstawiona w miejsce `$1` w skrypcie. Niewiele trudniejsze będzie sporządzenie wszystkich wykresów na raz:

```
$ for i in sym*.dat ; do
 > ./plot.sh ${i%.dat}
 > done
```

Tym sposobem nauczyliśmy się automatycznego generowania wykresów w Gnuplocie. Zalet takiego rozwiązania jest kilka, ale najważniejsza to z pewnością wydajność. Po sporządzeniu pierwszego skryptu dla wybranego pliku z danymi wykonanie pozostałych wykresów to kwestia sekund potrzebnych na wprowadzenie odpowiednich poleceń w powłoce, niezależnie od liczby plików. Przy tym gwarantowane jest, że wszystkie wykresy będą miały jednolitą szatę graficzną: rozmiar i krój czcionek, opisy, typy i kolory linii. Równie szybko dokonuje się zmian w wyglądzie wykresów - wystarczy zmodyfikować odpowiedni wpis w skrypcie i ponownie go wywołać dla wszystkich plików z danymi.

Bibliografia

[1] Eric S. Raymond. *UNIX. Sztuka programowania*. Helion, 2004.

[2] TOP500 Supercomputer Sites: `http://www.top500.org/`.

[3] The Linux Clustering Information Center: `http://lcic.org/`.

[4] Scientific Computing: C++ versus Fortran: `http://osl.iu.edu/~tveldhui/papers/DrDobbs2/drdobbs2.html`.

[5] GCC, the GNU Compiler Collection: `http://gcc.gnu.org/`.

[6] KDevelop - an Integrated Development Environment: `http://www.kdevelop.org/`.

[7] Anjuta Integrated Development Environment: `http://anjuta.sourceforge.net/`.

[8] Środowisko programistyczne Dev-C++: `http://www.bloodshed.net/devcpp.html`.

[9] Strona główna OpenWatcom: `http://www.openwatcom.org/`.

[10] L. Prechelt, "An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program": `http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=ira/2000/5`.

[11] The Python Programming Language: `http://www.python.org/`.

[12] GNU Scientific Library: `http://www.gnu.org/software/gsl/`.

[13] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[14] Fortran Library - A free technical programming resource: `http://www.fortranlib.com/`.

[15] W. H. Press, S. A. Teutolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes*. Cambridge University Press, Cambridge, 1992.

[16] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[17] ScientificPython - a collection of Python modules for scientific computing: http://starship.python.net/~hinsen/ScientificPython/.

[18] PyGSL - Python interface for GNU Scientific Library: http://pygsl.sf.net/.

[19] Strona główna GNU Octave: http://www.gnu.org/software/octave/.

[20] Scilab - A Free Scientific Software Package: http://www.scilab.org/.

[21] Strona główna programu Rlab: http://rlab.sourceforge.net/.

[22] Matlab/Octave Compatibility: http://wiki.octave.org/wiki.pl?MatlabOctaveCompatibility.

[23] Strona główna Gnuplota: http://www.gnuplot.info/.

[24] Grace - a WYSIWYG 2D plotting tool : http://plasma-gate.weizmann.ac.il/Grace/.

[25] SciGraphica - Scientific Graphics and Data Manipulation : http://scigraphica.sourceforge.net/.

[26] Maxima - A GPL CAS based on DOE-MACSYMA: http://maxima.sourceforge.net/.

[27] Axiom Computer Algebra System: http://www.axiom-developer.org/.

[28] Yacas - yet another computer algebra system: http://yacas.sourceforge.net/.

[29] C++ Big Integer Library: http://www.kepreon.com/~matt/bigint/index.html.

[30] GNU MP Bignum Library: http://www.kepreon.com/~matt/bigint/index.html.

[31] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *Computing Surveys*, March 1991.

[32] D. W. Matula and P. Kornerup. Finite precision rational arithmetic: Slash number systems. *IEEE Trans. Comput.*, C(34(1)):1, 1985.

[33] E. E. Swartzlander and A. G. Alexopoulos. The sign/logarithm number system. *IEEE Trans. Comput.*, C(24(12)):1238, 1975.

[34] B. M. Bush. The Perils of Floating Point, http://www.lahey.com/float.htm.

[35] W. Y. Yang, W. Cao, T.-S. Chung, and J. Morris. *Applied numerical methods using Matlab*. Wiley-Interscience, 2005.

[36] Å. Björck and G. Dahlquist. *Metody numeryczne*. Państwowe Wydawnictwo Naukowe, Warszawa, 1983.

[37] K. A. Ross and R. B. Wright. *Matematyka dyskretna*. Wydawnictwo Naukowe PWN, Warszawa, 2000.

[38] P. Wróblewski. *Algorytmy, struktury danych i techniki programowania*. Helion, Gliwice, 2001.

[39] `http://en.wikipedia.org/wiki/Software_optimization`.

[40] Dokumentacja profilera `gprof`: `http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html`.

[41] Strona główna programu `KProf`: `http://kprof.sourceforge.net/`.

[42] Dokumentacja modułów do profilowania programów w Pythonie: `http://docs.python.org/lib/profile.html`.

[43] Dokumentacja GNU Octave: `http://www.gnu.org/software/octave/doc/interpreter/index.html`.

[44] Strona główna pakietu `octave-forge`: `http://octave.sf.net/`.

[45] A. Mostowski and M. Stark. *Elementy algebry wyższej*. PWN, Warszawa, 1965.

[46] A. I. Kostrikin and J. I. Manin. *Algebra liniowa i geometria*. Wydawnictwo Naukowe PWN, Warszawa, 1993.

[47] J. Stoer and R. Bulirsch. *Wstęp do analizy numerycznej*. Państwowe Wydawnictwo Naukowe, Warszawa, 1987.

[48] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore and London, 1996.

[49] J. H. Wilkinson. *Błędy zaokrągleń w procesach algebraicznych*. PWN, Warszawa, 1967.

[50] Z. Fortuna, B. Macukow, and J. Wąsowski. *Metody numeryczne*. Wydawnictwa Naukowo–Techniczne, Warszawa, 1993.

[51] Z Wikipedii, wolnej encyklopedii: `http://en.wikipedia.org/wiki/Gaussian_elimination`.

[52] G. H. Golub and W. Kahn. *SIAM J. Numerical Analysis*, 2:205, 1965.

[53] P. A. Businger and G. H. Golub. *Comm. ACM*, 12:564, 1969.

[54] C. L. Lawson and R. J. Hanson. *Solving Least Squares Problems*. Prentice-Hall, Englewood Cliffs NJ, 1974.

[55] S. Brandt. *Analiza danych*. Wydawnictwo Naukowe PWN, Warszawa, 2002.

[56] J. Kiusalaas. *Numerical Methods in Engineering with Python*. Cambridge University Press, New York, 2005.

[57] G. Dahlquist and Å. Björck. *Numerical Methods in Scientific Computing*. to be published by SIAM, Philadelphia, `http://www.mai.liu.se/~akbjo/NMbook.html`.

[58] R. P. Brent. *Algorithms for Minimization without Derivatives.* Prentice Hall, 1973.

[59] J. H. Mathews and K. D. Fink. *Numerical methods using Matlab.* Prentice Hall, 1999.

[60] Fundamental theorem of algebra: `http://en.wikipedia.org/wiki/Fundamental_theorem_of_algebra`.

[61] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in Real Algebraic Geometry.* Springer, 2003.

[62] J. Stoer. *Wstęp do metod numerycznych. Tom pierwszy.* Państwowe Wydawnictwo Naukowe, Warszawa, 1979.

[63] Jorge J. Moré, Burton S. Garbow, and Kenneth E. Hillstrom. User guide for MINPACK-1. Technical Report ANL-80-74, Argonne National Laboratory, Argonne, IL, USA, August 1980.

[64] Minpack - Least Squares Minimization: `http://www.csit.fsu.edu/~burkardt/f_src/minpack/minpack.html`.

[65] M. Powell. A hybrid method for nonlinear equations. In P. Rabinowitz, editor, *Numerical Methods for Nonlinear Algebraic Equations.* Gordon and Breach Science, London, 1970.

[66] F. Leja. *Rachunek Różniczkowy i Całkowy.* PWN, Warszawa, 1959.

[67] M. K. Bowen and R. Smith. Derivative formulas and errors for non-uniformly spaced points. *Proc. Roy. Soc. A*, 461:1975–1997, 2005.

[68] B. Fornberg. Generation of finite difference formlas on arbitrarily spaced grids. *Math. Comput. 51*, 51:699–706, 1988.

[69] C. J. F. Ridders. *Advances in Engineering Software*, 4:75–76, 1982.

[70] Numerical Recipes Public Domain Area: `http://www.numerical-recipes.com/public-domain.html`.

[71] Numerical Algorithms Group: `http://www.nag.co.uk/`.

[72] Strona główna ADIFOR: `http://www-unix.mcs.anl.gov/autodiff/ADIFOR/`.

[73] Gnuplot tips (not so Frequently Asked Questions): `http://t16web.lanl.gov/Kawano/gnuplot/index-e.html`.

[74] Aproksymacja Pade: `http://mathworld.wolfram.com/PadeApproximant.html`.

[75] A. Vaught. Introduction to Named Pipes. *Linux Journal*, `http://www2.linuxjournal.com/article/2156`.

Skorowidz