6. Błędy w obliczeniach

Komputery potrafią płatać figle. Zdarza się, że choć wykonują obliczenia matematycznie poprawne, zwracają błędne wyniki. Warto znać przyczyny powstawania tych błędów, aby np. móc projektować systemy odporne na wynikające z nich niebezpieczeństwa. Dlatego w tym temacie poznasz podstawowe zagadnienia dotyczące błędów numerycznych.

Cele lekcji

- Poznasz różne przyczyny i rodzaje błędów w obliczeniach komputerowych.
- Dowiesz się, czym się różni błąd bezwzględny od względnego.
- Zrozumiesz, czym się charakteryzują algorytmy niestabilny i stabilny.
- Znajdziesz pierwiastki równania kwadratowego, wykorzystując zarówno algorytm niestabilny, jak i algorytm stabilny.

6.1. Źródła błędów w obliczeniach

Wiesz już, że liczby rzeczywiste są pamiętane w komputerze na skończonej liczbie bitów. Skutkuje to zaokrąglaniem reprezentacji binarnych wielu z nich, np. liczb wymiernych mających nieskończone rozwinięcie binarne i liczb niewymiernych. Takie reprezentacje binarne są więc obarczone pewnym błędem. Nazywamy go **błędem reprezentacji**.

Błąd reprezentacji

Kiedy wykonujemy działania arytmetyczne na liczbach reprezentowanych na skończonej liczbie bitów, do zapamiętania wyniku często potrzebujemy większej liczby bitów. Na przykład podczas **mnożenia dwóch liczb zmiennoprzecinkowych** mnożymy ich mantysy. Jeśli mantysy są reprezentowane na n bitach, to na wynik mnożenia warto byłoby przeznaczyć $2n$ bitów. Jednak wynik musimy reprezentować także na n bitach, dlatego trzeba go zaokrąglić. Powstaje błąd, który nazywamy **błędem zaokrąglenia**.

Mnożenie i dzielenie liczb zmiennoprzecinkowych, s. 109

Błąd zaokrąglenia

Konsekwencje błędów reprezentacji i zaokrąglenia można zaobserwować po wykonaniu programu dodającego liczby 0,1, dopóki wynik jest różny od 1. Oto fragment kodu źródłowego tego programu:

Fragment kodu źródłowego programu dodającego liczby 0,1, dopóki wynik jest różny od 1

```
int main()
{
    float x=0;
    while (x!=1)
    {
        x=x+0.1;
        cout<<x<<endl;
    }
    return 0;
}
```

Warto wiedzieć

W przykładach używamy liczb pojedynczej precyzji (typ **float**), aby łatwiej pokazać błędy w obliczeniach.

Program powinien zakończyć działanie po wykonaniu dziesięciu instrukcji `x=x+0.1`, czyli osiągnięciu przez zmienną `x` wartości `1`. Niestety program będzie działał w nieskończoność – nigdy nie osiągnie dokładnej wartości 1, ponieważ liczba 0,1 ma nieskończone rozwinięcie binarne (równe $0{,}0(0011)_2$) i jest pamiętana przez komputer w przybliżeniu. Konsekwencje przybliżania liczb mających nieskończone rozwinięcie binarne obserwowaliśmy już w **programie dodającym 100 000 liczb 0,1**, przedstawionym w poprzednim temacie.

Kod źródłowy programu dodającego 100 000 liczb 0,1 oraz wskazującego różnicę między otrzymanym a poprawnym wynikiem, s. 104

Zmodyfikujmy program tak, aby zakończył działanie po przekroczeniu przez zmienną `x` wartości `1`, a po każdym wyliczeniu wartości `x+0.1` wypisał wynik z dziesięcioma cyframi części ułamkowej. Oto fragment kodu źródłowego zmodyfikowanego programu:

Fragment kodu źródłowego programu dodającego liczby 0,1, do momentu, gdy wynik przekroczy wartość 1

```
1.  int main()
2.  {
3.      float x=0;
4.      cout<<setprecision(10);
5.      while (x<1)
6.      {
7.          x=x+0.1;
8.          cout<<x<<endl;
9.      }
10.     return 0;
11. }
```

W linii 4 wykorzystaliśmy **funkcję `setprecision`**, pozwalającą określić, ile pozycji części ułamkowej zostanie wyświetlonych. Wyniki zwracane przez program będą zaokrąglane. Jeśli cyfry od pewnego miejsca do końca wyniku są zerami, to nie zostaną wypisane. Funkcja `setprecision` jest zdefiniowana w bibliotece `iomanip`, dlatego w programie należy użyć dyrektywy `#include <iomanip>`. Rysunek 6.1 przedstawia efekt wykonania powyższego programu.

Funkcja `setprecision`

```
0.1000000015
0.200000003
0.3000000119
0.400000006
0.5
0.6000000238
0.7000000477
0.8000000715
0.9000000954
1.000000119
```

Rys. 6.1. Kolejne wyniki zwracane przez program dodający liczby 0,1, do momentu, gdy wynik przekroczy wartość 1

Warto wiedzieć

Funkcja `setprecision` (podobnie jak poznana wcześniej funkcja `setw`) jest przykładem manipulatora strumienia. Manipulatory strumieni są dostępne m.in. w bibliotekach `iostream` i `iomanip`.

Ponieważ wartości rzeczywiste wyliczane przez program są zaokrąglane, nie należy ich porównywać. Najczęściej w programach wprowadza się stałą, która informuje o **dokładności obliczeń**. Kiedy błąd jest mniejszy od tej stałej, kończymy obliczenia. Otrzymujemy rozwiązanie przybliżone obarczone pewnym błędem – tzw. **błędem przybliżenia**.

Dokładność obliczeń

Błąd przybliżenia

W programie przedstawionym powyżej moglibyśmy wprowadzić stałą, np. o nazwie EPS. Jej definicja wyglądałaby następująco.

> **Warto wiedzieć**
> Nazwa stałej EPS pochodzi od greckiej litery epsilon (ε).

```
const float EPS=0.01;
```

Wówczas warunek pętli określimy w następujący sposób:

```
while (abs(x-1)>EPS)
```

Ponieważ dodajemy za każdym razem wartość 0,1, wystarczy nam dokładność obliczeń równa 0,01. **Funkcja** abs z biblioteki cmath zwraca wartość bezwzględną liczby zmiennoprzecinkowej. W naszym przypadku parametrem funkcji jest różnica między otrzymaną wartością zmiennej x a oczekiwanym wynikiem 1.

Funkcja abs

Ćwiczenie 1

Napisz program, który będzie dodawał wartość 0.1 do zmiennej x, aż otrzyma wartość 1 z dokładnością określoną przez stałą. Wartość początkowa zmiennej x wynosi 0. Program powinien wypisywać wartości pośrednie dodawania z dziesięcioma cyframi po kropce dziesiętnej.

Przy obliczaniu pewnych wartości na komputerze trzeba ograniczyć liczbę wykonywanych operacji. Na przykład wartość funkcji sinus dla zmiennej rzeczywistej x można obliczyć ze wzoru:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \dots$$

> **Warto wiedzieć**
> Do obliczenia przybliżonej wartości dowolnej funkcji trygonometrycznej wystarczą cztery podstawowe działania arytmetyczne.

Według wzoru działania wykonywane są w nieskończoność. Jednak algorytm obliczający wartość funkcji sinus jest skończony – wykonuje skończoną liczbę operacji (pomija od pewnego momentu liczby o małych wartościach, niemających dużego wpływu na wynik). Stanowi to kolejne źródło błędów w obliczeniach. Błąd tego typu to **błąd obcięcia**.

Błąd obcięcia

Niektóre działania wykonywane przez komputer na określonych danych powodują kumulację błędów. Może tak się stać np. podczas odejmowania bliskich sobie liczb, dzielenia dużej liczby przez bardzo małą lub odejmowania bardzo małej liczby od dużej. Błędy mogą kumulować się także, gdy wykonujemy wiele działań arytmetycznych. Czasami kumulacja błędów może prowadzić do bardzo niedokładnych wyników pomimo małych błędów danych wejściowych.

> **Zapamiętaj**
> Istnieją różne źródła błędów w obliczeniach komputerowych. Jednym z nich jest błąd reprezentacji, wynikający z zaokrąglania reprezentacji binarnej liczb zmiennoprzecinkowych. Działania arytmetyczne na takich liczbach powodują błąd zaokrąglenia wyniku. Często też stosuje się algorytmy, które zwracają wynik przybliżony. Taka wartość jest obarczona błędem przybliżenia lub obcięcia.

6.2. Błąd bezwzględny i względny

Błąd bezwzględny informuje, o ile wartość dokładna różni się od otrzymanej przybliżonej wartości. Oblicza się go ze wzoru:

Błąd bezwzględny

$|x - x_p|$, gdzie x – wartość dokładna, a x_p – wartość przybliżona

Na przykład jeśli liczbę dziesiętną 100,5 zaokrąglimy do wartości całkowitej, to błąd bezwzględny tego przybliżenia wynosi 0,5. Taki sam błąd bezwzględny otrzymamy, gdy zaokrąglimy liczbę 0,5 do wartości całkowitej, choć w tym przypadku wartość zaokrąglona jest dwa razy większa od dokładnej.

Lepszym oszacowaniem wielkości błędu jest **błąd względny**. Określa on, o jaką część wartości dokładnej różni się wartość przybliżona. Błąd względny obliczamy ze wzoru:

Błąd względny

$\left|\frac{x - x_p}{x}\right|$, gdzie x – wartość dokładna, x_p – wartość przybliżona

Czasami wartość błędu względnego wyraża się w procentach. Wówczas obliczamy ją ze wzoru:

$\left|\frac{x - x_p}{x}\right| \cdot 100\%$, gdzie x – wartość dokładna, x_p – wartość przybliżona

W przypadku zaokrąglenia liczby 100,5 do wartości całkowitej błąd względny wynosi niecałe 0,5%, a w przypadku zaokrąglenia liczby 0,5 do wartości całkowitej – aż 100%.

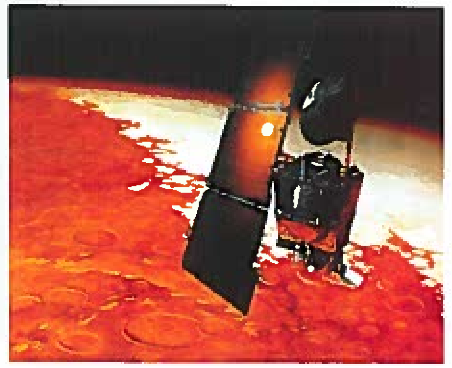
6.3. Algorytmy znajdowania pierwiastków równania kwadratowego

Napiszemy programy znajdujące rozwiązanie równania kwadratowego $ax^2 + bx + c = 0$, gdzie $a \neq 0$, czyli programy wyznaczające pierwiastki równania. Rozpatrzymy dwa algorytmy, które dla tych samych danych dadzą różne wyniki, wynikające z błędów reprezentacji danych.

A to ciekawe

Niespodziewany finał podróży

11 grudnia 1998 r. w kierunku Marsa wystartowała sonda NASA – Mars Climate Orbiter – której zadaniem było badanie Czerwonej Planety. 23 września 1999 r., po blisko 300 dniach podróży, sonda dotarła do Marsa. Jednak w czasie wchodzenia na orbitę utracono z nią kontakt. Okazało się, że Mars Climate Orbiter znalazł się ok. 100 km za blisko powierzchni planety i prawdopodobnie uległ zniszczeniu w jej atmosferze. Dochodzenie wykazało, że główną przyczyną niepowodzenia był błąd ludzki. Polegał on na używaniu do sterowania sondą danych wyrażonych w jednostkach z różnych systemów miar: anglosaskiego i metrycznego.



Znajdowanie pierwiastków równania kwadratowego – algorytm 1

Tradycyjny algorytm, nazywany czasem algorytmem z deltą, znasz z lekcji matematyki. Zgodnie z nim najpierw należy policzyć wyróżnik równania kwadratowego, oznaczany literą Δ, a następnie wyliczyć **pierwiastki**.

Pierwiastki równania kwadratowego to rozwiązania równania kwadratowego $ax^2 + bx + c = 0$, gdzie $a \neq 0$. Liczba pierwiastków zależy od wartości wyróżnika (którą obliczamy ze wzoru: $\Delta = b^2 - 4ac$)
- jeśli $\Delta > 0$, to równanie ma dwa pierwiastki: $x_1 = \frac{-b-\sqrt{\Delta}}{2a}$, $x_2 = \frac{-b+\sqrt{\Delta}}{2a}$,
- jeśli $\Delta = 0$, to równanie ma jeden pierwiastek: $x_1 = x_2 = \frac{-b}{2a}$,
- jeśli $\Delta < 0$, to równanie nie ma pierwiastków.

Sformułujmy specyfikację problemu znajdowania pierwiastków równania kwadratowego oraz zapiszmy w pseudokodzie algorytm tradycyjny z deltą, wyznaczający pierwiastki równania.

Specyfikacja

Dane: a, b, c – liczby rzeczywiste będące współczynnikami trójmianu kwadratowego $ax^2 + bx + c$, gdzie $a \neq 0$.

Wynik: x1, x2 – liczby rzeczywiste będące pierwiastkami równania kwadratowego $ax^2 + bx + c = 0$, gdzie $a \neq 0$, lub komunikat „Brak pierwiastków".

```
delta ← b * b – 4 * a * c
jeśli delta < 0 to wypisz "Brak pierwiastków"
w przeciwnym przypadku
    pdelta ← pierwiastek(delta)
    x1 ← (-b – pdelta)/(2 * a)
    x2 ← (-b + pdelta)/(2 * a)
    wypisz x1, x2
```

Fragment kodu źródłowego programu realizującego powyższy algorytm wygląda następująco:

Fragment kodu źródłowego programu wyznaczającego pierwiastki równania kwadratowego – algorytm 1

```
1.  int main()
2.  {
3.      float a, b, c, delta, pdelta, x1, x2;
4.      cout<<"a = "; cin>>a;
5.      cout<<"b = "; cin>>b;
6.      cout<<"c = "; cin>>c;
7.      delta=b*b-4*a*c;
8.      if (delta<0) cout<<"Brak pierwiastkow";
9.      else
10.     {
11.         pdelta=sqrt(delta);
12.         x1=(-b-pdelta)/(2*a);
13.         x2=(-b+pdelta)/(2*a);
14.         cout<<"x1 = "<<x1<<endl;
15.         cout<<"x2 = "<<x2;
16.     }
17.     return 0;
18. }
```


W linii 11 korzystamy z **funkcji** `sqrt`, obliczającej pierwiastek kwadratowy liczby. Funkcja ta znajduje się w bibliotece `cmath`, więc należy pamiętać o dodaniu w programie linii dołączającej tę bibliotekę – `#include <cmath>`.

Funkcja `sqrt`

Rysunek 6.2 przedstawia wynik działania programu dla równania $x^2 + 10\,000x + 1 = 0$.

```
a = 1
b = 10000
c = 1
x1 = -10000
x2 = 0
```

Rys. 6.2. Wynik działania programu dla równania $x^2 + 10000x + 1 = 0$

Sprawdźmy, czy wybrany pierwiastek równania, np. x2, jest rzeczywiście równy wyliczonej przez program wartości – liczbie 0. Po podstawieniu wartości pierwiastka do równania otrzymamy sprzeczność: jeden równa się zero. Zatem liczba 0 nie jest poszukiwanym pierwiastkiem.

Ćwiczenie 2

Napisz program znajdujący pierwiastki równania kwadratowego z wykorzystaniem przedstawionego w temacie algorytmu z deltą. Sprawdź działanie programu dla różnych danych.

W przykładzie na rysunku 6.2 danymi są liczby całkowite, ale w programie są one reprezentowane jako liczby zmiennoprzecinkowe (typ **float**). Program wykonał tylko kilka działań arytmetycznych, a otrzymał wynik z dużym **błędem względnym**. Jego przyczyną jest odejmowanie liczb. W naszym przypadku wartość iloczynu `4*a*c` jest bardzo mała względem wartości `b*b`, więc wartość `delta` jest w przybliżeniu równa wartości `b*b`. Powoduje to zaokrąglenie wartości pierwiastka x2 do 0.

Błąd względny, s. 115

Przedstawiony algorytm jest przykładem **algorytmu niestabilnego**. Charakteryzuje się on tym, że dla pewnych danych, które mogą być obarczone małym błędem względnym, powoduje duży błąd względny w wyniku.

Algorytm niestabilny

Ćwiczenie 3

Podaj przykład wartości a, b i c, innych niż na rysunku 6.2, dla których program znajdujący pierwiastki równania kwadratowego $ax^2 + bx + c = 0$, gdzie $a \neq 0$, za pomocą algorytmu tradycyjnego z deltą zachowuje się niestabilnie.

Znajdowanie pierwiastków równania kwadratowego – algorytm 2

Skonstruujemy teraz algorytm znajdujący rozwiązanie równania kwadratowego, w którym otrzymane wyniki będą się nieznacznie różniły od oczekiwanych. Jeśli algorytm dla danych, które mogą być obarczone niewielkimi błędami, zawsze zwraca wyniki nieznacznie tylko zaburzone, to nazywamy go **algorytmem stabilnym**.

Algorytm stabilny

W algorytmie jeden z pierwiastków policzymy z wykorzystaniem jednego ze wzorów na pierwiastek kwadratowy równania. Wybierzemy ten wzór, w którym nie wystąpi odejmowanie liczb prawie równych co do wartości. Drugi pierwiastek wyznaczymy, korzystając ze wzoru Viète'a na iloczyn pierwiastków: $x_1 \cdot x_2 = \frac{c}{a}$.

Warto wiedzieć

W algorytmie stabilnym obliczającym pierwiastki równania kwadratowego nie stosujemy wzoru Viète'a na sumę pierwiastków ($x_1 + x_2 = -\frac{b}{a}$), ponieważ moglibyśmy odejmować dwie liczby bliskie co do wartości (gdy pierwiastki mają różne znaki). To mogłoby prowadzić do otrzymania wyniku z dużym błędem względnym.

Oto zapis algorytmu w pseudokodzie:

```
delta ← b * b − 4 * a * c
jeśli delta < 0 to wypisz "Brak pierwiastków"
w przeciwnym przypadku
   pdelta ← pierwiastek(delta)
   jeśli b > 0 to
      x1 ← (−b − pdelta)/(2 * a)
   w przeciwnym przypadku
      x1 ← (−b + pdelta)/(2 * a)
   jeśli x1 ≠ 0 to x2 ← c/(a * x1)
   w przeciwnym przypadku x2 ← 0
   wypisz x1, x2
```

Dla współczynnika `b > 0` wybieramy wzór na pierwiastek, w którym odejmujemy wartość `pdelta` (liczby `–b` i `-pdelta` są niedodatnie), w przeciwnym przypadku – wzór na pierwiastek, w którym dodawana jest wartość `pdelta` (liczby `–b` i `+pdelta` są nieujemne). Zabezpieczamy się w ten sposób przed sytuacją, w której odejmiemy od siebie dwie liczby o prawie równych wartościach. Drugi pierwiastek obliczamy ze wzoru Viète'a pod warunkiem, że pierwszy jest różny od zera. Jeśli `x1 = 0`, to współczynnik `c` jest równy `0`, a w konsekwencji wartość `pdelta` jest równa |`b`|. Podczas obliczania `x1` w liczniku otrzymujemy wartość 2`b` lub –2`b`, więc współczynnik `b` ma także wartość 0. Równanie ma postać $ax^2 = 0$, dlatego gdy `x1 = 0`, wystarczy przyjąć `x2 = 0`.

Fragment kodu źródłowego programu realizującego algorytm zapisany powyżej w pseudokodzie może wyglądać następująco:

Fragment kodu źródłowego programu wyznaczającego pierwiastki równania kwadratowego – algorytm 2

```
1.  const float EPS=0.0000001;
2.
3.  int main()
4.  {
5.      float a, b, c, delta, pdelta, x1, x2;
6.      cout<<"a = "; cin>>a;
7.      cout<<"b = "; cin>>b;
8.      cout<<"c = "; cin>>c;
9.      delta=b*b-4*a*c;
10.     if (delta<0) cout<<"Brak pierwiastkow";
11.     else
12.     {
13.         pdelta=sqrt(delta);
14.         if (b>0)
15.             x1=(-b-pdelta)/(2*a);
16.         else
17.             x1=(-b+pdelta)/(2*a);
18.         if (abs(x1)>EPS) x2=c/(a*x1);
19.         else x2=0;
20.         cout<<"x1 = "<<x1<<endl;
21.         cout<<"x2 = "<<x2;
22.     }
23.     return 0;
24. }
```


Zwróć uwagę na zapis w linii 18. Wartość zmiennej `x1` porównujemy z zerem z dokładnością do stałej EPS. Jej wartość (linia 1) jest bardzo mała – na poziomie dokładności liczb pojedynczej precyzji.

Rysunek 6.3 przedstawia wynik działania programu z wykorzystaniem wzoru Viète'a dla równania $x^2 + 10\,000x + 1 = 0$.

```
a = 1
b = 10000
c = 1
x1 = -10000
x2 = -0.0001
```

Rys. 6.3. Wynik działania programu z wykorzystaniem wzoru Viète'a dla równania $x^2 + 10\,000x + 1 = 0$

Zwróć uwagę, że pierwiastek `x1` także jest policzony w przybliżeniu. Błąd przybliżenia jest jednak bardzo mały. Dokładne wartości pierwiastków równania $x^2 + 10\,000x + 1 = 0$ są następujące:

$x_1 = -5000 - \sqrt{24999999}$

$x_2 = -5000 + \sqrt{24999999}$

Dobra rada

Kiedy porównujesz wartości rzeczywiste, nie rób tego wprost, za pomocą równości lub nierówności. Korzystaj z relacji mniejszości lub większości z pewną dokładnością.

Ćwiczenie 4

Napisz program znajdujący pierwiastki równania kwadratowego z wykorzystaniem wzoru Viète'a na iloczyn pierwiastków. Sprawdź działanie programu dla różnych danych.

Zapamiętaj

Algorytm stabilny to algorytm, który dla danych obarczonych niewielkimi błędami względnymi zwróci wynik obarczony niewielkim błędem względnym. Algorytm niestabilny to algorytm, który dla danych obarczonych niewielkimi błędami względnymi może zwrócić wynik z dużym błędem względnym.

Podsumowanie

- Błąd reprezentacji powstaje w wyniku zaokrąglenia liczby zmiennoprzecinkowej.
- Błąd zaokrąglenia powstaje wskutek zaokrąglenia wyniku działań arytmetycznych na liczbach zmiennoprzecinkowych.
- Błąd obcięcia powstaje w wyniku wykonania skończonej liczby iteracji podczas obliczania wartości, które są wyrażone np. sumą nieskończenie wielu wyrazów.
- Błąd przybliżenia powstaje, gdy algorytm znajduje rozwiązanie problemu z pewną dokładnością.
- Błąd bezwzględny to wartość bezwzględna różnicy pomiędzy wartością dokładną i wartością przybliżoną. Informuje on, o ile wartość przybliżona różni się od dokładnej.
- Błąd względny określa, o jaką część wartości dokładnej różni się wartość przybliżona. Błąd względny można wyrazić w procentach.
- Algorytm stabilny to algorytm, który dla danych obarczonych niewielkimi błędami względnymi zwraca wynik obarczony niewielkim błędem względnym.
- Algorytm niestabilny to algorytm, który dla danych obarczonych niewielkimi błędami względnymi może zwrócić wynik z dużym błędem względnym.
- Przykładem algorytmu niestabilnego jest tzw. algorytm z deltą, obliczający pierwiastki równania kwadratowego.
- Pierwiastki równania kwadratowego można wyznaczyć algorytmem stabilnym z wykorzystaniem wzoru Viète'a na iloczyn pierwiastków równania.

Zadania

• **1** Oblicz błąd względny reprezentacji liczby dziesiętnej 0,1, jeśli na część ułamkową przeznaczonych jest 8 bitów.
Wskazówka: Możesz skorzystać z tabeli 5.3 na s. 104.

• **2** Zmodyfikuj wybrany program znajdujący pierwiastki równania kwadratowego tak, aby rozpoznawał sytuację, gdy wyróżnik równania kwadratowego jest równy 0. Pamiętaj, aby nie porównywać wartości rzeczywistych wprost, ale z pewną dokładnością.

•• **3** W programie znajdującym pierwiastki równania kwadratowego tradycyjnym algorytmem z deltą zamień typ pojedynczej precyzji (`float`) na typ podwójnej precyzji (`double`). Sprawdź działanie programu dla równania $x^2 + 10\,000x + 1 = 0$. Podaj przykład danych, dla których program zachowa się niestabilnie.

•• **4** Napisz program, który obliczy wartość sin 1, korzystając ze wzoru:

$$\sin 1 = 1 - \frac{1}{3!} + \frac{1}{5!} - \frac{1}{7!} + \frac{1}{9!} - \frac{1}{11!} + \ldots$$

Obliczenia wykonaj z dokładnością do 4 miejsc po kropce dziesiętnej.

•• **5** Napisz program, który obliczy wartość cos 1, korzystając ze wzoru:

$$\cos 1 = 1 - \frac{1}{2!} + \frac{1}{4!} - \frac{1}{6!} + \frac{1}{8!} - \frac{1}{10!} + \ldots$$

Obliczenia wykonaj z dokładnością do 4 miejsc po kropce dziesiętnej.

••• **6** Napisz program, który obliczy wartość sin x, korzystając ze wzoru:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \ldots$$

dla $0 \leqslant x \leqslant 6{,}28$. Obliczenia wykonaj z dokładnością do 4 miejsc po kropce dziesiętnej.

••• **7** Napisz program, który obliczy wartość cos x, korzystając ze wzoru:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \frac{x^{10}}{10!} + \ldots$$

dla $0 \leqslant x \leqslant 6{,}28$. Obliczenia wykonaj z dokładnością do 4 miejsc po kropce dziesiętnej.

••• **8** Napisz program rozwiązujący układ równań liniowych

$$\begin{cases} a_1x + b_1y + c_1 = 0 \\ a_2x + b_2y + c_2 = 0 \end{cases}$$

metodą wyznaczników. Podaj przykład danych, dla których program zachowa się niestabilnie.