

Działania na liczbach

„Każdy skomplikowany problem ma rozwiązanie,
które jest jasne, proste i niepoprawne.”

— H. L. Mencken

W tym rozdziale dokonamy przeglądu fundamentalnych językowych i bibliotecznych narzędzi do wykonywania działań na liczbach. Opiszemy podstawowe problemy dotyczące rozmiaru, precyzji i obcinania liczb. Centralną częścią tego rozdziału będzie opis tablic wielowymiarowych — zarówno w stylu języka C, jak i biblioteki macierzy N -wymiarowych. Przedstawimy wprowadzenie do liczb losowych, które są często potrzebne do testowania, przeprowadzania symulacji oraz w grach. Na koniec przedstawimy listę standardowych funkcji matematycznych oraz zwięźle opiszemy funkcjonalność biblioteki standardowej w zakresie liczb zespolonych.

24.1. Wprowadzenie

24.2. Rozmiar, precyzja i przekroczenie zakresu

24.2.1. Ograniczenia typów liczbowych

24.3. Tablice

24.4. Tablice wielowymiarowe w stylu języka C

24.5. Biblioteka Matrix

24.5.1. Wymiary i dostęp

24.5.2. Macierze jednowymiarowe

24.5.3. Macierze dwuwymiarowe

24.5.4. Wejście i wyjście macierzy

24.5.5. Macierze trójwymiarowe

24.6. Przykład — rozwiązywanie równań liniowych

24.6.1. Klasyczna eliminacja Gaussa

24.6.2. Wybór elementu centralnego

24.6.3. Testowanie

24.7. Liczby losowe

24.8. Standardowe funkcje matematyczne

24.9. Liczby zespolone

24.10. Źródła

24.1. Wprowadzenie



Dla niektórych poważne działania na liczbach są wszystkim. Do kategorii tej zalicza się wielu naukowców, inżynierów oraz statystyków. Dla niektórych działania na liczbach są czasami niezbędne. Do tej kategorii można zaliczyć informatyka od czasu do czasu współpracującego z fizykiem. Większość ludzi działania na liczbach — poza prostą arytmetyką liczb całkowitych i zmiennoprzecinkowych — wykonuje rzadko. Celem tego rozdziału jest przedstawienie technicznych możliwości języka w zakresie rozwiązywania prostych problemów związanych z wykonywaniem obliczeń. Nie próbujemy tu uczyć analizy numerycznej ani opisywać zaawansowanych zagadnień związanych z działaniami na liczbach zmiennoprzecinkowych. Zagadnienia te znacznie wykraczają poza tematykę tej książki i mieszają się ze szczegółowymi tematami związanymi z konkretnymi dziedzinami zastosowań. W tym rozdziale opiszemy:

- Zagadnienia związane z typami wbudowanymi o stałym rozmiarze, takie jak precyzja i przepełnienie zakresu.
- Tablice — zarówno wbudowaną implementację tablic wielowymiarowych, jak i bibliotekę *Matrix*, która lepiej nadaje się do wykonywania obliczeń.
- Podstawową definicję liczb losowych.
- Funkcje matematyczne biblioteki standardowej.
- Liczby zespolone.

Szczególny nacisk położymy na bibliotekę *Matrix*, która znacząco ułatwia pracę z macierzami (wielowymiarowymi tablicami).

24.2. Rozmiar, precyzja i przekroczenie zakresu



Jeśli wykorzystywane są typy wbudowane i stosowane typowe techniki obliczeniowe, liczby przechowywane są w stałych ilościach pamięci. Tzn. typy całkowitoliczbowe (*int*, *long* itp.) są tylko przybliżeniem matematycznego pojęcia liczb całkowitych, a typy zmiennoprzecinkowe (*float*, *double* itp.) są (tylko) przybliżeniem matematycznych liczb rzeczywistych. Oznacza to, że z matematycznego punktu widzenia niektóre obliczenia są nieprecyzyjne lub wręcz niepoprawne. Rozważmy:

```
float x = 1.0/333;  
float sum = 0;  
for (int i=0; i<333; ++i) sum+=x;  
cout << setprecision(15) << sum << "\n";
```

Powyższy kod nie zwróci wartości 1, jak można by było się naiwnie spodziewać, tylko:

```
0.999999463558197
```

Spodziewaliśmy się takiego wyniku. Mamy tu do czynienia z błędem zaokrąglania. Liczba zmiennoprzecinkowa dysponuje ograniczoną liczbą bitów, przez co można ją „oszukać”, pisząc takie wyrażenie arytmetyczne, którego wynik wymaga większej liczby bitów, niż udostępnia sprzęt. Na przykład wymiernej liczby $1/3$ nie da się precyzyjnie zaprezentować w postaci

dziesiątej bez względu na to, ilu użyje się cyfr. To samo dotyczy liczby $1/333$. Jeśli zatem dodamy 333 kopie x (najlepszego maszynowego przybliżenia liczby $1/333$ jako liczby typu `float`), uzyskamy wynik trochę inny niż 1 . Zawsze, gdy intensywnie wykorzystywane są liczby zmiennoprzecinkowe, występują błędy zaokrąglania. Kwestia dotyczy tylko tego, czy błędy te znacząco wpłyną na wynik.

Zawsze należy sprawdzać, czy uzyskany wynik jest prawdopodobny. Każdy, kto wykonuje obliczenia, powinien mniej więcej orientować się, jak powinien wyglądać poprawny wynik, aby nie dać się oszukać jakiemuś „głupiemu błędowi w kodzie” lub błędowi w obliczeniach. Należy mieć świadomość możliwości wystąpienia błędów zaokrąglania oraz w razie wątpliwości skonsultować się z jakimś specjalistą lub doczytać na temat wykonywania obliczeń.



WYPRÓBUJ



Zastąp w powyższym programie liczbę 333 liczbą 10 i uruchom go. Jakiego wyniku się spodziewałeś? Jaki wynik został zwrócony? Ostrzegaliśmy!

Efekt stałości rozmiaru liczb całkowitych może dać o sobie znać w bardziej dramatycznych okolicznościach. Powodem tego jest to, że liczby zmiennoprzecinkowe z definicji są przybliżeniem liczb rzeczywistych, a więc bywa, że są nieprecyzyjne (tj. gubią najmniej znaczące bity). Natomiast liczby całkowite czasami mają niewystarczający zakres (tj. gubią najbardziej znaczące bity). To sprawia, że błędy dotyczące liczb zmiennoprzecinkowych są często trudne do zauważenia (i często pozostają niezauważone przez początkujących). Natomiast błędy dotyczące liczb całkowitych są często widowiskowe (i zazwyczaj trudno ich nie zauważyć). Należy pamiętać, że lepiej jest, gdy błędy manifestują swoją obecność wcześniej i wyraźnie, dzięki czemu łatwiej je znaleźć i poprawić.



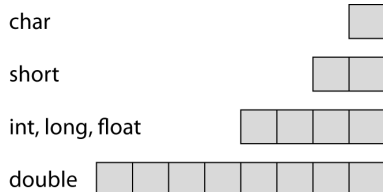
Rozważymy problem z liczbami całkowitymi:


```
short int y = 40000;
int i = 1000000;
cout << y << " " << i*i << "\n";
```

Powyższy kod zwróci następujący wynik:

```
-25536      -727379968
```

Tego się spodziewaliśmy. Jest to efekt przekroczenia zakresu liczby. Typy całkowitoliczbowe pozwalają reprezentować tylko względnie małe liczby. Nie ma po prostu wystarczającej liczby bitów, aby precyzyjnie zaprezentować każdą potrzebną liczbę w odpowiedni sposób do efektywnego wykonania obliczeń. Dwubajtowa liczba całkowita typu `short` nie może mieć wartości $40\,000$, a czterobajtowa liczba typu `int` nie może mieć wartości $1\,000\,000\,000\,000$. Dokładny rozmiar typów wbudowanych języka `C++` (punkt A.8) zależy od sprzętu i kompilatora. Funkcja `sizeof(x)` zwraca rozmiar x w bajtach dla zmiennej x lub typu x . Z definicji `sizeof(char)=1`. Rozmiary można przedstawić graficznie w następujący sposób:



 Przedstawione wartości dotyczą systemu Windows i kompilatora Microsoftu. Język C++ udostępnia liczby całkowite i zmiennoprzecinkowe różnych rozmiarów, ale jeśli jest to możliwe, najlepiej trzymać się typów char, int oraz double. W większości programów (ale nie we wszystkich) pozostałe typy liczbowe sprawiają więcej kłopotów, niż są warte.


Liczbę całkowitą można przypisać do typu zmiennoprzecinkowego. Jeśli liczba ta jest większa niż zakres owego typu, zostanie utracona precyzja. Na przykład:

```
cout << "Rozmiary: " << sizeof(int) << ' ' << sizeof(float) << '\n';
int x = 2100000009; // duża liczba całkowita
float f = x;
cout << x << ' ' << f << endl;
cout << setprecision(15) << x << ' ' << f << '\n';
```

Na naszym komputerze kod ten zwrócił następujący wynik:


```
Rozmiary: 4 4
2100000009 2.1e+009
2100000009 2100000000
```

Typy float i int zajmują takie same ilości pamięci — 4 bajty. Liczba typu float składa się z mantysy (zazwyczaj wartości z przedziału między 0 a 1) i wykładnika (mantysa $\cdot 10^{\text{wykładnik}}$), a więc nie może precyzyjnie reprezentować największej liczby typu int (gdybyśmy próbowali to zrobić, gdzie znaleźlibyśmy miejsce na mantysę, gdy część miejsca zabraliśmy dla wykładnika?). Zmienna f zgodnie z przewidywaniami zaprezentowała liczbę 2100000009 w jak najdokładniejszym przybliżeniu. Jednak znajdująca się na końcu cyfra 9 okazała się już zbyt dużym wyzwaniem — dlatego ją tam postawiliśmy.

 Jeśli natomiast do typu całkowitoliczbowego przypisz się liczbę zmiennoprzecinkową, następuje obcięcie, tzn. zostaje opuszczona część ułamkowa. Na przykład:

```
float f = 2.8;
int x = f;
cout << x << ' ' << f << '\n';
```

Zmienna x będzie miała wartość 2. Nie będzie wartości 3, jak można by było się spodziewać zgodnie z ogólnie przyjętą regułą zaokrąglania. Konwersja typu float na int powoduje obcięcie, a nie zaokrąglenie liczby.

 Wykonując obliczenia, należy pamiętać o możliwości przekroczenia zakresu typu lub obcięciu części ułamkowej. Język nie przechwyci tych problemów za nas. Rozważmy:

```
void f(int i, double fpd)
{
    char c = i;      // Tak, typ char reprezentuje naprawdę małe liczby całkowite
    short s = i;     // Uwaga: liczba całkowita może być za duża dla typu short
```

```

i = i+1;           // Co by było, gdyby i była największą liczbą typu int?
long lg = i*i;     // Uwaga: liczba typu long nie może być większa od typu int
float fps = fpd;   // Uwaga: duża liczba typu double może nie zmieścić się w typie float
i = fpd;           // Obcięcie: np. 5.7 -> 5
fps = i;           // Może zostać utracona precyzja (w przypadku bardzo dużych wartości typu int)
}

void g()
{
    char ch = 0;
    for (int i = 0; i<500; ++i)
        cout << int(ch++) << 't';
}

```

Jeśli masz wątpliwości, poeksperymentuj! Nie panikuj i nie ograniczaj się tylko do czytania dokumentacji. Jeśli nie masz dużego doświadczenia, możesz źle zrozumieć dokumentację związaną z wykonywaniem obliczeń.

WYPRÓBUJ

Uruchom funkcję `g()`. Zmodyfikuj funkcję `f()`, aby drukowała na wyjściu `c`, `s`, `i` itd. Przetestuj ją na wielu rozmaitych wartościach.

Bardziej szczegółowo o reprezentacji liczb całkowitych i ich konwertowaniu napiszemy w punkcie 25.5.3. Jeśli się da, wolimy stosować jak najmniej typów danych. W ten sposób łatwiej uniknąć nieporozumień. Na przykład dzięki używaniu w programie tylko typu `double` można uniknąć problemów związanych z konwersją typu `double` na `float`. W istocie staramy się w obliczeniach używać tylko typów `int`, `double` i `complex` (podrozdział 24.8), do reprezentowania znaków używamy typu `char`, a `bool` do wyrażania wartości logicznych. Pozostałych typów arytmetycznych używamy tylko, gdy musimy.

24.2.1. Ograniczenia typów liczbowych

Własności typów wbudowanych implementacji C++ są przechowywane w nagłówkach `<limits>`, `<climits>`, `<limits.h>` oraz `<float.h>`. Można w nich sprawdzić ograniczenia, ustawić wartowników itd. Lista tych wartości znajduje się w punkcie B.9.1 i może mieć kluczowe znaczenie dla programistów niskopoziomowych narzędzi. Jeśli ich potrzebujesz, prawdopodobnie pracujesz za blisko sprzętu, chociaż można mieć różne powody. Często na przykład programiści ciekawią się szczegółami implementacji języka, np. „Jaki rozmiar ma typ `int`?” albo „Czy typ `char` ma znak?”. Znalezienie jednoznacznych i poprawnych odpowiedzi na te pytania w dokumentacji systemu może być trudne, a standard opisuje tylko minimalne wymagania. Można natomiast z łatwością napisać program zwracający te informacje:

```

cout << "Liczba bajtów w typie int: " << sizeof(int) << '\n';
cout << "Największa wartość typu int: " << INT_MAX << endl;
cout << "Najmniejsza wartość typu int: " << numeric_limits<int>::min() << '\n';

if (numeric_limits<char>::is_signed)
    cout << "Typ char ma znak.\n";

```

```

else
    cout << "Typ char nie ma znaku.\n";

char ch = numeric_limits<char>::min() ; // najmniejsza wartość dodatnia
cout << "char o najmniejszej wartości dodatniej: " << ch << '\n';
cout << "wartość int zmiennej char o najmniejszej wartości dodatniej: "
    << int(ch) << '\n';

```

Jeśli pisze się kod, który ma być uruchamiany na kilku rodzajach sprzętu, czasami dostępność tych informacji w programie jest niezwykle przydatna. Alternatywnie można ręcznie wpisać te wartości w kod, ale to niosłoby za sobą ryzyko powstania problemów z utrzymaniem.

Ograniczenia te mogą być także przydatne przy wykrywaniu przekroczenia zakresu.

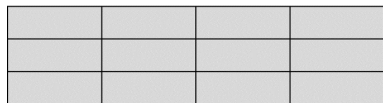
24.3. Tablice

Tablica to sekwencja elementów, do których dostęp można uzyskać za pomocą indeksów (określających ich położenie). Czasami używa się nazwy **wektor**. W tym podrozdziale interesują nas takie tablice, których elementy same są tablicami — tablice wielowymiarowe. Tego rodzaju tablice często nazywa się **macierzami** (ang. *matrix*). Taka różnorodność nazw stanowi dowód popularności i użyteczności tego ogólnego pojęcia. Standardowy typ `vector` (punkt B.4), `array` (podrozdział 20.9) i tablice wbudowane (punkt A.8.2) są jednowymiarowe. Co zrobić, jeśli będzie potrzebny drugi wymiar (np. macierz)? Nie mówiąc już o np. siedmiu?

Jedno- i dwuwymiarowe tablice można graficznie przedstawić następująco:



Wektor (np. `Matrix<int> v(4)`),
zwany także tablicą jednowymiarową
lub macierzą 1 na N

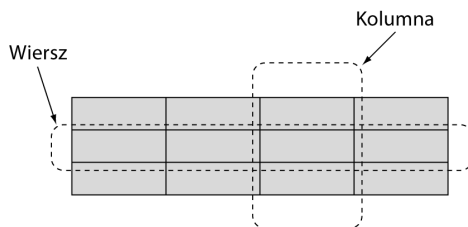


Macierz 3 na 4 (np. `Matrix<int, 2> m(3,4)`) nazywana także
tablicą dwuwymiarową

Tablice mają kluczowe znaczenie w większości programów obliczeniowych. Są powszechnie wykorzystywane w większości najciekawszych obliczeń naukowych, inżynierskich, statystycznych oraz finansowych.



Często tablicę przedstawia się jako obiekt złożony z wierszy i kolumn:



Macierz 3 na 4 także nazywana
tablicą dwuwymiarową
3 wiersze
4 kolumny

Kolumna to sekwencja elementów o wspólnej pierwszej współrzędnej (x). Wiersz to zbiór elementów o wspólnej drugiej współrzędnej (y).

24.4. Tablice wielowymiarowe w stylu języka C

Tablic wbudowanych języka C++ można używać jako tablic wielowymiarowych. Tablicę wielowymiarową traktuje się po prostu jako tablicę tablic, tzn. tablicę, której elementy też są tablicami. Na przykład:

```
int ai[4];           // tablica jednowymiarowa
double ad[3][4];    // tablica dwuwymiarowa
char ac[3][4][5];   // tablica trójwymiarowa
ai[1] = 7;
ad[2][3] = 7.2;
ac[2][3][4] = 'c';
```

Tablice tworzone tą metodą mają wady i zalety tablic jednowymiarowych:



- Zalety:
 - bezpośrednio odwzorowanie w sprzęt,
 - wysoka wydajność w niskopoziomowych operacjach,
 - bezpośrednio wsparcie ze strony języka.
- Wady:
 - Wielowymiarowe tablice w stylu języka C są tablicami tablic (zobacz niżej).
 - Stałe rozmiary (tzn. stałe w czasie kompilacji); aby móc określać rozmiar w czasie działania programu, należy użyć pamięci wolnej.
 - Nie można ich czysto przekazywać — tablica przy najmniejszej prowokacji zamienia się we wskaźnik do swojego pierwszego elementu.
 - Brak sprawdzania zakresu — tablica nie zna swojego rozmiaru.
 - Brak operacji tablicowych — nie ma nawet przypisania (kopiowania).

Wbudowane tablice są powszechnie wykorzystywane w obliczeniach. Stanowią one też **po-ważne** źródło błędów i komplikacji. Dla większości programistów tworzenie i oczyszczanie ich z błędów stanowi nie lada wyzwanie. Jeśli musisz ich używać, poszukaj dodatkowych informacji (np. w książce *Język C++*). Niestety tablice wielowymiarowe języka C++ są takie same jak C, przez co istnieje mnóstwo kodu, który z nich korzysta.

Najgorsze jest to, że nie można po prostu przekazać tablicy wielowymiarowej, przez co trzeba uciekać się do wskaźników i obliczania lokalizacji w tych tablicach. Na przykład:



```
void f1(int a[3][5]);           // Przydatne tylko dla macierzy [3][5]
void f2(int [ ][5], int dim1);  // Pierwszy wymiar może być zmienny
void f3(int [5][ ], int dim2);  // Błąd: drugi wymiar nie może być zmienny
void f4(int [ ][ ], int dim1, int dim2); // Błąd (i tak by nie zadziałało)
void f5(int* m, int dim1, int dim2) // Dziwne, ale działa
{
    for (int i=0; i<dim1; ++i)
        for (int j = 0; j<dim2; ++j) m[i*dim2+j] = 0;
}
```

Tutaj przekazujemy parametr `m` jako typ `int*`, mimo że jest to tablica dwuwymiarowa. Dopóki drugi wymiar musi być zmienny (parametr), nie ma sposobu na poinformowanie kompilatora, że `m` jest tablicą (`dim1, dim2`), a więc przekazujemy tylko wskaźnik na początek obszaru przechowującej ją pamięci. Wyrażenie `m[i*dim2+j]` w rzeczywistości oznacza `m[i,j]`, ale ponieważ kompilator nie wie, że `m` jest tablicą dwuwymiarową, trzeba obliczyć położenie elementu `m[i,j]` w pamięci.

To jest zbyt skomplikowane, prymitywne i podatne na błędy, jak na nasz gust. Ponadto może spowalniać program, ponieważ bezpośrednie obliczanie lokalizacji elementu komplikuje optymalizację. Zamiast szarpać się z tym, opisujemy bibliotekę `C++`, która jest pozbawiona tych wad wbudowanych tablic.

24.5. Biblioteka `Matrix`



Jakie podstawowe cechy powinna mieć tablica lub macierz przeznaczona do użytku w obliczeniach?

- „Mój kod powinien wyglądać bardzo podobnie do tekstu na temat tablic, który czytam w moich podręcznikach do matematyki i inżynierii”.
- Albo o wektorach, macierzach, tensorach.
- Sprawdzanie w czasie kompilacji i działania programu.
 - Tablice o dowolnej liczbie wymiarów.
 - Tablice z dowolną liczbą elementów w wymiarze.
- Tablice są prawidłowymi zmiennymi lub obiektami.
 - Można je przekazywać.
- Typowe operacje tablicowe:
 - Indeksowanie: `()`.
 - Wyznaczanie wycinków: `[]`.
 - Przypisanie: `=`.
 - Skalowanie (`+=`, `-=`, `*=`, `%=` itp.).
 - Łączone operacje wektorowe (np. `res[i]=a[i]*c+b[2]`).
 - Iloczyn skalarny (`res` = suma wszystkich działań `a[i]*b[i]`; inna nazwa to iloczyn wewnętrzny — `inner_product`).
- Zasadniczo przekształca konwencjonalną notację tablicową lub wektorową na kod, który trzeba by było skrupulatnie napisać samodzielnie (i działa przynajmniej tak samo szybko, jak ona).
- Można ją rozszerzać w razie potrzeby (nie zastosowano w jej implementacji żadnej „magii”).

Wszystko to, i tylko to, oferuje biblioteka *Matrix*. Jeśli potrzebujesz więcej, np. zaawansowanych funkcji tablicowych, macierzy rzadkich, kontroli nad układem pamięci itp., musisz postarać się o nie na własną rękę lub użyć biblioteki, która lepiej odpowiada Twoim potrzebom. Wiele tego rodzaju potrzeb można zaspokoić, budując algorytmy i struktury danych na bazie biblioteki *Matrix*. Biblioteka ta nie wchodzi w skład biblioteki standardowej C++ ISO. Można ją znaleźć na mojej stronie internetowej w pliku o nazwie *Matrix.h*. Jej zawartość przyporządkowano do przestrzeni nazw *Numeric_lib*. Zdecydowaliśmy się na nazwę *Matrix*, ponieważ słowa *vector* i *array* są jeszcze bardziej nadużywane w bibliotekach C++. W implementacji biblioteki *Matrix* zostały zastosowane zaawansowane techniki, których nie będziemy tu opisywać.

24.5.1. Wymiary i dostęp

Rozważmy prosty przykład:

```
#include "Matrix.h"
using namespace Numeric_lib;

void f(int n1, int n2, int n3)
{
    Matrix<double,1> ad1(n1);           // Elementy są typu double; jeden wymiar
    Matrix<int,1> ail(n1);              // Elementy są typu int; jeden wymiar
    ad1(7) = 0;                        // Indeksowanie za pomocą operatora ( ) — styl języka Fortran
    ad1[7] = 8;                        // Operator [ ] też działa — styl języka C

    Matrix<double,2> ad2(n1,n2);        // dwa wymiary
    Matrix<double,3> ad3(n1,n2,n3);     // trzy wymiary
    ad2(3,4) = 7.5;                    // prawdziwe wielowymiarowe indeksowanie
    ad3(3,4,5) = 9.2;
}
```

W definicji macierzy (obiektu klasy *Matrix*) należy określić typ elementów oraz liczbę wymiarów. Oczywiście klasa *Matrix* jest szablonem, a więc typ elementów i liczba wymiarów są parametrami szablonu. Jeśli szablonowi *Matrix* przekaże się tę parę argumentów (np. *Matrix<double,2>*), powstaje typ (klasa), której obiekty można tworzyć poprzez podanie argumentów (np. *Matrix<double,2> ad2(n1,n2)*) określających wymiary. W związku z tym *ad2* to dwuwymiarowa tablica o wymiarach *n1* i *n2*, inaczej mówiąc, macierz *n1* na *n2*. Aby dostać się do elementu jednowymiarowej macierzy, należy zastosować pojedynczy indeks. Aby dostać się do elementu dwuwymiarowej macierzy, należy zastosować dwa indeksy itd.

Podobnie jak w tablicach wbudowanych i wektorach, w macierzach numerowanie indeksów zaczyna się od zera (a nie 1 jak np. w języku Fortran). Innymi słowy elementy obiektu klasy *Matrix* są ponumerowane liczbami z przedziału $[0, \text{maks})$, gdzie *maks* oznacza liczbę elementów.

Jest to proste i łatwe do zrozumienia zagadnienie. Jeśli jednak masz z nim problemy, zwróć do odpowiedniego podręcznika matematyki zamiast podręcznika do programowania. Jedyna „sprytna sztuczka” polega na tym, że można nie określać liczby wymiarów macierzy — domyślnie tworzona jest macierz jednowymiarowa. Należy również zauważyć, że do indeksowania można używać operatorów `[]` (styl języków C i C++) lub `()` (styl języka Fortran).



Dzięki dostępności obu rodzajów indeksowania mamy ułatwione posługiwanie się tablicami wielowymiarowymi. Notacja $[x]$ zawsze pobiera jeden indeks i zwraca odpowiedni wiersz macierzy. Jeśli a jest n -wymiarową macierzą, $a[x]$ jest $n-1$ -wymiarową macierzą. Notacja (x,y,z) pobiera jeden lub więcej indeksów i zwraca odpowiedni element macierzy. Liczba indeksów musi odpowiadać liczbie wymiarów w macierzy.

Zobaczymy, co się dzieje, gdy zrobimy błąd:

```
void f(int n1, int n2, int n3)
{
    Matrix<int,0> ai0;           // Błąd: nie ma macierzy 0-wymiarowych

    Matrix<double,1> ad1(5);
    Matrix<int,1> ai(5);
    Matrix<double,1> ad11(7);

    ad1(7) = 0;                 // Wyjątek Matrix_error (7 jest poza zakresem)
    ad1 = ai;                   // Błąd: różne typy elementów
    ad1 = ad11;                 // Wyjątek Matrix_error (różne wymiary)

    Matrix<double,2> ad2(n1);   // Błąd: nie podano rozmiaru drugiego wymiaru
    ad2(3) = 7.5;               // Błąd: nieprawidłowa liczba indeksów
    ad2(1,2,3) = 7.5;          // Błąd: nieprawidłowa liczba indeksów

    Matrix<double,3> ad3(n1,n2,n3);
    Matrix<double,3> ad33(n1,n2,n3);
    ad3 = ad33;                 // Dobrze: ten sam typ elementów, taka sama liczba wymiarów
}
```

Niezgodności między zadeklarowaną a używaną liczbą wymiarów są wykrywane w czasie kompilacji. Błędy zakresu zostają wykrywane w czasie działania i powodują zgłoszenie wyjątku `Matrix_error`.



Pierwszy wymiar to wiersz, a drugi kolumna, a więc dwuwymiarowe macierze (dzwuwymiarowe tablice) indeksuje się parami (wiersz,kolumna). Można też użyć notacji $[wiersz][kolumna]$, ponieważ indeksowanie macierzy dwuwymiarowej jednym indeksem daje jednowymiarową macierz będącą wierszem. Można to zilustrować graficznie w następujący sposób:

				a[1][2]	
a[0]:	00	01	02	03	
a[1]:	10	11	12	13	a(1,2)
a[2]:	20	21	22	23	

Ta macierz zostanie w pamięci zapisana w kolejności „najpierw wiersz”:

00	01	02	03	10	11	12	13	20	21	22	23
----	----	----	----	----	----	----	----	----	----	----	----

Macierz „zna” swoje wymiary, dzięki czemu można bardzo łatwo odwoływać się do elementów macierzy przekazywanych jako argumenty:

```

void init(Matrix<int,2>& a)           // Inicjalizacja każdego elementu charakterystyczną wartością
{
    for (int i=0; i<a.dim1(); ++i)
        for (int j = 0; j<a.dim2(); ++j)
            a(i,j) = 10*i+j;
}

void print(const Matrix<int,2>& a) // Drukuje elementy wiersz po wierszu
{
    for (int i=0; i<a.dim1(); ++i) {
        for (int j = 0; j<a.dim2(); ++j)
            cout << a(i,j) <<'\\t';
        cout << '\\n';
    }
}

```

Zatem `dim1()` określa liczbę elementów w pierwszym wymiarze, `dim2()` liczbę elementów w drugim wymiarze itd. Typ elementów i liczba wymiarów są częściami typu `Matrix`, przez co nie można napisać funkcji przyjmującej jako argument dowolny obiekt typu `Matrix` (ale można napisać taki szablon):



```

void init(Matrix& a); // Błąd: brakuje typu elementów i liczby wymiarów

```

Należy zauważyć, że biblioteka `Matrix` nie udostępnia operacji na macierzach, jak dodawanie dwóch macierzy czterowymiarowych lub mnożenie macierzy dwuwymiarowej przez jednowymiarową. Eleganckie i wydajne funkcje tego typu są obecnie poza zakresem tej biblioteki. Na bazie biblioteki `Matrix` można zbudować rozmaite inne biblioteki macierzy (zobacz 12. zadanie pracy domowej).

24.5.2. Macierze jednowymiarowe

Co można zrobić z najprostszą macierzą, czyli jednowymiarową?

Można pominąć deklarację liczby wymiarów, ponieważ jeden wymiar jest tworzony domyślnie:

```

Matrix<int,1> a1(8); // a1 jest jednowymiarową macierzą liczb typu int
Matrix<int> a(8);    // Oznacza Matrix<int,1> a(8);

```

Macierze `a` i `a1` są tego samego typu (`Matrix<int,1>`). Można sprawdzić ich rozmiary (liczbę elementów) oraz wymiary (liczbę elementów w wymiarze). W macierzy jednowymiarowej wartości te są takie same.

```

a.size();           // Liczba elementów w macierzy
a.dim1();           // Liczba elementów w pierwszym wymiarze

```

Można sprawdzać elementy zgodnie z ich ułożeniem w pamięci, tzn. za pomocą wskaźnika na pierwszy element:

```

int* p = a.data(); // Wydobywa dane jako wskaźnik na tablicę

```

Jest to przydatne przy przekazywaniu danych macierzy do funkcji w stylu języka C, które pobierają argumenty będące wskaźnikami. Indeksowanie:

```
a(i);    // i-ty element (styl języka Fortran), ale sprawdzany jest zakres
a[i];    // i-ty element (styl języka C), ze sprawdzaniem zakresu
a(1,2);  // Błąd: a jest macierzą jednowymiarową
```



Algorytmy często odwołują się tylko do określonej części macierzy. Taka część nazywa się wycinkiem (`slice()`) — podmacierz lub przedział elementów — i występuje w dwóch wersjach:

```
a.slice(i);    // Elementy od elementu a[i] do końca
a.slice(i,n);  // n elementów od a[i] do a[i+n-1]
```

Indeksów i wycinków można używać zarówno po lewej, jak i prawej stronie przypisania. Odwołują się do elementów swoich macierzy, nie wykonując ich kopii. Na przykład:

```
a.slice(4,4) = a.slice(0,4); // Przypisuje pierwszą połowę a do jej drugiej połowy
```

Jeśli np. `a` ma na początku następujące elementy:

```
{ 1 2 3 4 5 6 7 8 }
```

wynik będzie następujący:

```
{ 1 2 3 4 1 2 3 4 }
```

Należy zauważyć, że najczęściej używa się jako wycinków „początkowych i końcowych elementów” macierzy. Tzn. `a.slice(0,j)` określa przedział `[0,j)`, a `a.slice(j)` przedział `[j,a.size())`. Powyższy przykład można z łatwością napisać tak:

```
a.slice(4) = a.slice(0,4); // Przypisuje pierwszą połowę do drugiej
```

Oznacza to, że notacja faworyzuje często występujące przypadki. Można wpisać takie wartości `i` i `n`, że `a.slice(i,n)` znajdzie się poza przedziałem `a`. Jednak powstała w wyniku tego część będzie odwoływać się tylko do elementów, które rzeczywiście znajdują się w `a`. Na przykład `a.slice(i,a.size())` odwołuje się do przedziału `[i,a.size())`, a `a.slice(a.size())` i `a.slice(i,0)` to puste macierze. Konwencja ta jest bardzo przydatna w wielu algorytmach. Pożyczyliśmy ją z matematyki. Oczywiście `a.slice(i,0)` jest pustą macierzą. Nie napisalibyśmy tego celowo, ale niektóre algorytmy są prostsze, jeśli `a.slice(i,n)`, gdzie `n` ma wartość 0, jest pustą macierzą (a nie błędem, którego trzeba unikać).



Dostępne są typowe (dla obiektów C++) operacje kopiowania kopiujące wszystkie elementy:

```
Matrix<int> a2 = a; // inicjalizowanie kopiujące
a = a2;           // przypisanie kopiujące
```



Na rzecz każdego elementu macierzy można stosować wbudowane operacje:

```
a *= 7; // Skalowanie: a[i]*=7 dla każdego i (także +=, -=, /=, itp.)
a = 7;  // a[i]=7 dla każdego i
```

Jest to dozwolone dla każdego przypisania i każdego złożonego operatora przypisania (=, +=, -=, /=, *=, %=, ^=, &=, |=, >>=, <<=), pod warunkiem że operator ten jest obsługiwany przez dany typ elementów. Na rzecz każdego elementu macierzy można także wywołać funkcję:

```
a.apply(f); // a[i]=f(a[i]) dla każdego elementu a[i]
a.apply(f,7); // a[i]=f(a[i],7) dla każdego elementu a[i]
```

Złożone operatory przypisania i funkcja `apply()` modyfikują elementy macierzy, przekazanej im jako argument. Aby zamiast modyfikować, utworzyć nową macierz, można napisać tak:

```
b = apply(abs,a); // Tworzy nową macierz, w której b(i)=abs(a(i))
```

Użyta w tym kodzie funkcja `abs` to funkcja wartości bezwzględnej z biblioteki standardowej (podrozdział 24.8). Zasadniczo zapis `apply(f,x)` ma się do `x.apply(f)` analogicznie jak + do +=. Na przykład:

```
b = a*7; // b[i] = a[i]*7 dla każdego i
a *= 7; // a[i] = a[i]*7 dla każdego i
y = apply(f,x); // y[i] = f(x[i]) dla każdego i
x.apply(f); // x[i] = f(x[i]) dla każdego i
```

Otrzymujemy `a=b` i `x=y`.

W języku Fortran to drugie `apply` nazywa się funkcją „rozgłoszeniową” (ang. *broadcast function*) i zazwyczaj zapisuje się to jako `f(x)`, a nie `apply(f,x)`. Aby udostępnić to wszystkim funkcjom `f` (a nie tylko kilku wybranym, jak w języku Fortran), musimy tę operację „rozgłaszania” jakoś nazwać — użyjemy nazwy `apply`.

Ponadto, aby uzupełnić dwuargumentową wersję składowej `apply`, `a.apply(f,x)`, dostarczamy:

```
b = apply(f,a,x); // b[i]=f(a[i],x) dla każdego i
```

Na przykład:

```
double scale(double d, double s) { return d*s; }
b = apply(scale,a,7); // b[i] = a[i]*7 dla każdego i
```

Należy zauważyć, że „wolnostojąca” funkcja `apply()` pobiera funkcję, która wytwarza wynik ze swojego argumentu. Następnie `apply()` wykorzystuje te wyniki do zainicjowania powstałej macierzy. Zwykle nie modyfikuje macierzy, na rzecz której zostanie zastosowana. Składowa `apply()` różni się tym, że pobiera funkcję, która modyfikuje swoje argumenty, tzn. modyfikuje elementy macierzy, na rzecz której zostanie zastosowana. Na przykład:

```
void scale_in_place(double& d, double s) { d *= s; }
b.apply(scale_in_place,7); // b[i] *= 7 dla każdego i
```

Dodatkowo udostępniamy kilka najbardziej przydatnych funkcji z tradycyjnych bibliotek obliczeniowych:

```
Matrix<int> a3 = scale_and_add(a,8,a2); // Dodawanie i mnożenie w jednej operacji
int r = dot_product(a3,a); // iloczyn skalarny
```



Funkcję `scale_and_add()` często nazywa się funkcją **FMA** (ang. *fused multiply-add* — jedno-czesne mnożenie i dodawanie). Jej definicja to `result(i)=arg1(i)*arg2+arg3(i)` dla każdego `i` w macierzy. Iloczyn skalarny jest też czasami nazywany iloczynem wewnętrznym i został opisany w punkcie 21.5.3. Jego definicja to `result+=arg1(i)*arg2(i)` dla każdego `i` w macierzy, gdzie `result` ma początkowo wartość 0.

Tablic jednowymiarowych używa się bardzo często. Mogą być reprezentowane jako tablice wbudowane, wektory lub obiekty typu `Matrix`. Ostatnią z wymienionych opcji wykorzystuje się, gdy potrzebne są operacje na macierzach, jak `*`, lub jeśli macierz ma w jakiś sposób współ-pracować z macierzami o większej liczbie wymiarów.



Użyteczność tego rodzaju biblioteki można wyjaśnić słowami: „lepiej odpowiada konwen-cjom matematycznym” lub „pozwala uniknąć pisania pętli, aby wykonać operacje na wszystkich elementach”. W każdym razie kod napisany przy ich użyciu jest dużo krótszy i programista ma mniej okazji do popełnienia błędu przy jego pisaniu. Operacje klasy `Matrix` — takie jak kopiowanie, przypisanie do wszystkich elementów i operacje na wszystkich elementach — pozwalają uniknąć pisania i czytania pętli (oraz zachodzenia w głowę, czy napisana pętla robi dokładnie to, co powinna).

Klasa `Matrix` udostępnia dwa konstruktory do kopiowania danych z wbudowanej tablicy do macierzy. Na przykład:

```
void some_function(double* p, int n)
{
    double val[] = { 1.2, 2.3, 3.4, 4.5 };
    Matrix<double> data(p,n);
    Matrix<double> constants(val);
    // ...
}
```

To się często przydaje, gdy dostarczane są dane w tablicach lub wektorach z części programu, w których nie wykorzystywane są macierze.

Należy zauważyć, że kompilator może wydedukować liczbę elementów zainicjowanej tablicy, dzięki czemu nie trzeba jej podawać w definicji `constants` — wynosi 4. Z drugiej strony kompilator nie zna liczby elementów, jeśli otrzyma tylko wskaźnik, a więc dla `data` trzeba podać zarówno wskaźnik (`p`), jak i liczbę elementów (`n`).

24.5.3. Macierze dwuwymiarowe

Ogólnie w bibliotece `Matrix` przyjęto, że macierze o różnych liczbach wymiarów są w istocie bardzo podobne do siebie, poza wyjątkami dotyczącymi liczby wymiarów. Dzięki temu większość tego, co napisaliśmy o macierzach jednowymiarowych, ma zastosowanie także do macierzy dwuwymiarowych:

```
Matrix<int,2> a(3,4);

int s = a.size(); // liczba elementów
int d1 = a.dim1(); // liczba elementów w wierszu
int d2 = a.dim2(); // liczba elementów w kolumnie
int* p = a.data(); // Wyodrębnia dane jako wskaźnik do tablicy w stylu języka C
```

Można sprawdzić sumę wszystkich elementów, jak również liczbę elementów w każdym wymiarze. Można uzyskać wskaźniki do elementów zgodnie z ich rozłożeniem w pamięci.

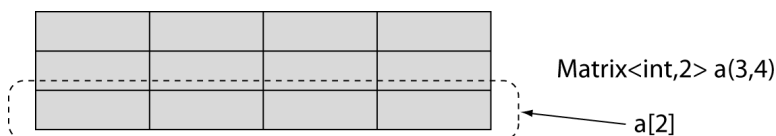
Można stosować następujące rodzaje indeksowania:

`a(i,j)`; // (i,j)-ty element (styl języka Fortran), ale ze sprawdzaniem zakresu

`a[i]`; // i-ty wiersz (styl języka C), ze sprawdzaniem zakresu

`a[i][j]`; // (i,j)-ty element (styl języka C)

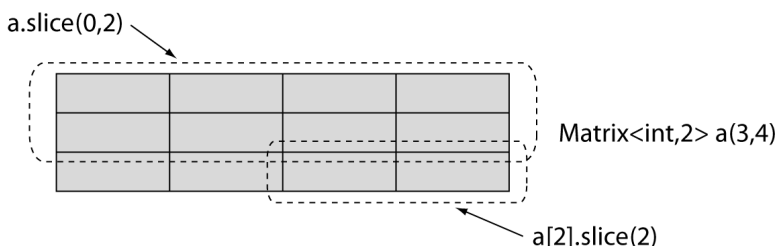
Indeks `[i]` w macierzy dwuwymiarowej zwraca macierz jednowymiarową będącą i-tym wierszem tej macierzy. Oznacza to, że można wyodrębnić wiersze i przekazywać je do operacji i funkcji, które wymagają macierzy jednowymiarowych lub nawet tworzyć tablice wbudowane (`a[i].data()`). Należy zauważyć, że indeksowanie typu `a(i,j)` może być szybsze od `a[i][j]`, chociaż dużo zależy od kompilatora i programu optymalizującego.



Można pobierać wycinki:

`a.slice(i)`; // wiersze od elementu `a[i]` do ostatniego

`a.slice(i,n)`; // wiersze od elementu `a[i]` do `a[i+n-1]`



Należy zauważyć, że wycinek macierzy dwuwymiarowej sam też jest macierzą dwuwymiarową (prawdopodobnie z mniejszą liczbą wierszy).

Operacje są takie same jak dla macierzy jednowymiarowych. Nie jest dla nich ważny układ elementów, ponieważ dotyczą wszystkich elementów w takiej kolejności, w jakiej są rozłożone:

`Matrix<int,2> a2 = a;` // inicjalizowanie kopiujące

`a = a2;` // przypisanie kopiujące

`a *= 7;` // skalowanie (także `+=`, `-=`, `/=` itp.)

`a.apply(f)`; // $a(i,j) = f(a(i,j))$ dla każdego elementu $a(i,j)$

`a.apply(f,7)`; // $a(i,j) = f(a(i,j), 7)$ dla każdego elementu $a(i,j)$

`b=apply(f,a)`; // Tworzy nową macierz, w której $b(i,j) = f(a(i,j))$

`b=apply(f,a,7)`; // Tworzy nową macierz, w której $b(i,j) = f(a(i,j), 7)$

Okazało się, że często przydaje się możliwość zamiany wierszy miejscami, a więc udostępni-
liśmy tę operację:

```
a.swap_rows(1,2); // Zamienia wiersze —  $a[1] \leftrightarrow a[2]$ 
```



Nie ma funkcji `swap_columns()`. Jeśli takiej potrzebujesz, musisz ją napisać samodzielnie (11. zadanie pracy domowej). Z powodu układu typu „wiersz pierwszy” wiersze i kolumny nie są w pełni symetrycznymi jednostkami. Asymetria ta przejawia się także w tym, że `[i]` zwraca wiersz (i nie ma operatora wyboru kolumny). W `(i,j)` pierwszy indeks, `i`, wybiera wiersz. Asymetria ta odzwierciedla też głębokie własności matematyczne.

Wydaje się, że liczba „rzeczy”, które mogą być dwuwymiarowe, jest nieskończona. Poniżej przedstawiamy oczywistego kandydata do zastosowania dwuwymiarowej macierzy:

```
enum Piece { none, pawn, knight, queen, king, bishop, rook };
Matrix<Piece,2> board(8,8); // szachownica

const int white_start_row = 0;
const int black_start_row = 7;

Matrix<Piece> start_row = {rook, knight, bishop, queen, king, bishop, knight, rook};

Matrix<Piece> clear_row(8) ; // 8 elementów o domyślnej wartości
```

W inicjalizacji macierzy `clear_row` został wykorzystany fakt, że `none==0` oraz że elementy są domyślnie inicjalizowane wartością 0. Macierzy `start_row` i `clear_row` można użyć następująco:

```
board[white_start_row] = start_row; // Resetuje białe figury
for (int i = 1; i<7; ++i) board[i] = clear_row; // Czyści środek planszy
board[black_start_row] = start_row; // Resetuje czarne figury
```

Należy zauważyć, że gdy wyodrębniamy wiersz za pomocą `[i]`, uzyskujemy l-wartość (podrozdział 4.3). To znaczy, możemy przypisywać do wyniku `board[i]`.

24.5.4. Wejście i wyjście macierzy

W bibliotece `Matrix` dostępne są bardzo proste narzędzia wejścia i wyjścia dla macierzy jedno- i dwuwymiarowych:

```
Matrix<double> a(4);
cin >> a;
cout << a;
```

Powyższy kod wczyta cztery oddzielane spacjami liczby typu `double`, które muszą znajdować się w klamrach, np.:

```
{ 1.2 3.4 5.6 7.8 }
```

Wynik na wyjściu jest bardzo podobny, dzięki czemu można wczytać, co się wydrukowało.

Operacje wejścia i wyjścia macierzy dwuwymiarowych wczytują i drukują zamknięte w klamrach sekwencje macierzy jednowymiarowych. Na przykład:

```
Matrix<int,2> m(2,2);
cin >> m;
cout << m;
```

Ten kod wczyta następujące dane:

```
{
{ 1 2 }
{ 3 4 }
}
```

Wynik operacji wyjściowej będzie bardzo podobny.

Operatory << i >> macierzy mają przede wszystkim za zadanie ułatwić pisanie prostych programów. W bardziej zaawansowanych programach najczęściej trzeba je zastąpić własnymi. W związku z tym operatory te zostały umieszczone w nagłówku `MatrixIO.h` (a nie `Matrix.h`), aby można było ich nie dołączać przy korzystaniu z macierzy.

24.5.5. Macierze trójwymiarowe

Zasadniczo macierz trójwymiarowa (i wyższego rzędu) jest bardzo podobna do macierzy dwuwymiarowej, tylko ma więcej wymiarów. Rozważmy:

```
Matrix<int,3> a(10,20,30);

a.size();           // liczba elementów
a.dim1();           // liczba elementów w wymiarze 1
a.dim2();           // liczba elementów w wymiarze 2
a.dim3();           // liczba elementów w wymiarze 3
int* p = a.data();  // wydobywa dane jako wskaźnik do tablicy w stylu języka C
a(i,j,k);           // (i,j,k)-ty element (styl języka Fortran), ale ze sprawdzaniem zakresu
a[i];               // i-ty wiersz (styl języka C), ze sprawdzaniem zakresu
a[i][j][k];         // (i,j,k)-ty element (styl języka C)
a.slice(i);          // wiersze od i-tego do ostatniego
a.slice(i,j);        // wiersze od i-tego do j-tego
Matrix<int,3> a2 = a; // inicjalizowanie kopiujące
a = a2;              // przypisanie kopiujące
a *= 7;              // skalowanie (oraz +=, -=, /= itd.)
a.apply(f);          // a(i,j,k)=f(a(i,j,k)) dla każdego elementu a(i,j,k)
a.apply(f,7);        // a(i,j,k)=f(a(i,j,k),7) dla każdego elementu a(i,j,k)
b=apply(f,a);        // tworzy nową macierz, w której b(i,j,k)=f(a(i,j,k))
b=apply(f,a,7);      // tworzy nową macierz, w której b(i,j,k)=f(a(i,j,k),7)
a.swap_rows(7,9);    // zamienia wiersze a[7] <-> a[9]
```

Jeśli rozumiesz macierze dwuwymiarowe, to zrozumiesz też i trójwymiarowe. Na przykład w tym przypadku `a` jest macierzą trójwymiarową, `a[i]` dwuwymiarową (pod warunkiem że `i` mieści się w zakresie), `a[i][j]` jednowymiarową (pod warunkiem że `j` mieści się w zakresie), a `a[i][j][k]` jest elementem typu `int` (pod warunkiem że `k` mieści się w zakresie).

Zwykle świat postrzegamy w trzech wymiarach, co oczywiście podsuwa wiele pomysłów na wykorzystanie macierzy trójwymiarowych w modelowaniu (np. symulacje fizyczne przy użyciu kartezjańskiego układu współrzędnych):

```
int grid_nx; // rozdzielczość siatki; ustawiana na początku
int grid_ny;
int grid_nz;
Matrix<double,3> cube(grid_nx, grid_ny, grid_nz);
```

Jeśli dodamy czas jako czwarty wymiar, uzyskamy przestrzeń czterowymiarową, do przedstawienia której będzie potrzebna nam macierz czterowymiarowa. I tak dalej.

Bardziej zaawansowaną wersję typu `Matrix`, obsługującą ogólne macierze N -wymiarowe, opisałem w rozdziale 29. książki *Język C++. Kompendium wiedzy*.

24.6. Przykład — rozwiązywanie równań liniowych



Kod obliczeń komputerowych jest sensowny tylko dla tych, którzy rozumieją wyrażone w nim pojęcia matematyczne. Pozostałym osobom najczęściej wydaje się bzdurny. Przedstawiony w tym podrozdziale przykład powinien być banalny dla osób znających podstawy algebry liniowej. Jeśli ich nie znasz, potraktuj to jako przykład transkrypcji rozwiązania opisanego w tekście książki na kod z minimalną korzyścią.

Wybraliśmy względnie realistyczny i ważny sposób wykorzystania macierzy do zademonstrowania. Będziemy rozwiązywać układy równań liniowych o następującej postaci:

$$\begin{aligned} a_{1,1}x_1 + \cdots + a_{1,n}x_n &= b_1 \\ &\vdots \\ a_{n,1}x_1 + \cdots + a_{n,n}x_n &= b_n \end{aligned}$$

W tym zapisie litery x oznaczają n -te niewiadome, natomiast a i b są znanymi stałymi. Dla uproszczenia przyjmujemy założenie, że stałe te są wartościami zmiennoprzecinkowymi.

Naszym celem jest znaleźć takie wartości niewiadomych, które jednocześnie rozwiązują n równań. Równania te można zwięźle przedstawić w postaci macierzy i dwóch wektorów:

$$Ax = b$$

Tutaj A jest kwadratową macierzą o wymiarach n na n zdefiniowaną przez współczynniki:

$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix}$$

Wektory x i b zawierają odpowiednio niewiadome i stałe:

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad i \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Układ ten może mieć zero, jedno lub nieskończoną liczbę rozwiązań w zależności od współczynników macierzy A i wektora b . Układy równań liniowych można rozwiązywać na wiele sposobów. My zastostujemy klasyczną technikę zwaną metodą eliminacji Gaussa (zobacz Freeman i Philips, *Parallel Numerical Algorithms*, Stewart, *Matrix Algorithms, Volume I* oraz Wood, *Introduction to Numerical Analysis*). Najpierw dokonamy transformacji A i b , aby zamienić A na macierz górnorójkątną, czyli taką, w której wszystkie współczynniki znajdujące się pod przekątną są równe zero. Innymi słowy, sprowadzimy układ do następującej postaci:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ 0 & \ddots & \vdots \\ 0 & 0 & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Łatwo to zrobić. Zero w elemencie $a(i,j)$ uzyskuje się poprzez pomnożenie równania wiersza i przez stałą, tak aby element ten był równy innemu elementowi w kolumnie j , np. $a(k,j)$. Następnie należy odjąć od siebie te dwa równania, w wyniku czego $a(i,j) = 0$, a pozostałe wartości w wierszu i zmieniają wartości w odpowiedni sposób.

Jeśli da się sprawić, aby wszystkie współczynniki na przekątnej były różne od zera, układ ma unikatowe rozwiązanie, które można znaleźć metodą podstawiania wstecz. Ostatnie równanie łatwo rozwiązać:

$$a_{n,n} x_n = b_n$$

Oczywiście $x[n]$ to $b[n]/a(n,n)$. Następnie usuń wiersz n z układu i przejdź do szukania wartości $x[n-1]$ itd., aż do obliczenia wartości $x[1]$.

Dla każdego n dzielimy przez $a(n,n)$, aby wartości na przekątnej były różne od zera. Jeśli tak się nie stanie, metoda podstawiania wstecz nie powiodła się, co oznacza, że układ ma zero lub nieskończoną liczbę rozwiązań.

24.6.1. Klasyczna eliminacja Gaussa

Spójrzmy teraz na odpowiadający temu kod w języku C++. Najpierw uprościmy sobie notację, nadając konwencjonalne nazwy dwóm rodzajom macierzy, których będziemy używać:

```
typedef Numeric_lib::Matrix<double, 2> Matrix;
typedef Numeric_lib::Matrix<double, 1> Vector;
```

Następnie zapiszemy w postaci kodu obliczenia:

```
Vector classical_gaussian_elimination(Matrix A, Vector b)
{
    classical_elimination(A, b);
    return back_substitution(A, b);
}
```

Wykonaliśmy kopie naszych danych wejściowych A i b (przy użyciu wywołania przez wartość), wywołaliśmy funkcję rozwiązującą układ, a na końcu obliczyliśmy wynik do zwrócenia przez podstawianie wstecz. Najważniejsze, że nasz podział problemu i notacja, za pomocą której wyraziliśmy rozwiązanie, są wzięte bezpośrednio z tekstu książki. Aby dokończyć zadanie,

musimy zaimplementować funkcje `classical_elimination()` i `back_substitution()`. Znowu rozwiązanie jest w książce:

```
void classical_elimination(Matrix& A, Vector& b)
{
    const Index n = A.dim1();

    // Przechodzi od pierwszej kolumny do przedostatniej
    // i nadaje wartość 0 wszystkim elementom pod przekątną:
    for (Index j = 0; j<n-1; ++j) {
        const double pivot = A(j, j);
        if (pivot == 0) throw Elim_failure(j);

        // Nadaje wartość 0 każdemu elementowi znajdującemu się pod przekątną i-tego wiersza:
        for (Index i = j+1; i<n; ++i) {
            const double mult = A(i, j) / pivot;
            A[i].slice(j) = scale_and_add(A[j].slice(j), -mult, A[i].slice(j));
            b(i) -= mult * b(j); // Odpowiednio zmienia b
        }
    }
}
```

Element `pivot` to leżący na przekątnej element wiersza, którym się w danej chwili zajmujemy. Musi mieć wartość różną od zera, ponieważ będziemy przez niego dzielić. Jeśli ma wartość zero, poddajemy się i zgłaszamy wyjątek:

```
Vector back_substitution(const Matrix& A, const Vector& b)
{
    const Index n = A.dim1();
    Vector x(n);

    for (Index i = n - 1; i >= 0; --i) {
        double s = b(i)-dot_product(A[i].slice(i+1),x.slice(i+1));

        if (double m = A(i, i))
            x(i) = s / m;
        else
            throw Back_subst_failure(i);
    }

    return x;
}
```

24.6.2. Wybór elementu centralnego

Możemy uniknąć problemu dzielenia przez zero i przy okazji uzyskać solidniejsze rozwiązanie poprzez takie posortowanie wierszy, aby zera i małe wartości trzymać z dala od przekątnej. Piszac „solidniejsze”, mamy na myśli mniej podatne na błędy zaokrąglania. Jednak wartości zmieniają się cały czas, gdy umieszczamy zera pod przekątną, przez co musimy zmieniać kolejność wierszy także w czasie pracy (tzn. nie możemy po prostu zmienić kolejności i zastosować klasycznego algorytmu):

```

void elim_with_partial_pivot(Matrix& A, Vector& b)
{
    const Index n = A.dim1();

    for (Index j = 0; j < n; ++j) {
        Index pivot_row = j;

        // Szuka odpowiedniego punktu centralnego (pivot):
        for (Index k = j + 1; k < n; ++k)
            if (abs(A(k, j)) > abs(A(pivot_row, j))) pivot_row = k;

        // Zamienia wiersze miejscami, jeśli znajdzie lepszy punkt centralny:
        if (pivot_row != j) {
            A.swap_rows(j, pivot_row);
            std::swap(b(j), b(pivot_row));
        }

        // Eliminacja:
        for (Index i = j + 1; i < n; ++i) {
            const double pivot = A(j, j);
            if (pivot==0) error("Nie da się rozwiązać: pivot==0");
            const double mult = A(i, j)/pivot;
            A[i].slice(j) = scale_and_add(A[j].slice(j), -mult, A[i].slice(j));
            b(i) -= mult * b(j);
        }
    }
}

```

Użyliśmy funkcji `swap_rows()` i `scale_and_add()`, aby kod był bardziej konwencjonalny i aby uniknąć pisania pętli.

24.6.3. Testowanie

Oczywiście musimy przetestować nasz kod. Na szczęście istnieje na to prosty sposób:

```

void solve_random_system(Index n)
{
    Matrix A = random_matrix(n); // Zobacz podrozdział 24.7
    Vector b = random_vector(n);

    cout << "A = " << A << '\n';
    cout << "b = " << b << '\n';

    try {
        Vector x = classical_gaussian_elimination(A, b);
        cout << "Rozwiązaniem klasycznej eliminacji jest x = " << x << '\n';
        Vector v = A * x;
        cout << " A * x = " << v << '\n';
    }
    catch(const exception& e) {
        cerr << e.what() << '\n';
    }
}

```

Do klauzuli catch możemy dojść na trzy sposoby:

- Z powodu błędu w kodzie (ale ponieważ jesteśmy optymistami, nie wydaje nam się, żeby jakieś tam były).
- Z powodu danych wejściowych, które potrafią ominąć klasyczną eliminację (w wielu przypadkach lepiej byłoby użyć `elim_with_partial_pivot()`).
- Z powodu błędów zaokrąglania.

Nasz test nie jest jednak tak realistyczny, jakbyśmy chcieli, ponieważ realne losowe macierze rzadko sprawiają problemy klasycznej eliminacji.

Aby zweryfikować nasze rozwiązanie, drukujemy $A \cdot x$, czego wynik powinien być równy b (lub być mu bardzo bliski w naszym przypadku, ponieważ bierzemy pod uwagę błędy zaokrąglania). Możliwość wystąpienia błędów zaokrąglania spowodowała, że nie napisaliśmy po prostu:

```
if (A*x!=b) error("Podstawianie nie powiodło się.");
```

Ponieważ liczby zmiennoprzecinkowe są tylko przybliżeniem wartości liczb rzeczywistych, musimy zaakceptować przybliżone poprawne wartości. Ogólnie najlepiej unikać używania operatorów `==` i `!=` na rzecz wyników obliczeń przy użyciu liczb zmiennoprzecinkowych — liczba zmiennoprzecinkowa jest z zasady przybliżeniem.

W bibliotece `Matrix` nie ma operacji mnożenia macierzy przez wektor, a więc musieliśmy ją napisać samodzielnie:

```
Vector operator*(const Matrix& m, const Vector& u)
{
    const Index n = m.dim1();
    Vector v(n);
    for (Index i = 0; i < n; ++i) v(i) = dot_product(m[i], u);
    return v;
}
```

Znowu prosta operacja macierzowa wykonała za nas większość pracy. Operacje wyjściowe, których użyliśmy, pochodzą z nagłówka `Matrix10.h` — ich opis znajduje się w punkcie 24.5.4. Funkcje `random_matrix()` i `random_vector()` to proste przykłady użycia liczb losowych (podrozdział 24.7) — ich implementację pozostawiamy jako pracę domową. `Index` to alias typu (punkt A.15) typu indeksu używanego przez bibliotekę `Matrix`. Wnieśliśmy go do naszego zasięgu za pomocą deklaracji `using`.

```
using Numeric_lib::Index;
```

24.7. Liczby losowe

Jeśli spyta się kogoś o podanie dowolnej liczby, większość odpowiada 7 lub 17, a więc ustalono, że są to „najbardziej losowe” ze wszystkich liczb. W zasadzie nikt nie podaje liczby 0. Jest ona tak ładnie zaokrąglona, że nikomu nie wydaje się, aby była losowa, dlatego zero można uważać za „najmniej losową liczbę”. Z matematycznego punktu widzenia wszystkie te rozważania to kompletne bzdury. Losowość nie odnosi się do żadnej konkretnej liczby. To, co zazwyczaj jest potrzebne i nazywa się liczbami losowymi, to szereg liczb, w którym nie da się łatwo przewidzieć następnego elementu, znając poprzedni.



Tego rodzaju liczb najczęściej używa się w testowaniu (stanowią jeden ze sposobów na wygenerowanie dużej liczby przypadków testowych), grach (jeden ze sposobów na zapewnienie, że kolejne uruchomienie gry będzie się różniło od poprzedniego) oraz symulacjach (można sprawić, aby symulowana jednostka zachowywała się w „losowy” sposób w granicach swoich parametrów).

Będąc praktycznym narzędziem i matematycznym problemem, sztuczne liczby losowe muszą być bardzo zaawansowane, aby były równie użyteczne jak realne liczby losowe. W tym podrozdziale opiszemy tylko podstawowe wiadomości, które wystarczą do przeprowadzenia prostych testów i symulacji. W nagłówku `<random>` biblioteki standardowej znajduje się zestaw zaawansowanych narzędzi do generowania liczb losowych dla różnych rozkładów matematycznych. Narzędzia te są oparte na dwóch podstawowych filarach:



- **Generatory** (generatory liczb losowych — ang. *random number engines*) — są to obiekty funkcyjne, które generują sekwencję liczb całkowitych o rozkładzie jednostajnym.
- **Rozkłady** — są to obiekty funkcyjne, które generują sekwencje wartości na podstawie wzoru matematycznego, pobierając na wejściu sekwencję wartości wygenerowane przez generator.

Przeanalizujemy na przykład użytą w punkcie 24.6.3 funkcję `random_vector()`. Wywołanie `random_vector(n)` zwraca macierz `Matrix<double,1>` z `n` elementami typu `double` o losowych wartościach z przedziału `[0,n]`:

```
Vector random_vector(Index n)
{
    Vector v(n);
    default_random_engine ran{};           // Generuje liczby całkowite
    uniform_real_distribution<> ureal{0,max}; // Mapuje liczby całkowite na double
                                           // w [0:max)

    for (Index i = 0; i < n; ++i)
        v(i) = ureal(ran);

    return v;
}
```

Domyślny generator (`default_random_engine`) jest prosty, efektywny i wystarczająco dobry do okazjonalnego użycia. Natomiast do zastosowań profesjonalnych w bibliotece standardowej przygotowano generatory zapewniające lepsze właściwości losowe i charakteryzujące się różną złożonością obliczeniową, np. `linear_congurential_engine`, `mersenne_twister_engine` i `random_device`. Jeśli zechcesz ich użyć, czyli będziesz potrzebować czegoś lepszego niż `default_random_engine`, musisz najpierw trochę poczytać. Aby dowiedzieć się, jakiej jakości jest generator liczb losowych w Twoim systemie, wykonaj ćwiczenie 10.

Oto definicje dwóch generatorów liczb losowych z pliku `std_lib_facilities.h`:

```
int randint(int min, int max)
{
    static default_random_engine ran;
    return uniform_int_distribution<>{min,max}(ran);
}

int randint(int max)
{
    return randint(0,max);
}
```

Te proste funkcje są bardzo pomocne, ale w ramach ćwiczeń wygenerujemy rozkład normalny:

```
auto gen = bind(normal_distribution<double>{15,4.0},
default_random_engine{ } );
```

Funkcja `bind()` z nagłówka `<functional>` biblioteki standardowej tworzy obiekt funkcyjny, który wywołuje swój pierwszy argument i przekazuje mu jako argument swój drugi argument. W tym przypadku więc funkcja `gen()` zwraca wartości w rozkładzie standardowym o wartości średniej 15 i wariancji 4.0 przy użyciu generatora `default_random_engine`. Oto przykład jej użycia:

```
vector<int> hist(2*15);
for (int i = 0; i < 500; ++i)           // Generuje histogram 500 wartości
    ++hist[int(round(gen()))]);
for (int i = 0; i != hist.size(); ++i) { // Drukuje histogram
    cout << i << '\t';
    for (int j = 0; j != hist[i]; ++j)
        cout << '*';
    cout << '\n';
}
```

Otrzymujemy:

```
0
1
2
3 **
4 *
5 *****
6 ****
7 ****
8 *****
9 *****
10 *****
11 *****
12 *****
13 *****
14 *****
15 *****
16 *****
17 *****
18 *****
19 *****
20 *****
21 *****
22 *****
23 *****
24 *****
25 *
26 *
27
28
29
```


Rozkład normalny jest powszechnie znany i często nazywa się go też rozkładem Gaussa lub (z oczywistego powodu) krzywą dzwonową. Inne rozkłady to `bernoulli_distribution`, `exponential_distribution` i `chi_squared_distribution`. Ich opis znajduje się w książce *Język C++*. *Kompendium wiedzy*. Rozkłady całkowitoliczbowe zwracają wartości w przedziale zamkniętym `[a:b]`, podczas gdy rozkłady rzeczywiste (zmiennoprzecinkowe) — w przedziale jednostronnie otwartym `[a:b)`.

Domyślnie generator (może z wyjątkiem `random_device`) za każdym razem daje ten sam wynik. To bardzo przydatna cecha na początku debugowania. Aby otrzymać inne wartości z generatora, musimy go zainicjalizować przy użyciu innych wartości. Inicjalizatory takie nazywa się ziarnem (ang. *seed*). Na przykład:

```
auto gen1 = bind(uniform_int_distribution<>{0,9},
default_random_engine{} );
auto gen2 = bind(uniform_int_distribution<>{0,9},
default_random_engine{10});
auto gen3 = bind(uniform_int_distribution<>{0,9},
default_random_engine{5});
```

Aby uzyskać nieprzewidywalną sekwencję liczb, jako ziarno programiści często wykorzystują na przykład czas (z dokładnością nawet do nanosekundy — punkt 26.6.1).

24.8. Standardowe funkcje matematyczne

Standardowe funkcje matematyczne (`cos`, `sin`, `log` itp.) są dostępne w bibliotece standardowej. Ich deklaracje znajdują się w nagłówku `<cmath>`.

Standardowe funkcje matematyczne

<code>abs(x)</code>	wartość bezwzględna
<code>ceil(x)</code>	najmniejsza liczba całkowita $\geq x$
<code>floor(x)</code>	największa liczba całkowita $\leq x$
<code>sqrt(x)</code>	pierwiastek kwadratowy, x musi mieć wartość większą od 0
<code>cos(x)</code>	cosinus
<code>sin(x)</code>	sinus
<code>tan(x)</code>	tangens
<code>acos(x)</code>	arcus cosinus, wynik jest nieujemny
<code>asin(x)</code>	arcus sinus, zwraca wynik najbliższy 0
<code>atan(x)</code>	arcus tangens
<code>sinh(x)</code>	sinus hiperboliczny
<code>cosh(x)</code>	cosinus hiperboliczny
<code>tanh(x)</code>	tangens hiperboliczny
<code>exp(x)</code>	funkcja wykładnicza o podstawie e
<code>log(x)</code>	logarytm naturalny o podstawie e , x musi być dodatni
<code>log10(x)</code>	logarytm dziesiętny (o podstawie 10)

Standardowe funkcje matematyczne przyjmują argumenty typów float, double, long double oraz complex (podrozdział 24.9). Jeśli wykonujesz obliczenia przy użyciu liczb zmiennoprzecinkowych, funkcje te pewnie Ci się przydadzą. Więcej szczegółów można znaleźć w dokumentacji, która jest ogólnodostępna. Dobrym miejscem do rozpoczęcia poszukiwań jest dokumentacja internetowa.



Jeśli standardowa funkcja matematyczna nie może zwrócić matematycznie poprawnego wyniku, odpowiednio ustawia wartość zmiennej errno. Na przykład:

```
errno = 0;
double s2 = sqrt(-1);
if (errno) cerr << "Coś, gdzieś się czemuś nie udało.";
if (errno == EDOM) // Błąd dziedziny
    cerr << "Funkcja sqrt() nie przyjmuje ujemnych argumentów.";
pow(very_large, 2); // Zły pomysł
if (errno == ERANGE) // Błąd zakresu
    cerr << "Wartość pow(" << very_large << ", 2) za duża dla typu double.";
```

Jeśli wykonuje się poważne obliczenia matematyczne, należy po uzyskaniu wyniku zawsze sprawdzić, czy zmienna errno ma wartość 0. Jeśli nie, znaczy, że coś poszło nie tak. Sprawdź w swoim podręczniku lub dokumentacji internetowej, które funkcje matematyczne mogą ustawiać wartość zmiennej errno i jakich wartości wówczas używają.

Jak pokazaliśmy w przykładzie, wartość inna niż 0 w zmiennej errno oznacza, że coś poszło nie tak. Często funkcje nienależące do biblioteki standardowej ustawiają wartość tej zmiennej w przypadku błędu, a więc aby dowiedzieć się, co dokładnie się stało, należy dokładniej się tej wartości przyjrzeć. Jeśli sprawdzi się ją bezpośrednio po zakończeniu działania funkcji z biblioteki standardowej i upewni się, że ma wartość 0 przed jej wywołaniem, można na niej polegać — tak zrobiliśmy w przypadkach EDOM i ERANGE w powyższym przykładzie. EDOM oznacza błąd dziedziny (tzn. problem z wynikiem). ERANGE oznacza błąd zakresu (tzn. problem z argumentami).



Obsługa błędów przy użyciu zmiennej errno jest dość prymitywną techniką. Jej powstanie wiąże się z pierwszym (z 1975 roku) zestawem funkcji matematycznych w języku C.

24.9. Liczby zespolone

Liczby zespolone są powszechnie wykorzystywane w obliczeniach naukowych i inżynierskich. Zakładamy, że jeśli ktoś ich potrzebuje, musi znać ich matematyczne własności, a więc tutaj opiszemy tylko, jak są reprezentowane w bibliotece standardowej w standardzie ISO języka C++. Definicja liczb zespolonych i związane z nimi standardowe funkcje matematyczne znajdują się w nagłówku <complex>:

```
template<class Scalar> class complex {
    // Liczba zespolona to para wartości skalarnych
    Scalar re, im;
public:
    constexpr complex(const Scalar & r, const Scalar & i) :re(r), im(i) { }
    constexpr complex(const Scalar & r) :re(r), im(Scalar ()) { }
    complex() :re(Scalar ()), im(Scalar ()) { }

    constexpr Scalar real() { return re; } // część rzeczywista
```

```
constexpr Scalar imag() { return im; } // część urojona
// operacje: = += -= *= /=
};
```

Szablon `complex` z biblioteki standardowej na pewno pozwala na stosowanie typów skalar-nych `float`, `double` i `long double`. Poza składowymi klasy `complex` i standardowymi funkcjami matematycznymi (podrozdział 24.8), w nagłówku `<complex>` znajduje się trochę innych przy-datnych operacji:

Operatory liczb zespolonych	
<code>z1+z2</code>	dodawanie
<code>z1-z2</code>	odejmowanie
<code>z1*z2</code>	mnożenie
<code>z1/z2</code>	dzielenie
<code>z1==z2</code>	równość
<code>z1!=z2</code>	nierówność
<code>norm(z)</code>	kwadrat <code>abs(z)</code>
<code>conj(z)</code>	sprzężenie: jeśli <code>z</code> jest <code>(re,im)</code> , to <code>conj(z)</code> jest <code>(re,-im)</code>
<code>polar(rho, theta)</code>	tworzy liczbę zespoloną przy użyciu współrzędnych biegunowych <code>(rho,theta)</code>
<code>real(z)</code>	część rzeczywista
<code>imag(z)</code>	część urojona
<code>abs(z)</code>	inaczej <code>Rho</code>
<code>arg(z)</code>	inaczej <code>theta</code>
<code>out << z</code>	operator wyjścia
<code>in >> z</code>	operator wejścia

Uwaga: klasa `complex` nie udostępnia operatorów `<`, `>` i `%`. Szablonu `complex<T>` używa się tak samo jak typów wbudowanych, np. `double`. Na przykład:

```
Using cmplx = complex<double>; // Czasami zapis complex<double>
// jest zbyt rozwlekły
void f(dcmplx z, vector<dcmplx>& vc)
{
    cmplx z2 = pow(z,2);
    cmplx z3 = z2*9.3+vc[3];
    cmplx sum = accumulate(vc.begin(), vc.end(), cmplx());
    // ...
}
```

Należy pamiętać, że nie wszystkie operacje, do których się przyzwyczailiśmy w typach `int` czy `double`, są dostępne w `complex`. Na przykład:

```
if (z2<z3) // Błąd: nie ma operatora < dla liczb zespolonych
```

Należy zauważyć, że reprezentacja (układ) liczb zespolonych z biblioteki standardowej C++ jest zgodna z odpowiadającymi im typami w językach C i Fortran.

24.10. Źródła

Zasadniczo poruszone w tym rozdziale tematy, jak błędy zaokrąglania, działania na macierzach i arytmetyka liczb zespolonych, bez odpowiedniego kontekstu są bez sensu. Opisałiśmy po prostu niektóre techniki dostępne w języku C++, za pomocą których osoby posiadające odpowiednią wiedzę matematyczną mogą wykonywać obliczenia.

Dla tych, którzy już pozapominali te rzeczy lub są ciekawi nowej wiedzy, przedstawiamy listę zalecanych źródeł:

Archiwum MacTutor History of Mathematics: <http://www-gap.dcs.st-and.ac.uk/~history>.

- Świetne źródło dla wszystkich, którzy lubią matematykę lub jej po prostu potrzebują.
- Świetne źródło dla tych, którzy chcą zobaczyć ludzką twarz matematyki. Np. jak brzmi nazwisko jedynego słynnego matematyka, który zdobył medal olimpijski?
 - Sławni matematycy — biografie, opis dokonań.
 - Ciekawostki.
- Słynne krzywe.
- Słynne problemy.
- Dziedziny matematyki:
 - Algebra,
 - Analiza,
 - Liczby i teoria liczb,
 - Geometria i topologia,
 - Fizyka matematyczna,
 - Astronomia matematyczna,
 - Historia matematyki,
 - ...

Freeman T. L., Phillips Chris, *Parallel Numerical Algorithms*, Prentice Hall, 1992.

Gullberg Jan, *Mathematics — From the Birth of Numbers*, W. W. Norton, 1996. Jedna z najprzystępniejszych przydatnych książek o podstawach matematyki. Rzadko spotykana publikacja, którą można przeczytać dla przyjemności oraz wykorzystywać do wyszukiwania interesujących tematów, np. informacji o macierzach.

Knuth Donald E., *Sztuka programowania, tom 2: Algorytmy seminumeryczne*, Wydawnictwa Naukowo-Techniczne, 2003.

Stewart G. W., *Matrix Algorithms, Volume I: Basic Decompositions*, SIAM, 1998.

Wood Alistair, *Introduction to Numerical Analysis*, Addison-Wesley, 1999.

Ćwiczenia

1. Wydrukuj rozmiary typów `char`, `short`, `int`, `long`, `float`, `double`, `int*` oraz `double*` (użyj funkcji `sizeof()`, nie nagłówka `<limits>`).
2. Wydrukuj rozmiary według funkcji `sizeof()` macierzy `Matrix<int> a(10)`, `Matrix<int> b(100)`, `Matrix<double> c(10)`, `Matrix<int,2> d(10,10)` oraz `Matrix<int,3> e(10,10,10)`.
3. Wydrukuj liczbę elementów każdej macierzy z ćwiczenia 2.
4. Napisz program pobierający na wejściu (`cin`) liczby typu `int` i wysyłający na wyjście ich pierwiastki kwadratowe (`sqrt()`) lub komunikat: „Nie da się obliczyć pierwiastka.”, jeśli wywołanie `sqrt(x)` jest nieprawidłowe dla danego `x` (tzn. należy sprawdzić wartość zwrótną funkcji `sqrt()`).
5. Wczytaj na wejściu dziesięć wartości zmiennoprzecinkowych i zapisz je w macierzy `Matrix<double>`. Macierze nie mają funkcji `push_back()`, a więc musisz obsłużyć możliwość wystąpienia próby zapisu nieprawidłowej liczby tych wartości. Wydrukuj zawartość swojej macierzy.
6. Opracuj tabliczkę mnożenia dla $[0,n] \times [0,m]$ i wynik przedstaw w postaci macierzy dwuwymiarowej. Wartości `n` i `m` pobierz ze strumienia `cin` i wydrukuj ładnie sformatowaną tabelę (przyjmij założenie, że wartość `m` jest na tyle mała, że wyniki zmieszczą się w wierszu).
7. Wczytaj dziesięć obiektów typu `complex<double>` ze strumienia `cin` (tak, strumień `cin` ma operator `>>` dla typu `complex`) i wstaw je do macierzy. Oblicz i wydrukuj na wyjściu sumę tych liczb zespolonych.
8. Wczytaj sześć liczb typu `int` do macierzy `Matrix<int,2> m(2,3)` i wydrukuj je.

Powtórzenie

1. Kto korzysta z bibliotek do wykonywania obliczeń?
2. Co to jest precyzja?
3. Co to jest przepełnienie?
4. Jaki najczęściej rozmiar ma typ `double`, a jaki `int`?
5. Jak wykrywa się przepełnienia?
6. Jak można znaleźć ograniczenia typów liczbowych, np. największą wartość typu `int`?
7. Co to jest tablica, wiersz i kolumna?
8. Co to jest wielowymiarowa tablica w stylu języka C?
9. Jakie narzędzia powinien posiadać język programowania (np. biblioteki) wspierający obliczenia przy użyciu macierzy?
10. Co to jest wymiar macierzy?
11. Ile wymiarów może mieć macierz (w teorii i w matematyce)?
12. Co to jest wycinek?
13. Co to jest operacja rozgłaszania? Wymień kilka.
14. Jaka jest różnica między indeksowaniem w stylu języka Fortran a w stylu języka C?
15. Jak wykonuje się operacje na rzecz wszystkich elementów macierzy? Podaj przykłady.

16. Co to jest operacja łączona?
17. Podaj definicję **iloczynu skalarnego**.
18. Co to jest algebra liniowa?
19. Co to jest metoda eliminacji Gaussa?
20. Co to jest punkt centralny (ang. *pivot*) — w algebrze liniowej i w życiu codziennym?
21. Jakie cechy ma liczba losowa?
22. Co to jest jednolity rozkład?
23. Gdzie znajdują się standardowe funkcje matematyczne? Dla jakich typów argumentów zostały zdefiniowane?
24. Co to jest urojona część liczby zespolonej?
25. Ile wynosi pierwiastek kwadratowy z liczby -1 ?

Terminologia

C	liczba zespolona	tablica
errno	Matrix	urojony
Fortran	operacja łączona	wielowymiarowy
iloczyn skalarny	operacja na elementach	wiersz
indeksowanie	rozmiar	wycinanie
jednolity rozkład	rzeczywisty	wymiar
kolumna	sizeof	
liczba losowa	skalowanie	

Praca domowa

1. Argumenty funkcyjne w wywołaniach `a.apply(f)` i `a.apply(f,a)` są różne. Napisz dla każdego funkcję `triple()` potrającą elementy tablicy `{ 1 2 3 4 5 }`. Zdefiniuj jedną funkcję `triple()`, której można użyć zarówno w `a.apply(triple)`, jak i `a.apply(triple,a)`. Wyjaśnij, czemu może być złym pomysłem napisanie wszystkich funkcji w taki sposób, aby mogły być tak wykorzystane przez funkcję `apply()`.
2. Jeszcze raz wykonaj zadanie 1., ale zamiast funkcji utwórz obiekty funkcyjne. Wskazówka: w nagłówku `Matrix.h` można znaleźć przykłady.
3. Dla zaawansowanych (nie da się tego zadania wykonać tylko przy użyciu narzędzi opisanych w tej książce): napisz funkcję `apply(f,e)`, która może pobierać funkcje `void(T&)`, `T(const T&)` oraz ich odpowiedniki w postaci obiektów funkcyjnych. Wskazówka: `Boost::bind`.
4. Uruchom program rozwiązujący układy równań liniowych metodą eliminacji Gaussa. Tzn. dokończ go, doprowadź do stanu, w którym da się skompilować, oraz przetestuj na prostym przykładzie.
5. Wypróbuj program eliminacji Gaussa na `A={{0 1}{1 0}}` i `b={{5 6}}` i zobacz jego niepowodzenie. Następnie spróbuj użyć funkcji `elim_with_partial_pivot()`.
6. Zastąp w programie eliminacji Gaussa operacje wektorowe `dot_product()` i `scale_and_add()` pętlami. Przetestuj program i dodaj komentarze, które ułatwią zrozumienie jego kodu.

7. Napisz program eliminacji Gaussa bez używania biblioteki `Matrix`, tzn. w zamian użyj tablic wbudowanych lub wektorów.
8. Opracuj animację działania metody eliminacji Gaussa.
9. Napisz ponownie niebędące składowymi funkcje `apply()`, aby zwracały macierz elementów typu, jaki zwraca stosowana przez nie funkcja. Tzn. funkcja `apply(f,a)` powinna zwracać `Matrix<R>`, gdzie `R` to typ zwrotny funkcji `f`. Ostrzeżenie: aby rozwiązać to zadanie, trzeba posiadać informacje o szablonach, których nie ma w tej książce.
10. Jak losowy jest Twój rozkład `default_random_engine`? Napisz program pobierający na wejściu dwie liczby całkowite `n` i `d` i wywołujący funkcję `randint(n)` `d` razy, zapisując wynik. Wydrukuj wszystkie wyniki losowań dla każdego przedziału `[0,n)` i sprawdź „na oko”, jak bardzo są do siebie podobne. Spróbuj zarówno małych wartości `n`, jak i `d`, aby dowiedzieć się, czy losowanie tylko kilku liczb losowych powoduje jakieś oczywiste ukierunkowania.
11. Napisz funkcję `swap_columns()` odpowiadającą funkcji `swap_rows()` z punktu 24.5.3. Aby to zrobić, musisz oczywiście przeczytać i zrozumieć trochę kodu z biblioteki `Matrix`. Nie przejmuj się zbytnio wydajnością — nie da się sprawić, aby funkcja `swap_columns()` dorównała szybkością funkcji `swap_rows()`.
12. Zaimplementuj:

```
Matrix<double> operator*(Matrix<double,2>&,Matrix<double>&);
i
Matrix<double,N> operator+(Matrix<double,N>&,Matrix<double,N>&)
```

W razie potrzeby poszukaj matematycznych definicji w jakimś podręczniku.

Podsumowanie

Jeśli nie czujesz się najlepiej w zagadnieniach matematycznych, pewnie nie spodobał Ci się ten rozdział i najprawdopodobniej wybierzesz dziedzinę, w której będzie małe prawdopodobieństwo, że się z tym spotkasz. Jeśli natomiast lubisz matematykę, mamy nadzieję, że spodoba Ci się to, jak wiernie można przedstawić podstawowe koncepcje matematyczne w postaci kodu.

