Metody numeryczne w C++

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf”.
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

Uniwersytet Marii Curie-Skłodowskiej
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki

Metody numeryczne w C++

Paweł Mikołajczak
Marcin Ważny

Lublin 2012

Instytut Informatyki UMCS
Lublin 2012

Paweł Mikołajczak
Marcin Ważny

Metody numeryczne w C++

Recenzent: Jakub Smołka

Opracowanie techniczne: Marcin Ważny, Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach
Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl.

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk
FIGARO Group Sp. z o.o. z siedziba w Rykach
ul. Warszawska 10
08-500 Ryki
www: www.figaro.pl

ISBN: 978-83-62773-34-3

Spis treści

Przedmowa

Metody numeryczne zaliczamy do klasy metod rozwiązujących szerokie spektrum zadań matematycznych. Zadania matematyczne możemy rozwiązywać dokładnie (mówimy wtedy o rozwiązaniach analitycznych) lub w sposób przybliżony. W celu zrozumienia otaczającej nas rzeczywistości tworzymy matematyczne modele (opis formalny) różnego typu zjawisk. Metody numeryczne początkowo stosowano do rozwiązywania ciekawych zadań matematycznych (na przykład do wyliczenia pierwiastka z liczby) lub do wyliczenia określonych parametrów w zjawisku fizycznym (na przykład całkowanie równań ruchu). Z czasem, gdy osiągnięto wystarczający rozwój technik komputerowych (wprowadzenie do powszechnego użytku komputerów domowych), skomplikowane modele matematyczne zostały wykorzystane w takich dziedzinach, jak przewidywanie pogody, medycyna, nauki ekonomiczne, statystyka stosowana oraz wielu innych. Obecnie trudno jest wymienić jakąkolwiek dziedzinę działalności człowieka, w której nie korzystano by z obliczeń numerycznych.

Stosowane modele matematyczne nie zawsze dają się rozwiązać dokładnie. Śmiemy twierdzić, że liczba zadań, które możemy dokładnie rozwiązać, jest stosunkowo mała, ogromna liczba zadań jest rozwiązywana w sposób przybliżony przy pomocy komputerów i technik numerycznych. Modele, które mogą opisywać jakieś zjawisko mogą być bardzo skomplikowane, w praktyce ograniczamy się do rozważania uproszczonych modeli, które dadzą się rozwiązać stosunkowo małym nakładem środków. Stosujemy dość często modele liniowe, na przykład zadanie redukujemy do liniowego równania różniczkowego, albo stosujemy wielomian niskiego stopnia (często drugiego lub trzeciego).

Metody numeryczne są rozwijane od wielu lat, istnieje na ten temat bardzo bogata literatura. Istnieją rozbudowane biblioteki algorytmów numerycznych, klasycznym przykładem jest praca *„Numerical Recipies in C++”*. Tworząc własne programy można korzystać z gotowych algorytmów. Wydaje się jednak, że w celu efektywnego stosowania doskonale opracowanych algorytmów numerycznych, użytkownik powinien posiadać pewien zasób wiedzy zarówno teoretycznej, jak i praktycznej.

Celem prowadzenia wykładów i ćwiczeń z metod numerycznych jest zapoznanie studentów z nowoczesnymi metodami wykonywania obliczeń matematycznych i wszelkiego typu symulacji oraz modelowania komputerowego. Obecnie ważną rolę odgrywa aspekt praktyczny, co prowadzi do szczegółowego omawiania procedur i funkcji, dzięki którym można rozwiązywać zadania wykorzystując komputery.

Niniejszy podręcznik jest przeznaczony dla studentów informatyki na studiach

pierwszego stopnia (licencjat). Jego zadaniem jest łagodne wprowadzenie w fascynujący świat obliczeń numerycznych oraz algorytmów numerycznych. Autorzy kładą nacisk na stronę praktyczną – po krótkim wstępie teoretycznym, każda metoda numeryczna jest implementowana w postaci kodu gotowego do wykonania na komputerze.

Od czytelnika wymagamy znajomości podstaw analizy matematycznej, algebry liniowej oraz liniowych równań różniczkowych. Oczywiście student także powinien znać podstawy programowania.

Wybór języka programowania nie jest zbyt istotny, autorzy zdecydowali się na język C++ ze względu na jego popularność (obecnie, w roku 2012, język C++ i jego podzbiór – język C są najpopularniejszymi językami programowania wśród zawodowców).

Rozdział 1

Wiadomości wstępne

1.1 Wstęp

W skrypcie omawiamy zagadnienia związane z rozwiązywaniem matematycznych problemów przy pomocy komputerów. Do realizacji tego zadania należy tworzyć i analizować metody komputerowe, dzięki którym będzie można rozwiązywać problemy wszelkiego typu. Te metody nazywamy metodami numerycznymi, a badania tego typu nazywamy analizą numeryczną. Analiza numeryczna jest trudną dziedziną, wymaga doskonałej znajomości matematyki.

Metody numeryczne mają duże zastosowanie praktyczne, wykorzystywane są przez ludzi zainteresowanych rozwiązywaniem rozmaitych problemów matematycznych. Przeciętny użytkownik posługujący się metodami numerycznymi zwykle nie jest zainteresowany aspektami teoretycznymi używanej metody, dopóki konkretny algorytm numeryczny poprawnie rozwiązuje zadanie.

Należy jednak pamiętać, że numeryczne metody nie są tzw. „czarną skrzynką". Użytkownik metod numerycznych powinien dysponować minimalną wiedzą o podstawach teoretycznych danej metody oraz jej zakresie stosowania i dokładności. Przy wyborze metody numerycznej należy wiedzieć:

- jakie metody są dostępne;
- jak te metody działają;
- jakie są zalety i wady wybranej metody.

Ze względu na praktyczne aspekty wykorzystania metod numerycznych, powstało wiele bibliotek z gotowymi procedurami i kompletnymi programami komputerowymi rozwiązującymi najróżniejsze zadania. W zasadzie nie zalecamy pisania własnych procedur numerycznych, gdyż bardzo często najbardziej efektywne działanie to skorzystanie z gotowych procedur.

Naszym celem jest dostarczenie odpowiedniej wiedzy w zakresie metod numerycznych tak, aby student stał się inteligentnym użytkownikiem bibliotecznych programów komputerowych.

1.2 Iteracja

Jedną z podstawowych operacji, często stosowanych w metodach numerycznych, jest iteracja (łac. *iteratio* – powtarzanie). Formalnie iteracja jest powtarzaniem pewnej ustalonej sekwencji wyrażeń.

W celu przybliżenia pojęcia iteracji rozważymy rozwiązywanie równania w ogólnej postaci:

$$x = f(x). \tag{1.1}$$

Zakładamy, że $f(x)$ jest funkcją różniczkowalną, a jej wartości można obliczyć dla dowolnej wartości zmiennej rzeczywistej x. Metoda iteracyjna polega na wybraniu wartości początkowej x_0 i wyliczeniu wartości funkcji:

$$x_1 = f(x_0). \tag{1.2}$$

Następnie obliczoną wartość x_1 wykorzystujemy do obliczenia wartości x_2 i następnych:

$$\begin{aligned} x_2 &= f(x_1), \\ x_3 &= f(x_2), \\ &\vdots \\ x_{i+1} &= f(x_i). \end{aligned}$$

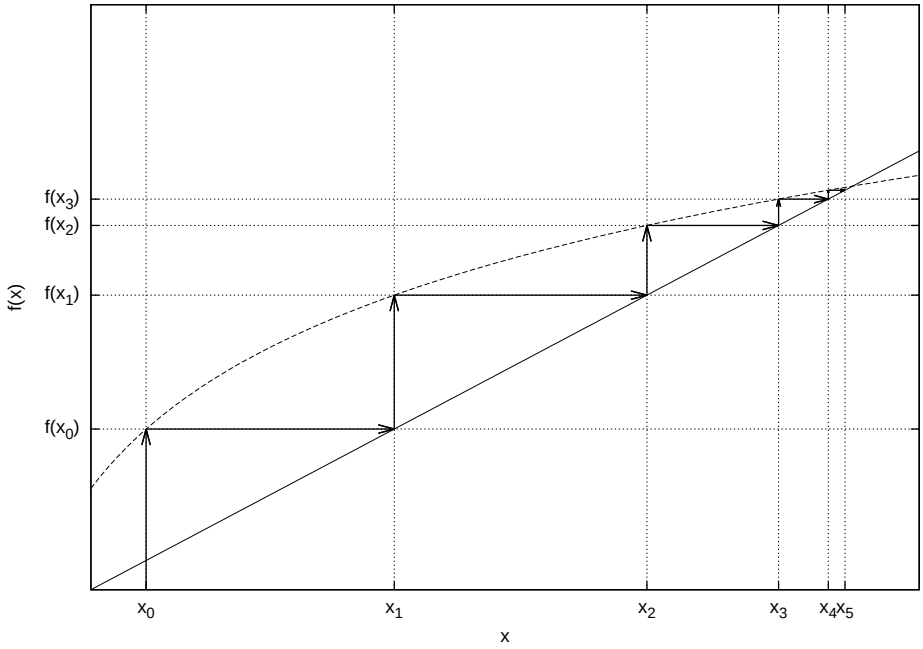
Poszczególne wyliczenie x_{i+1} nazywamy iteracją. W procesie iteracji otrzymujemy ciąg wartości x_i. Taki ciąg może być zbieżny do wartości granicznej:

$$\lim_{i \to \infty} f(x_i) = f(\alpha). \tag{1.3}$$

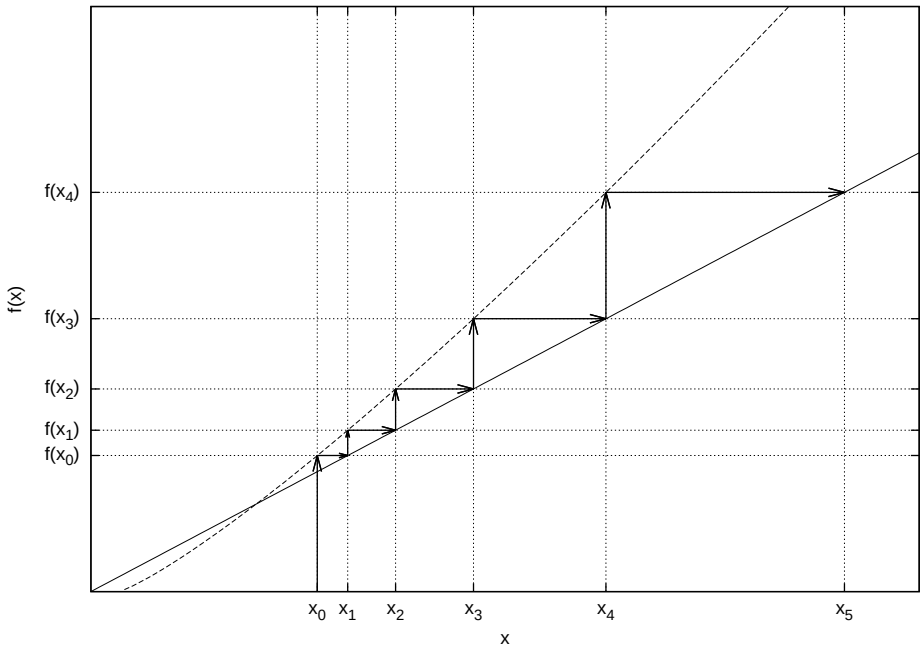
Mówimy, że dla $x = \alpha$, spełnione jest nasze równanie. W procesie iteracji zakładamy, że wraz ze wzrostem i, wzrasta dokładność rozwiązania. Zazwyczaj rozwiązania są asymptotyczne, nie możemy obliczać ciągu iteracyjnego w nieskończoność. Zawsze jest pytanie, kiedy należy przerwać proces iteracji. Odpowiedź jest prosta – przerywamy proces iteracji, gdy osiągniemy zakładaną dokładność. Ustalenie „zakładanej dokładności" nie jest zadaniem trywialnym. Na Rysunku 1.1 pokazana jest interpretacja geometryczna metody rozwiązania naszego równania (1.1). Pierwiastek równania (1.1) otrzymany jest jako punkt przecięcia się krzywej $y = f(x)$ oraz prostej $y = x$. Rozpoczynamy proces iteracji wybierając punkt o współrzędnych $(x_0, f(x_0))$. Obliczamy kolejny punkt $x_1 = f(x_0)$. Z Rysunku 1.1 widzimy wyraźnie, że metoda jest zbieżna, w kolejnych krokach iteracyjnych zbliżamy się monotonicznie do szukanego rozwiązania. Nie zawsze proces iteracyjny jest zbieżny. Na rysunku 1.2 pokazano przykład iteracji rozbieżnej. W trakcie procesu iteracyjnego, kolejne wartości oddalają się coraz wyraźniej od wartości prawdziwej rozwiązania równania (1.1).

Metodę iteracyjną zilustrujemy klasycznym przykładem wyliczania pierwiastka kwadratowego. Zgodnie z tradycją, wynalezienie tej metody przypisujemy I. Newtonowi. Równanie kwadratowe:

$$x^2 = a, \tag{1.4}$$



Rysunek 1.1: Iteracja zbieżna.



Rysunek 1.2: Iteracja rozbieżna.

zapisuje się w postaci

$$x = f(x), \tag{1.5}$$

gdzie:

$$f(x) = 0,5\left(x + \frac{a}{x}\right), \tag{1.6}$$

zakładamy, że $a > 0$. Pierwiastkiem jest $\alpha = \sqrt{a}$. Wzór iteracyjny szybkiego obliczania pierwiastka kwadratowego ma postać:

$$x_{i+1} = 0,5\left(x_i + \frac{a}{x_i}\right), \tag{1.7}$$

przy czym:

$$x_i \to \sqrt{a} : i \to \infty. \tag{1.8}$$

Wybór wartości początkowej nie jest zasadniczym problem, oczywiście, im lepsze będzie przybliżenie początkowe, tym mniej potrzeba będzie iteracji. W praktyce wystarcza, że za wartość początkową przyjmiemy wartość równą jedności.

Na Listingu 1.1 pokazany jest program realizujący szybkie obliczanie pierwiastka kwadratowego. Do funkcji o nazwie `sqrt` przekazujemy argument (liczbę, z której chcemy obliczyć pierwiastek kwadratowy).

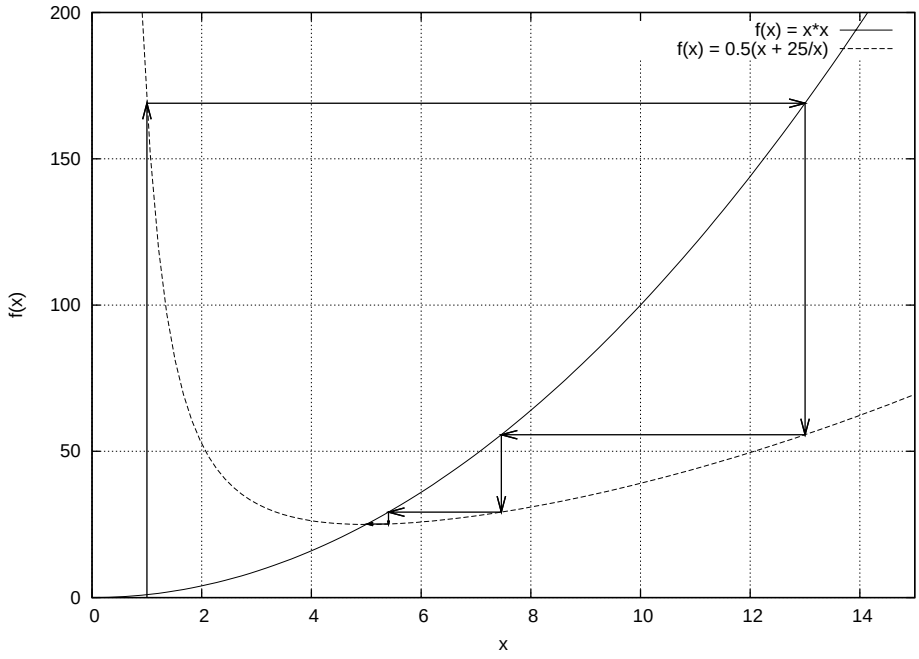
Listing 1.1: Obliczanie pierwiastka kwadratowego.

```
1  #include <iostream>
2  #include <cmath>
3
4  const double E = 0.01;
5
6  double sqrt(double, double, unsigned int &);
7
8  int main()
9  {
10   std::cout << "Podaj liczbe wieksza od zera: ";
11   double a;
12   std::cin >> a;
13   unsigned int n;
14   double x = sqrt(a, E, n);
15   std::cout << "Pierwiastek kwadratowy z liczby "
16             << a << " rowny jest " << x << std::endl
17             << "Liczba iteracji: " << n << std::endl;
18   return 0;
19 }
20
21 double sqrt(double a, double e, unsigned int & n)
22 {
23   if (a == 0)
24     return 1.0;
25   double x = 1.0;
26   n = 0;
27   while (std::fabs(x * x - a) > e)
28     {
29       x = 0.5 * (x + a / x);
30       n++;
31     }
32   return x;
33 }
```

Jako pierwsze przybliżenie (x_1) przyjmuje się 1. Proces obliczeniowy kończymy, gdy wartość obliczanego pierwiastka osiągnie żądaną dokładność. Po uruchomieniu programu otrzymujemy komunikat:

```
1 Pierwiastek kwadratowy z liczby 25 rowny jest 5.00002
2 Liczba iteracji: 5
```

Zauważmy, że metoda jest bardzo wydajna, niewiele iteracji potrzeba, aby osiągnąć żądaną dokładność. Opisany algorytm stosuje się w kalkulatorach kieszonkowych oraz w niektórych pakietach matematycznych. Interpretacja geometryczna opisanego procesu pokazana jest na Rysunku 1.3. W metodach iteracyjnych zazwyczaj



Rysunek 1.3: Iteracyjne obliczenie pierwiastka kwadratowego.

musimy rozwiązać dwa zagadnienia:

- dokładność oszacowania,
- dozwolona ilość iteracji.

W wielu przypadkach obliczeń numerycznych nie potrafimy podać, jaki jest błąd metody. Jeżeli metoda jest zbieżna, to zakładamy, że w miarę wzrostu liczby iteracji, otrzymywana wartość będzie zbliżała się asymptotycznie do wartości prawdziwej. W takim przypadku porównujemy wartość x_i z wartością x_{i-1} i wyliczamy różnicę. Gdy ta różnica jest mniejsza niż zadana początkowa wartość (np. tak jak w naszym programie, gdy różnica jest mniejsza niż 0,01), to uznajemy, że możemy

przerwać proces iteracji a otrzymana wartość x_i jest wystarczająco dobrym przybliżeniem. W zależności od wartości wyrażenia warunkowego, w pętli **while()** będzie wykonywana iteracja aż do osiągnięcia zadanej dokładności:

```
1  while (std::fabs(x * x - a) > e)
```

Może się zdarzyć, że proces będzie rozbieżny lub kryterium precyzji zbyt wygórowane. Należy zadbać o kontrolę liczby iteracji, zazwyczaj ustawiamy licznik pętli i w momencie, gdy liczba iteracji przekroczy ustalona granicę, zatrzymujemy obliczenia.

Rozwiązując zadanie metodami numerycznymi spotkamy się z przypadkami tak zwanej niestabilności numerycznej. Mówiąc nieprecyzyjnie niestabilność numeryczna polega na nieoczekiwanym zachowaniu się rozwiązania. Na przykład, zmieniając jakiś parametr w metodzie numerycznej, otrzymujemy całkowicie inne rozwiązanie. Rozważmy klasyczny przykład niestabilności numerycznej z podręcznika R. Hornecka. Chcemy rozwiązać równanie różniczkowe postaci:

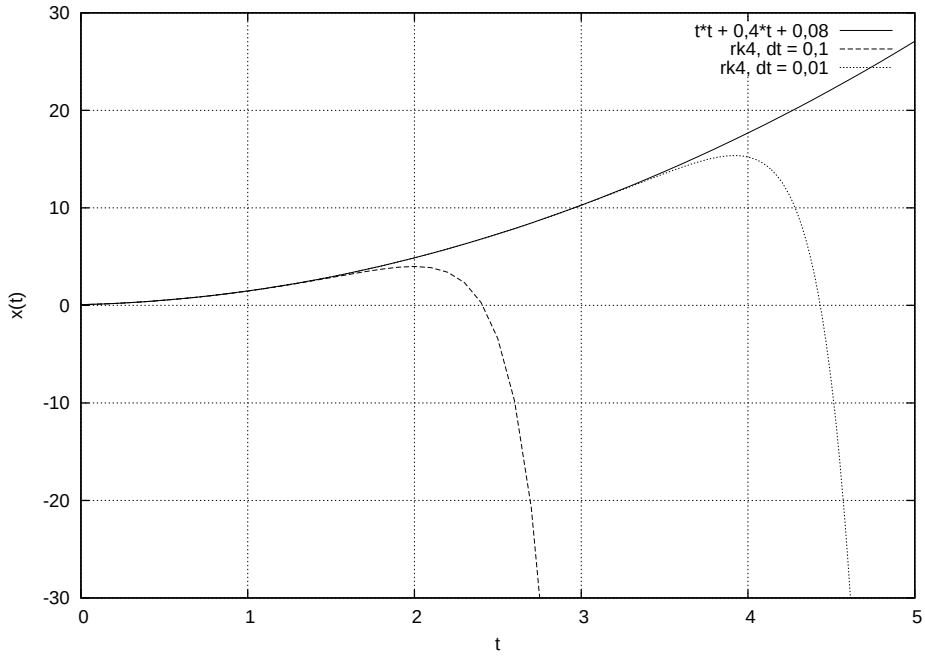
$$\begin{cases} x(0) & = 0,08, \\ \dfrac{dx}{dt} & = 5(x - t^2), \end{cases} \tag{1.9}$$

przy pomocy metody Runge-Kutty 4 rzędu (numeryczne metody rozwiązywania równań różniczkowych zwyczajnych zostaną omówione w dalszych rozdziałach) w przedziale $0 < t < 5$. Przy warunkach początkowych ($x(0) = 0.08$) dokładne rozwiązanie jest znane i ma postać:

$$x = t^2 + 0,4t + 0,08, \tag{1.10}$$

W metodzie Runge-Kutty zasadniczym parametrem jest krok iteracji. Jeżeli zastosujemy krok $\Delta t = 0,1$, to dostaniemy prawie dokładne rozwiązanie dla $0 < t < 1,7$. Powyżej wartości $t = 1,7$ rozwiązanie numeryczne drastycznie odbiega od rozwiązania prawdziwego (zobacz Rysunek 1.4). Zmiana kroku iteracji rozwiązania numerycznego na $\Delta t = 0,01$ przesunie granicę dobrego rozwiązania do wartości $t = 3,7$.

Niestabilność numeryczna może sprawiać wiele kłopotów przy rozwiązywaniu zadań metodami numerycznymi. W opisanym przypadku mogliśmy porównać nasze rozwiązanie numeryczne z rozwiązaniem dokładnym. W praktyce rzadko znamy rozwiązanie analityczne (nie ma sensu rozwiązywanie zadania metodami numerycznymi, gdy znamy dokładne rozwiązanie), stosujemy metody numeryczne, gdy inne techniki są nieosiągalne. Dlatego należy wykonać wiele testów rozwiązań numerycznych, aby mieć pogląd na dokładność rozwiązania. W opisanym przykładzie niestabilności numerycznej rozwiązanie problemu jest znane – do tego typu równań różniczkowych metoda Runge-Kutty się nie nadaje. Rekomendowana jest metoda Geara.



Rysunek 1.4: Przykład niestabilności numerycznej, rozwiązywanie równania różniczkowego.

1.3 Błędy w metodach numerycznych

Rozwiązując zadania metodami numerycznymi popełniamy błędy. Występujące błędy są powodowane wieloma czynnikami. Doskonałe omówienie błędów występujących w analizie numerycznej znajduje się w monografiach A. Ralstona (*Wstęp do analizy numerycznej*) oraz A. Bjorcka i G. Dahlquista (*Metody numeryczne*). Niektóre typy błędów są łatwe do usunięcia, inne ewentualnie można wyeliminować lub zredukować. Ważne jest jednak, aby zdawać sobie sprawę z natury błędów, jakie mogą powstać w wyniku obliczeń przeprowadzanych przy pomocy komputerów. A. Ralston wyróżnia trzy główne źródła błędów:

- grube błędy i pomyłki,
- błędy metody (obcięcia),
- błędy zaokrąglenia.

Pomyłki zdarzają się zawsze, na przykład wprowadzając dużą ilość danych, można źle wprowadzić konkretną liczbę. Uważne, ponowne sprawdzenie wprowadzonych danych może wyeliminować pojawienie się tego typu pomyłki. Błędy metody spowodowane są faktem, że:

- Korzystając z kilku początkowych składników szeregu Taylora. Szacowanie wartości funkcji elementarnych (np. e^x) – zadanie jest rozwiązywane nie w postaci dokładnej, ale w postaci przybliżonej.

- Rozwiązywanie równania $f(x) = 0$ metodą iteracyjną. Otrzymujemy rozwiązanie tylko w granicy, gdy liczba iteracji dąży do nieskończoności. Ponadto zer funkcji w systemach komputerowych szukamy w zdefiniowanym przedziale, a nie w nieskończonym.
- W prostych metodach całkowania numerycznego (np. metoda prostokątów lub trapezów) korzystamy ze skończonej sumy wartości funkcji.

Błędy zaokrąglenia są spowodowane przyczynami technicznymi – w odróżnieniu od metod analizy matematycznej, gdzie operujemy pojęciem ciągłości i nieskończoności, w systemach komputerowych działamy na systemach skończonych i dyskretnych. Używane liczby mają zazwyczaj nieskończone rozwinięcie dziesiętne i trzeba je zaokrąglać.

Jeżeli określimy symbolem w wartość dokładną, symbolem wp wartość przybliżoną, a przez e – błąd bezwzględny, to mamy równość:

$$w = wp + e. \tag{1.11}$$

Wyróżniamy dwa pojęcia błędu:

- błąd bezwzględny (e),
- błąd względny (ew).

Błąd bezwzględny ma postać:

$$e = w - wp. \tag{1.12}$$

Błąd względny definiujemy jako:

$$ew = \frac{e}{w} = \frac{w - wp}{w}. \tag{1.13}$$

Błąd względny (w nie może być równe 0) wyraża się często w procentach. Na przykład błąd względny 5% oznacza tyle samo, co błąd względny $0,05$. W wielu podręcznikach błąd względny definiuje się ze znakiem przeciwnym – nie jest to istotne. Jeżeli liczbę $0,333$ przybliżamy ilorazem $1/3$, to błąd bezwzględny:

$$e = \frac{1}{3}10^{-3}.$$

A błąd względny:

$$ew = 10^{-3}.$$

Należy odróżniać błąd, który może być dodatni albo ujemny. Oznaczenie:

$$w = wp \pm e \tag{1.14}$$

Jest skrótem nierówności:

$$|wp - w \leqslant e| \tag{1.15}$$

Jeżeli mamy oszacowanie $= 0,6677 \pm 0,0001$, to wartość prawdziwa zawiera się w przedziale:

$$0,6676 \leqslant w \leqslant 0,6678. \tag{1.16}$$

Liczby rzeczywiste mają ogólnie nieskończone rozwinięcie dziesiętne. W systemach komputerowych możemy wykorzystywać tylko liczby z rozwinięciem skończonym. Rozróżniamy pojęcie zaokrąglania liczb oraz obcinania liczb. Zawsze musimy określić ile cyfr znaczących będzie zawierała liczba. Zaokrąglanie polega na wyborze liczby najbliższej liczbie zaokrąglanej. Obcinanie (inna nazwy – ucinanie, skracanie) polega na odrzuceniu wszystkich cyfr leżących na prawo od ostatniej ustalonej cyfry znaczącej. W Tabeli 1.1 podajemy przykłady zaokrąglania i skracania liczb. W wielu przypadkach wykonuje się miliony operacji na liczbach – wtedy

Tabela 1.1: Przykłady skracania do trzech liczb ułamkowych.

Liczba	Zaokrąglenie	Skrócenie
0,4398	0,440	0,439
-0,4398	-0,440	-0,439
0,43750	0,438	0,437
0,43650	0,436	0,436
0,43652	0,437	0,436

zaokrąglanie liczb może prowadzić do kumulowania się błędów, co w konsekwencji może prowadzić do błędnych wyników.

1.4 Szereg Taylora

Szeregi Taylora stanowią fundament metod numerycznych. Wiele technik stosowanych w metodach numerycznych korzysta wprost z szeregów Taylora, na przykład do oszacowania błędów.

Jeżeli funkcja $f(x)$ ma n-tą pochodną $f^{(n)}(x)$ w pewnym domkniętym przedziale zawierającym punkt a, wówczas dla każdego x z tego przedziału mamy następujący wzór:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n-1)}(a)}{(n-1)!}(x-a)^{n-1} + \frac{f^{(n)}(c_n)}{n!}(x-a)^n. \tag{1.17}$$

Jest to *wzór Taylora*. Ostatni wyraz we wzorze Taylora jest nazywany *resztą wzoru Taylora*:

$$R_n = \frac{f^{(n)}(c_n)}{n!}(x-a)^n. \tag{1.18}$$

Jeżeli w szeregu Taylora przyjmiemy $a = 0$, to otrzymamy *szereg Maclaurina*:

$$f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \cdots + \frac{f^{(n-1)}(0)}{(n-1)!}x^{n-1} + \frac{f^{(n)}(c_n)}{n!}x^n. \tag{1.19}$$

Wykorzystamy szereg Maclaurina aby rozwinąć funkcję $\sin(x)$. Zgodnie ze wzorem

mamy:

$$\sin(x) = \sin(0) + x\cos(0) - \frac{x^2}{2!}\sin(0) - \frac{x^3}{3!}\cos(0) + \frac{x^4}{4!}\sin(0) + \frac{x^5}{5!}\cos(0) + \dots \ . \tag{1.20}$$

Ponieważ $\sin(0) = 0$ oraz $\cos(0) = 1$, ostatecznie mamy:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \dots \ . \tag{1.21}$$

Podobnie można rozwinąć inne funkcje, rozwinięcie funkcji e^x na szereg Maclaurina ma postać:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots \ . \tag{1.22}$$

Rozdział 2

Interpolacja

2.1 Wstęp

Większość funkcji przechowywanych w formie numerycznej ma postać dyskretnych wartości ze zbioru dziedziny i przeciwdziedziny owej funkcji. Taka funkcja jest zwykle rezultatem przeprowadzonych pomiarów lub wynikiem metody numerycznej. Dane tego typu pokazane są na Rysunku 2.1. Często istnieje potrzeba określenia funkcji dla argumentu, którego wartość znajduje się pomiędzy dwoma wartościami z dyskretnego zbioru argumentów tej funkcji. Technikę określającą tą szukaną wartość nazywa się *interpolacją* (szukanie wartości funkcji poza przedziałem danych nazywamy *ekstrapolacją*).

Niech dana będzie funkcja $f(x) : [a; b] \to \mathbb{R}; a, b \in \mathbb{R}$, która może być aproksymowana przez skończoną liczbę funkcji $s_i(x)$:

$$f(x) = \sum_{i=0}^{n} a_i s_i(x). \tag{2.1}$$

Jeśli funkcja $f(x)$ określona jest przez dyskretny zbiór danych $w(x_i)$ tak, że

$$w(x_1) = f(x_1), w(x_2) = f(x_2), \ldots, w(x_n) = f(x_n), i = 0, 1, \ldots, n$$

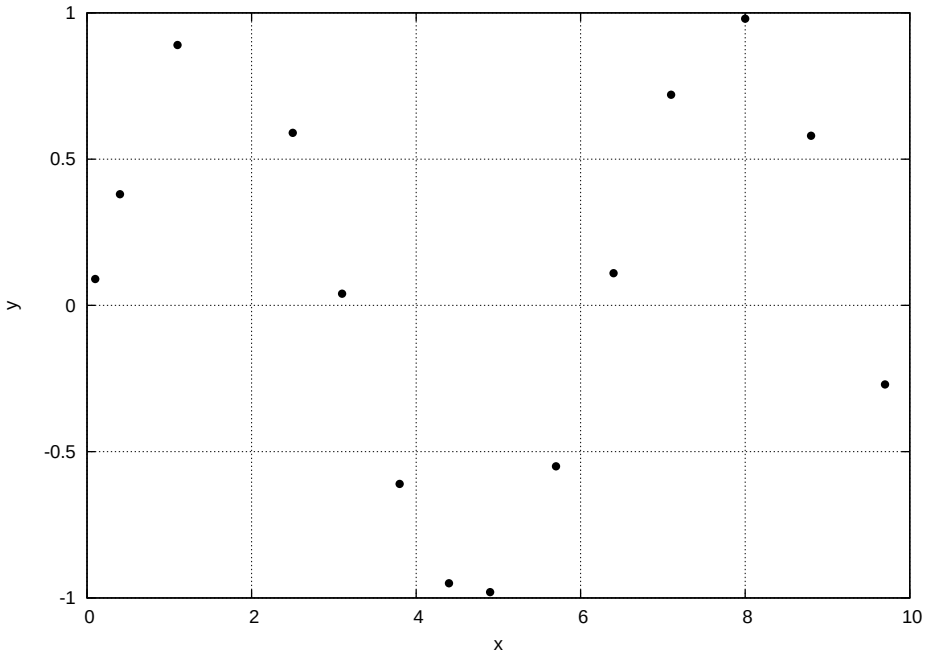
oraz spełniony jest warunek

$$a \leqslant x_1 < x_2 < \cdots < x_n \leqslant b,$$

to interpolacją określa się szukanie funkcji $w(x)$ spełniającej warunek:

$$w(x_j) = \sum_{i=0}^{n} a_i s_i(x_j), j = 0, 1, \ldots, n. \tag{2.2}$$

Zmienne $x_i, i = 1, 2, \ldots, n$ często określa się mianem *węzłów*. Funkcję $w(x)$ nazywa się *wielomianem interpolacyjnym*; $a_i, i = 1, 2, \ldots, n$ to współczynniki tego wielomianu; $s_i(x), i = 1, 2, \ldots, n$ to *funkcje bazowe*. Na funkcję interpolacyjną można



Rysunek 2.1: Przykład funkcji $f(x)$ danej jedynie w dyskretnych wartościach.

nałożyć szereg innych warunków tak, aby spełniała ona pożądane właściwości, często zakłada się różniczkowalność tej funkcji w węzłach.

Aby wielomian $w(x)$ interpolował funkcję $f(x)$, konieczne jest rozwiązanie układu $n+1$ równań (z $n+1$ niewiadomymi c_j) wynikających z (2.1) i (2.2). W formie macierzowo-wektorowej jest to:

$$\begin{bmatrix} s_0(x_0) + s_1(x_0) + \cdots + s_n(x_0) \\ s_0(x_1) + s_1(x_1) + \cdots + s_n(x_1) \\ \vdots \\ s_0(x_n) + s_1(x_n) + \cdots + s_n(x_n) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix}, \tag{2.3}$$

gdzie $s_i(x), i = 0, 1, \ldots, n$ stanowi dowolną bazę wielomianów.

Zanim zdefiniujemy konkretne metody interpolacji podamy niezbędne informacje o wielomianach. Wielomianem n-tego stopnia nazywamy funkcję postaci:

$$w(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_n. \tag{2.4}$$

Użyteczność wielomianów w metodach numerycznych wynika z faktu, że wielomianami możemy przybliżać (aproksymować) wszystkie funkcje ciągłe, opieramy się na słynnym twierdzeniu Weierstrassa:

Twierdzenie 2.1 *Dla dowolnej funkcji $f(x)$ ciągłej w przedziale $[a; b]$ i dowolnego*

$\varepsilon > 0$ istnieje taki wielomian $w(x)$, że:

$$\max_{x\in[a;b]} |f(x) - w(x)| < \varepsilon.$$

W literaturze przedmiotu opisano wiele metod konstrukcji wielomianów aproksymujących funkcje ciągłe. Klasycznym przykładem jest aproksymacja funkcji $\sin(x)$ następującym wielomianem:

$$w(x) = x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \frac{1}{7!}x^7 + \frac{1}{9!}x^9. \tag{2.5}$$

Jest to doskonałe przybliżenie, dla x w całym przedziale $[0; \pi/4]$ spełniony jest warunek:

$$|\sin(x) - w(x)| < 2 \cdot 10^{-9}. \tag{2.6}$$

Fakt, że w metodach numerycznych do przybliżania funkcji ciągłych wykorzystujemy wielomiany, powoduje konieczność wydajnego obliczania wartości odpowiedniego wielomianu. Oczywisty sposób wyliczania wartości wielomianu polega na wyliczeniu odpowiedniej potęgi ustalonej wartości x_i a następnie pomnożenie tej wartości przez odpowiedni czynnik a_i i zsumowaniu wszystkich jednomianów. Ta metoda dla wykonania obliczeń dla wielomianu n-tego stopnia wymaga wykonania $2n-1$ mnożeń i n dodawań (lub odejmowań). Bardzo wcześnie zorientowano się, że wykorzystując komputery do wykonywania obliczeń wielomianów istnieje bardziej wydajny schemat. Wielomian postaci:

$$w(x) = a_0x^n + a_1x^{n-1} + \cdots + a_n,$$

przekształcamy do postaci:

$$(\ldots((a_0x + a_1)x + \cdots + a_{n-1})x + a_n.$$

Ten schemat obliczania wartości wielomianu nosi nazwę schematu Hornera. Wartość wielomianu $w(x)$ zgodnie ze schematem Hornera obliczamy przy pomocy wzorów:

1. $w_0 = a_0$,
2. $w_k = w_{k-1}x + a_k : k = 1, 2, \ldots, n$,
3. $w(x) = w_n$.

W schemacie Hornera podczas obliczana wartości wielomianu $w(x)$ należy wykonać n mnożeń i n dodawań (lub odejmowań).

2.2 Interpolacja Lagrange'a

Rozważmy szereg punktów $(x_j, f(x_j))$, dla których x_j nie są równoodległe $(j = 0, 1, \ldots, n)$. Do tych punktów można dopasować wielomian stopnia n. Ba-

za wielomianów $s_i(x)$ dana jest przez funkcje Lagrange'a:

$$l_i(x) = \frac{\prod_{j=0,j\neq i}^{n}(x - x_j)}{\prod_{j=0,j\neq i}^{n}(x_i - x_j)} = \begin{cases} 1 : x = x_i \\ 0 : x = x_j \neq x_i \end{cases} . \tag{2.7}$$

A więc wielomian Lagrange'a można zapisać:

$$w(x) = \sum_{i=0}^{n} a_i l_i(x) = \sum_{i=0}^{n} a_i \frac{\prod_{j=0,j\neq i}^{n}(x - x_j)}{\prod_{j=0,j\neq i}^{n}(x_i - x_j)}. \tag{2.8}$$

Na mocy (2.7) oraz warunku, że w węzłach wielomian interpolacyjny jest równy funkcji interpolowanej, otrzymuje się $a_i = f(x_i)$, a więc:

$$w(x) = \sum_{i=0}^{n} f(x_i) \frac{\prod_{j=0,j\neq i}^{n}(x - x_j)}{\prod_{j=0,j\neq i}^{n}(x_i - x_j)}, \tag{2.9}$$

co stanowi zależność, za pomocą której można interpolować funkcję $f(x_i), i = 1, 2, \ldots, n$ w punktach $x \in \mathbb{R}$.

Aproksymację metodą Lagrange'a zilustrujemy przykładem (R. Hornbeck). Mamy następujący zbiór przedstawionych w Tabeli 2.1: Chcemy znać wartość (interpolowaną) dla $x = 7$. Wyliczamy odpowiednie wartości:

Tabela 2.1: Przykładowa funkcja zadana dla dyskretnych argumentów.

i	x_i	$f(x_i)$
0	1	1
1	2	3
2	4	7
3	8	11

$$\begin{aligned} w(x) = f(1)\frac{(7-2)(7-4)(7-8)}{(1-2)(1-4)(1-8)} + f(2)\frac{(7-1)(7-4)(7-8)}{(2-1)(2-4)(2-8)} \\ + f(4)\frac{(7-1)(7-2)(7-8)}{(4-1)(4-2)(4-8)} + f(8)\frac{(7-1)(7-2)(7-4)}{(8-1)(8-2)(8-4)} \\ = 0,71429 + 3(-1,5) + 7 \cdot 1,25 + 11 \cdot 0,53571 = 10,8571. \end{aligned}$$

Na Listingu 2.2 przedstawiona została funkcja `lagrange_interpolation`, funkcja ta wyznacza wartość wielomianu interpolującego na podstawie (2.9). Listing 2.1 zawiera pomocniczą strukturę wykorzystywaną przez funkcję `lagrange_interpolation`.

Listing 2.1: Definicja struktury `XF`

```
1 typedef struct
2 {
3   double x, f;
4 } XF;
```

Listing 2.2: Funkcja lagrange_interpolation.

```
1 double lagrange_interpolation(const XF *xf, unsigned int xf_size,
      double x)
2 {
3   double result = 0.0;
4   for (int i = 0; i < xf_size; i++)
5     {
6       double f1 = 1.0;
7       double f2 = 1.0;
8       for (int j = 0; j < xf_size; j++)
9         {
10          if (i != j)
11            {
12              f1 *= x - xf[j].x;
13              f2 *= xf[i].x - xf[j].x;
14            }
15        }
16      result += xf[i].f * f1 / f2;
17    }
18  return result;
19 }
```

Funkcja `lagrange_interpolation` przyjmuje trzy parametry, są to:

- `xf` – wskaźnik na tablicę struktur `XF`, struktura ta definiuje dwa pola typu `double` i reprezentuje węzeł;
- `xf_size` – rozmiar tablicy `xf`;
- `x` – argument, dla którego funkcja zwróci wartość wielomianu interpolującego.

Funkcja `lagrange_interpolation` zwraca wartość, która interpoluje funkcję przekazaną w parametrze `xf` dla argumentu `x`.

2.3 Interpolacja Newtona

Dla interpolacji Newtona definiuje się bazę wielomianów (zwanych wielomianami Newtona) jak następuje:

$$\begin{cases} s_0(x) & = 1, \\ s_i(x) & = \prod_{j=0}^{i-1}(x - x_j) : i = 1, 2, \ldots, n. \end{cases} \tag{2.10}$$

Wielomian interpolacyjny można więc zapisać:

$$w(x) = \sum_{i=0}^{n} a_i s_i(x) = a_0 + \sum_{i=1}^{n} a_i \prod_{j=0}^{i-1}(x - x_j). \tag{2.11}$$

Korzystając z założenia, że wielomian interpolacyjny powinien być równy funkcji interpolowanej w węzłach, otrzymuje się układ równań z niewiadomymi $a_i, i =$

$1, 2, \ldots, n$:

$$\begin{cases} f(x_0) & = a_0, \\ f(x_1) & = a_0 + a_1(x_1 - x_0), \\ f(x_2) & = a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1), \\ & \vdots \\ f(x_n) & = a_0 + a_1(x_n - x_0) + \cdots + a_n(x_n - x_0)(x_n - x_1)\ldots(x_n - x_{n-1}). \end{cases} \tag{2.12}$$

Rozwiązując układ równań (2.12) od a_0 otrzymuje się:

$$\begin{aligned} a_0 &= f(x_0), \\ a_1 &= \frac{f(x_1) - a_0}{x_1 - x_0}, \\ a_2 &= \frac{f(x_2) - a_0 - a_1(x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)} = \frac{\frac{f(x_2) - a_0}{x_2 - x_0} - a_1}{x_2 - x_1}. \end{aligned}$$

Na podstawie indukcji matematycznej można zapisać ogólną zależność określającą współczynniki wielomianu interpolacyjnego a_n:

$$a_n = \begin{cases} t_0 = f(x_n), \\ t_i = \dfrac{t_{i-1} - a_{i-1}}{x_n - x_{i-1}} : i = n, n-1, \ldots, 1. \end{cases} \tag{2.13}$$

Jako przykład interpolacji z użyciem wielomianów Newtona rozważmy funkcję zapisaną w Tabeli 2.2. Wartością argumentu dla którego chcemy interpolować

Tabela 2.2: Przykładowa funkcja zadana dla dyskretnych argumentów.

i	x_i	$f(x_i)$
0	1	1
1	3	5
2	7	9
3	12	13

zadaną funkcję niech będzie $x = 5$. Na podstawie (2.11) otrzymuje się:

$$\begin{aligned} w(5) &= 1 + \frac{5-1}{3-1}(5-1) + \frac{\frac{9-1}{7-1} - \frac{5-1}{3-1}}{7-3}(5-1)(5-3) + a_3(5-1)(5-3)(5-7) \\ &= 1 + 2 \cdot 4 - 0,166 \cdot 8 - \frac{\frac{\frac{13-1}{12-1} - 2}{12-3} + 0,166}{12-7}16 \\ &= 1 + 8 - 1,333 - 0,207 = 7,46. \end{aligned}$$

Listing 2.4 przedstawia kod funkcji `newton_interpolation`, która realizuje algorytm interpolacji Newtona.

Listing 2.3: Definicja struktury `XF`

```
typedef struct
{
  double x, f;
} XF;
```

Listing 2.4: Funkcja newton_interpolation.

```
double newton_interpolation(const XF *xf, unsigned int xf_size, double
      x)
{
  double result = xf[0].f;
  double *a = new double[xf_size];

  for (int i = 0; i < xf_size; i++)
    a[i] = xf[i].f;
  for (int i = 1; i < xf_size; i++)
    for (int j = 0; j < i; j++)
      a[i] = (a[i] - a[j]) / (xf[i].x - xf[j].x);

  for (int i = 1; i < xf_size; i++)
    {
      double f = 1.0;
      for (int j = 0; j < i; j++)
        f *= x - xf[j].x;
      result += a[i] * f;
    }
  delete [] a;
  return result;
}
```

Funkcja z Listingu 2.4 przyjmuje trzy parametry, są to:

- `xf` – wskaźnik na tablicę struktur `XF`, zdefiniowaną tak jak na Listingu 2.3;
- `xf_size` – rozmiar tablicy `xf`;
- `x` – argument, dla którego funkcja zwróci wartość wielomianu interpolującego.

Wartością zwracaną przez funkcję `newton_interpolation` jest wartość interpolująca funkcję `xf` dla argumentu `x`.

2.4 Interpolacja Czybyszewa

Istnieje wiele metod konstruowania wielomianów interpolacyjnych dla funkcji $f(x)$ w przedziale $[a; b]$. W zasadzie nie przyjmuje się żadnych warunków na wybór węzłów interpolacyjnych, poza oczywistym, aby węzły leżały w tym przedziale. Ponieważ możemy wybierać węzły na różne sposoby naturalne jest pytanie jak wybrać węzły interpolacyjne $x_0, x_1, \ldots, x_n$ tak aby odpowiadający im wielomian interpolacyjny najlepiej przybliżał daną funkcję $f(x)$ w zadanym przedziale $[a; b]$. Wzór interpolacyjny można zapisać w postaci:

$$f(x) = w(x) + r_n(x), \tag{2.14}$$

gdzie $w(x)$ jest wielomianem interpolacyjnym, który w punktach $x_0, x_1, \ldots, x_n$ przyjmuje wartości równe $f(x)$, $r(x)$ jest resztą postaci:

$$r_n(x) = f(x, x_0, x_1, \ldots, x_n)s(x), \tag{2.15}$$

gdzie $s(x)$ jest dane wzorem:

$$s(x) = (x - x_0)(x - x_1)\ldots(x - x_n). \tag{2.16}$$

We wzorze (2.14) może wystąpić zarówno wielomian interpolacyjny Lagrange'a jak i wielomian interpolacyjny Newtona. Istnieje twierdzenie (patrz A. Ralston, *Wstęp do metod numerycznych, PWN, 1971*), które określa $r_n(x)$:

$$r_n(x) = f(x) - w(x) = \frac{1}{(n+1)!} f^{(n+1)}(\alpha)s(x), \tag{2.17}$$

gdzie α zawiera się między najmniejszą i największą z liczb $x_0, x_1, \ldots x_n$ i x. We wzorze (2.17) czynnik: $f^{(n+1)}(\alpha)$ nie zależy od węzłów, to zadanie sprowadza się do wyznaczenia postaci wielomianu $s(x)$ w taki sposób, aby ten wielomian stopnia $n+1$ najmniej odchylał się od zera w całym przedziale $[a; b]$. Aby uprościć zadanie możemy rozważania zawęzić do przedziału $[-1; 1]$. Tego typu redukcje zawsze można wprowadzić, zamiast zmiennej x wprowadzamy przeskalowaną wielkość y:

$$x = \frac{b-a}{2}y + \frac{a+b}{2} \tag{2.18}$$

W ten sposób przedział $[a; b]$ zmiennej x został sprowadzony do przedziału $[-1; 1]$ dla zmiennej y. A co za tym idzie mamy następujące zadanie: *Spośród wszystkich wielomianów $s(x)$ co najwyżej $n+1$ stopnia, znaleźć taki wielomian, który w przedziale $[-1; 1]$ będzie najmniej odchylał się od zera.* To zadanie zostało rozwiązane w roku 1857 przez P. Czebyszewa. Czebyszew udowodnił, że powyższy warunek spełniają wielomiany postaci:

$$T_n(x) = \cos(n \cdot \arccos(x)). \tag{2.19}$$

Wielomiany $T_n(x)$ są zwane wielomianami Czebyszewa. Przypominamy, że w literaturze polskiej funkcje odwrotne do trygonometrycznych oznaczane są symbolem arcsin, w literaturze anglosaskiej symbolem $\sin^{-1}$. Wielomiany Czebyszewa spełniają związek rekurencyjny:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x). \tag{2.20}$$

Na podstawie tego wzoru można wyliczyć wszystkie wielomiany Czebyszewa jeżeli znane są dwa pierwsze:

$$T_0(x) = \cos(0) = 1, \tag{2.21}$$

$$T_1(x) = \cos(\arccos(x)) = x. \tag{2.22}$$

Początkowe wielomiany Czebyszewa mają postać:

$$\begin{aligned}
T_0 &= 1,\\
T_1 &= x,\\
T_2 &= 2x^2 - 1,\\
T_3 &= 4x^3 - 3x,\\
T_4 &= 8x^4 - 8x^2 + 1,\\
T_5 &= 16x^5 - 20x^3 + 5x,\\
T_6 &= 32x^6 - 48x^4 + 18x^2 - 1,\\
&\vdots
\end{aligned}$$

Ponieważ funkcją generującą wielomiany Czebyszewa jest $\cos(\cdot)$, maksymalna wartość tych wielomianów równa jest jedności. Wielomian $s(x)$, który minimalizuje błąd aproksymacji może być wyrażony poprzez wielomiany Czebyszewa:

$$s(x) = 2^{-n} T_{n+1}(x). \tag{2.23}$$

Najlepszy wielomian interpolacyjny otrzymamy, gdy jako węzły zostaną wybrane zera wielomianu Czebyszewa $T_{n+1}(x)$, które nie jest trudno wyliczyć. Problem sprowadza się do rozwiązania równania trygonometrycznego:

$$\cos((n+1)\arccos(x)) = 0. \tag{2.24}$$

Rozwiązanie tego równania ma postać:

$$x_k = \cos\left(\frac{(2k+1)\pi}{2n+2}\right), \tag{2.25}$$

dla $k = 0, 1, \ldots, n$. Jak już poprzednio wskazaliśmy, każdy przedział $[a; b]$ można sprowadzić do przedziału $[-1; 1]$. Ogólny wzór na optymalny, w sensie Czebyszewa, wielomian interpolacyjny, otrzymamy wtedy, gdy wybierzemy następujące węzły interpolacyjne:

$$x_k = \frac{b-a}{2}\cos\left(\frac{(2k+1)\pi}{2n+2}\right) + \frac{a+b}{2}. \tag{2.26}$$

Możemy na przykład podać pierwiastki równania dla $T_6(x)$:

$$\begin{aligned}
x_0 &= 0,96592583,\\
x_1 &= 0,70710678,\\
x_2 &= 0,25881905,\\
x_3 &= -0,25881905,\\
x_4 &= -0,70710678,\\
x_5 &= -0,96592583.
\end{aligned}$$

Wielomian interpolacyjny zbudowany na powyższych węzłach nosi nazwę optymalnego wielomianu interpolacyjnego funkcji $f(x)$ w przedziale $[a; b]$.

Omawiane zagadnienie zilustrujemy przykładem (R. Hornbeck). Mamy równanie w postaci:

$$f(x) = \sin^2(x) + 2\cos(3x), \tag{2.27}$$

dla $0 \leqslant x \leqslant \pi$. Naszym zadaniem będzie skonstruowanie wielomianu interpolacyjnego piątego stopnia. W ustalonym przedziale wybierzemy 6 punktów wykorzystując metodę Czebyszewa, funkcja $f(x)$ będzie próbkowana w punktach wyznaczonych przez pierwiastki $T_6(x')$, w przedziale $-1 \leqslant x' \leqslant 1$. Pamiętamy, że w przedziale $[a; b]$ mamy skalowanie:

$$x_i = \frac{\pi - 0}{2} x'_i + \frac{\pi + 0}{2} = \frac{\pi}{2} x'_i + \frac{\pi}{2}, \tag{2.28}$$

dla $i = 0, 1, \ldots, 5$. W Tabeli 2.3 mamy pokazane wartości x' (są to pierwiastki równania $T_6(x')$), przeskalowane wartości x oraz $f(x)$. Gdy ustalone mamy węzły

Tabela 2.3: Węzły interpolacyjne Czebyszewa.

i	x'_i	x_i	$f(x_i)$
0	-0,965926	0,053524	1.977134
1	-0,707107	0,460076	0.575988
2	-0,258819	1,164244	-1.034340
3	0,258819	1,977347	2.721584
4	0,707107	2,681516	-0.181677
5	0,965926	3,088068	-1.971407

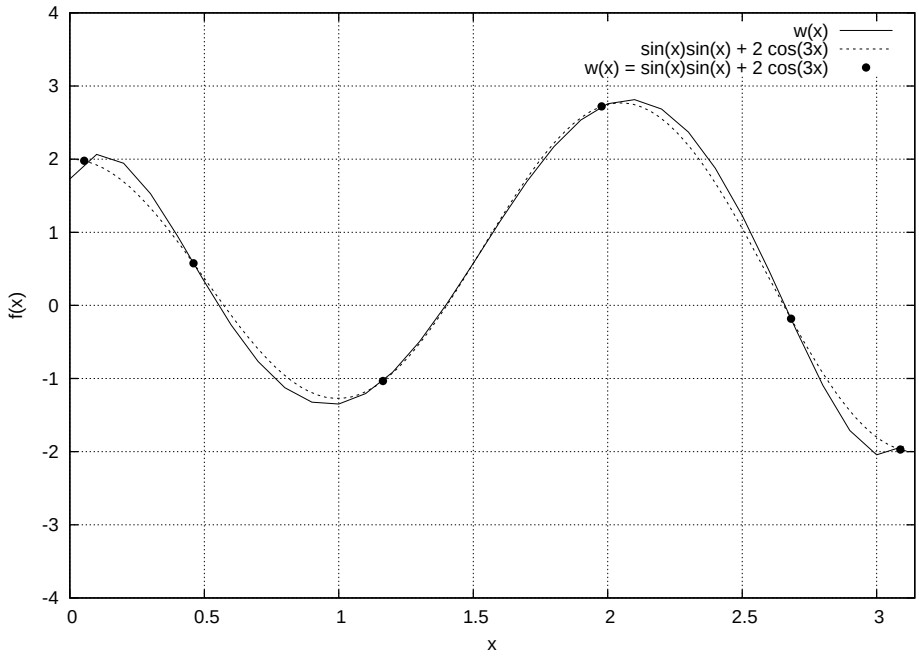
interpolacyjne możemy przy pomocy interpolacji Lagrange'a lub Newtona otrzymać wielomian interpolacyjny, którym można aproksymować rozważaną funkcję. Na Rysunku 2.2 przedstawiono interpolację funkcji (2.28) dobierając węzły interpolacyjne Czybyszewa oraz stosując interpolację wielomianami Lagrange'a.

Należy zaznaczyć, że użycie interpolacji Czybyszewa jest możliwe tylko wtedy gdy węzły interpolacji mogą być dobierane zgodnie z (2.26). Niestety nie zawsze jest to możliwe, jeśli dane określające funkcję, która ma być interpolowana, pochodzą z urządzenia, które dokonuje pomiaru co pewien stały przedział czasu lub są zadane z góry, interpolacja w sensie Czybyszewa nie jest możliwa. W przypadku gdy węzły można wybrać dowolnie (np. funkcja zadana jest w postaci jawnej) interpolacja daje dobre rezultaty, co zostanie pokazane na końcu rozdziału.

2.5 Interpolacja funkcjami sklejanymi

Aproksymacja wielomianowa nie zawsze daje dobre rezultaty, szczególnie gdy zachodzi potrzeba użycia wielomianów wysokiego stopnia i dopasowanie staje się w charakterze oscylacyjne (ang. *wiggly*). Zachodzi wtedy potrzeba znalezienia metod, które będą dawały bardziej gładkie dopasowanie.

Projektanci okrętów w swoich warsztatach od stuleci używali elastycznych liniałów (ang. *spline*), które odpowiednio wygięte przechodziły w gładki sposób przez



Rysunek 2.2: Interpolacja funkcji za pomocą wielomianów Lagrange'a i węzłów Czybyszewa.

dane punkty (x_i, y_i). Dzięki tej metodzie można było modelować bardzo skomplikowane kształty burt okrętów.

Niech $y = f(x)$ będzie równaniem krzywej, jaką modeluje liniał dopasowany do zadanych punktów (Rysunek 2.3). Tego typu funkcje (np. korzystając z teorii elastyczności) można opisać jako zbudowane ze zbioru wielomianów trzeciego stopnia w ten sposób, że funkcja $f(x)$ i jej dwie pierwsze pochodne są wszędzie ciągłe. Tą funkcję nazywamy *funkcją sklejaną sześcienną* (ang. *cubic spline function*), a punkty $x_i : i = 0, 1, 2, \ldots, n$ nazywamy węzłami.

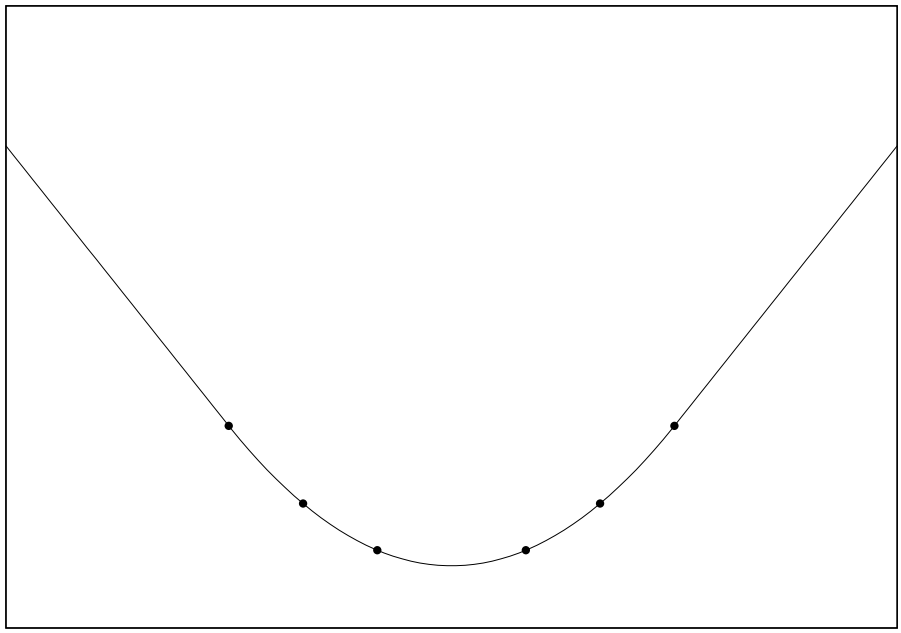
Idea funkcji sklejanych dla potrzeb metod numerycznych została stosunkowo późno zaadaptowana. Te funkcje do praktyki matematyki stosowanej wprowadził w 1946 roku Schoenberg.

Konstrukcja funkcji interpolacyjnej typu funkcji sklejanej sześciennej jest następująca. Niech będzie dana seria punktów $x_i : i = 0, 1, \ldots, n$, które w ogólności nie muszą być równoodległe oraz wartości funkcji $f(x_i)$. Rozważmy dwa przyległe punkty x_i i x_{i+1}. Chcemy przeprowadzić przez te dwa punkty funkcję sześcienną i wykorzystać ją jako funkcję interpolacyjną między tymi punktami. Funkcja ma postać:

$$F_i(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3, \tag{2.29}$$

dla $x_i \leqslant x \leqslant x_{i+1}$. Widzimy, że mamy cztery niewiadome i oczywiste dwa warunki:

$$F_i(x_i) = f(x_i),$$



Rysunek 2.3: Powyginany liniał, dopasowany do punktów.

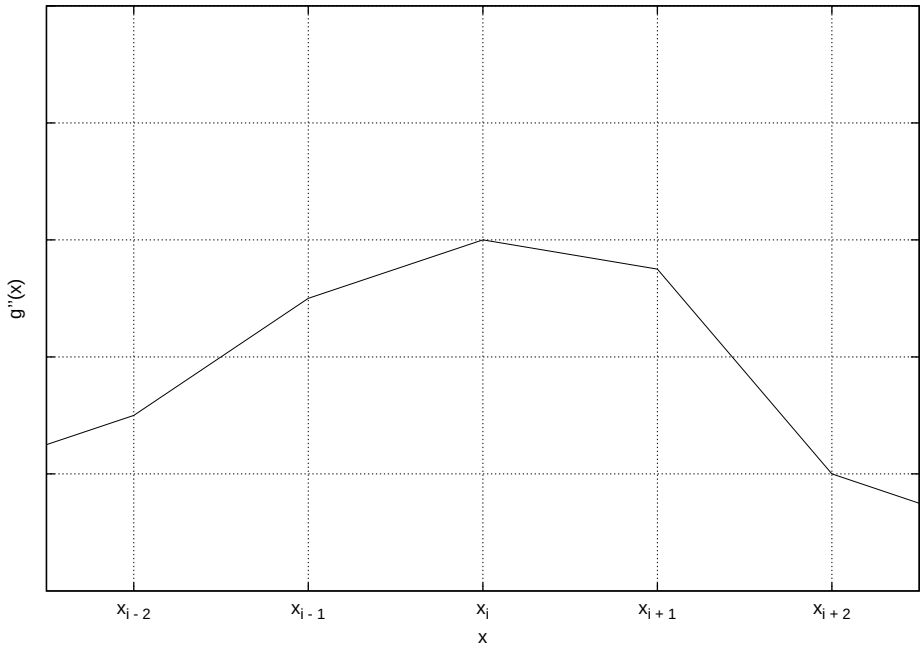
$$F_i(x_{i+1}) = f(x_{i+1}).$$

Musimy znaleźć pozostałe dwa warunki. Najbardziej efektywną metodą jest „zszycie" pierwszych i drugich pochodnych. Tego typu procedura jest przeprowadzana przez punkty w przedziale $x_0 \leqslant x \leqslant x_n$. Należy pamiętać, że punkt początkowy i końcowy są traktowane odrębnie. Z tych funkcji konstruujemy funkcję aproksymującą dla całego obszaru. Oznaczamy funkcję, aproksymującą symbolem $g(x)$ i nazywamy ją funkcją sklejaną (ang. *cubic spline*). Należy zauważyć, że druga pochodna funkcji $g(x)$ jest ciągła w całym obszarze $x_0 \leqslant x \leqslant x_n$. Ten fakt ilustruje Rysunek 2.4. Pamiętamy, że druga pochodna funkcji wielomianu trzeciego stopnia jest linią prostą. Dzięki liniowości, druga pochodna w punkcie x jest dana wzorem:

$$g''(x) = g''(x_i) + \frac{x - x_i}{x_{i+1} - x_i}(g''(x_{i+1}) - g''(x_i)). \tag{2.30}$$

Całkując to równanie dwa razy (wiemy, że $g(x_i) = f(x_i)$ oraz $g(x_{i+1}) = f(x_{i+1})$) otrzymujemy:

$$\begin{aligned} g(x) = F_i(x) = \frac{g''(x_i)}{6}\left(\frac{(x_{i+1} - x)^3}{\Delta x_i} - \Delta x_i(x_{i+1} - x)\right) \\ + \frac{g''(x_{i+1})}{6}\left(\frac{(x - x_i)^3}{\Delta x_i} - \Delta x_i(x - x_i)\right) \end{aligned}$$



Rysunek 2.4: Druga pochodna funkcji sklejanej.

$$+ f(x_i)\left(\frac{x_{i+1}-x}{\Delta x_i}\right) + f(x_{i+1})\left(\frac{x-x_i}{\Delta x_i}\right). \tag{2.31}$$

W powyższym wzorze $\Delta x_i = x_{i+1} - x_i$ i druga pochodna nie jest znana. Musi być wyliczona. Można ją obliczyć narzucając warunki zszycia (ang. *matching conditions*):

$$F_i'(x_i) = F_{i-1}'(x_i),$$
$$F_i''(x_i) = F_{i-1}''(x_i).$$

Jeżeli wykorzystamy te warunki to równanie (2.31) przybierze postać:

$$\left(\frac{\Delta x_{i-1}}{\Delta x_i}\right) g''(x_{i-1}) + \left(\frac{2(x_{i+1}-x_{i-1})}{\Delta x_i}\right) g''(x_i) + (1)g''(x_{i+1})$$
$$= 6\left(\frac{f(x_{i+1})-f(x_i)}{(\Delta x_i)^2} - \frac{f(x_i)-f(x_{i-1})}{(\Delta x_i)(\Delta x_{i-1})}\right), \tag{2.32}$$

dla $i = 1, 2, \ldots, n-1$. Wzór ten upraszcza się jeśli przedziały są równoodległe do:

$$(1)g''(x_{i-1}) + (4)g''(x_i) + (1)g''(x_{i+1}) = 6\left(\frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{(\Delta x_i)^2}\right), \tag{2.33}$$

dla $i = 1, 2, \ldots, n-1$. W dwóch ostatnich wzorach mamy $n-1$ równań i $n+1$ niewiadomych. Dwa pozostałe równania otrzymamy narzucając warunki na punkty

końcowe:

$$g''(x_0) = 0, \tag{2.34}$$
$$g''(x_n) = 0. \tag{2.35}$$

Otrzymana postać $g(x)$ nosi nazwę naturalnych funkcji sklejanych (ang. *natural cubic spline*). Układ równań (2.31) i (2.32) może teraz być rozwiązany dla:

$$g''(x_1), g''(x_2), \ldots, g''(x_{n-1}). \tag{2.36}$$

Taki układ równań ma specyficzną strukturę i może być efektywnie rozwiązany przez metodę Thomasa, omówioną w Rozdziale 7.

Metodę funkcji sklejanych zilustrujemy przykładem dla danych przedstawionych w Tabli 2.4. Chcemy określić wartość funkcji $f(x)$ dla $x = 5$ wykorzystując

Tabela 2.4: Przykładowa funkcja zadana dla dyskretnych argumentów.

i	x_i	$f(x_i)$
0	1	4
1	4	9
2	6	15
3	9	7
4	10	3

aproksymację funkcjami sklejanymi. Na początku ustalamy, że:

$$g''(1) = g''(10) = 0$$

Dla $i = 1$ mamy:

$$\left(\frac{4-1}{6-4}\right) g''(1) + \left(\frac{2(6-1)}{6-4}\right) g''(4) + (1)g''(6) = \left(\frac{15-9}{(6-4)^2} - \frac{9-4}{(6-4)(4-1)}\right),$$

co daje:

$$5g''(4) + g''(6) = 4.$$

Podobnie dla $i = 2$ mamy:

$$0,66667g''(4) + 3,33333g''(6) + g''(9) = -11,33333.$$

Dla $i = 3$ otrzymujemy:

$$3g''(6) + 8g''(9) = -8.$$

Rozwiązując ten układ równań mamy następujące rozwiązania:

$$g''(4) = 1,56932, \tag{2.37}$$
$$g''(6) = -3,84661, \tag{2.38}$$
$$g''(9) = 0,44248. \tag{2.39}$$

Dla określenia funkcji dla $x = 5$ musimy wykorzystać przedział $4 \leqslant x \leqslant 6$. Na podstawie wzoru (2.31) mamy:

$$\begin{aligned} F_1 = &\frac{g''(4)}{6}\left(\frac{(6-5)^3}{6-4} - (6-4)(6-5)\right) \\ &+ \frac{g''(6)}{6}\left(\frac{(5-4)^3}{6-4} - (6-4)(5-4)\right) \\ &+ 9\left(\frac{(6-5)^3}{6-4} + 15\left(\frac{5-4}{6-4}\right)\right). \end{aligned} \tag{2.40}$$

Jeśli do wzoru (2.40) wstawimy wartości na drugie pochodne, to otrzymamy wynik:

$$F_1(5) = 12,56932. \tag{2.41}$$

Na Listingu 2.6 zamieszczono kod funkcji realizującej metodę interpolacji za pomocą omówionych funkcji sklejanych.

Listing 2.5: Definicja struktury `XF`

```
typedef struct
{
  double x, f;
} XF;
```

Listing 2.6: Funkcja spline_interpolation.

```
double spline_interpolation(const XF *xf, unsigned int xf_size, double
      x)
{
  double *_a = new double[xf_size];
  double *_b = new double[xf_size];
  double *_c = new double[xf_size];
  double *_x = new double[xf_size];
  double *_d = new double[xf_size];
  int i0 = 0;
  bool f = false;

  for (int i = 1; i < xf_size - 1; i++)
    {
      double dxi = xf[i + 1].x - xf[i].x;
      double dxi1 = xf[i].x - xf[i - 1].x;
      _a[i] = dxi1 / dxi;
      _b[i] = 2 * (xf[i + 1].x - xf[i - 1].x) / (dxi);
      _c[i] = 1;
      _d[i] = 6 * (((xf[i + 1].f - xf[i].f) / (dxi * dxi)) - ((xf[i].f
            - xf[i - 1].f) / (dxi * dxi1)));

      if (!f && xf[i].x <= x && x <= xf[i + 1].x)
        {
          i0 = i;
          f = true;
        }
    }

  _x[0] = _x[xf_size - 1] = 0;
  thomas_method(_a + 1, _b + 1, _c + 1, _x + 1, _d + 1, xf_size - 2);
```

```
29
30    double dxi = xf[i0 + 1].x - xf[i0].x;
31    double dxfi0 = x - xf[i0].x;
32    double dxfi1 = xf[i0 + 1].x - x;
33    double fx = (_x[i0] / 6.0) * ((dxfi1 * dxfi1 * dxfi1) / dxi - dxi *
          dxfi1)
34      + (_x[i0 + 1] / 6.0) * ((dxfi0 * dxfi0 * dxfi0) / dxi - dxi *
            dxfi0)
35      + xf[i0].f * dxfi1 / dxi + xf[i0 + 1].f * dxfi0 / dxi;
36
37    delete [] _d;
38    delete [] _x;
39    delete [] _c;
40    delete [] _b;
41    delete [] _a;
42    return fx;
43  }
```

Zamieszczona funkcja określona jest dla trzech parametrów, są to:

- `xf` – wskaźnik na tablicę struktur `XF`, zdefiniowaną tak jak na Listingu 2.5;
- `xf_size` – rozmiar tablicy `xf`;
- `x` – argument, dla którego funkcja zwróci wartość wielomianu interpolującego.

Wartością zwracaną jest wartość interpolująca funkcję `xf` dla argumentu `x`. Na początku rozważanej funkcji zdefiniowane zostają tablice które w pętli `for` inicjowane są wartościami zgodnie ze wzorem (2.32). Przypisanie polega na umieszczeniu w tablicy `_a` wartości stojącej przy $g''(x_{i-1})$, w tablicy `_b` wartości przy $g''(x_i)$ oraz w tablicy `_c` wartości przy $g''(x_{i+1})$. Tablica `_d` inicjowana jest wartościami wyliczonymi z prawej strony równości (2.32). Prócz tego w pętli wyliczany jest indeks `i0` stanowiący parametr i dla równania (2.31). W kolejnej części rozwiązywany jest układ równań względem niewiadomych $g''(x_i) : i = 1, 2, \ldots, n-1$ za pomocą algorytmu Thomasa (patrz Rozdział 7) oraz zgodnie z (2.34) i (2.35) przypisywane są zera dla $g''(x_0)$ i $g''(x_n)$. Ostatecznie na podstawie (2.31) i wyliczonych wartości obliczana jest wartość interpolująca funkcję `xf` w punkcie `x`.

2.6 Porównanie omówionych metod interpolacji

Jako przykład niech dane będą węzły $(x_i, x(x_i))$, funkcja $f(x_i)$ zdefiniowana jest jak następuje:

$$f(x_i) = 10\exp(-x_i)\cos(5x_i) : x_0 = 0; x_i = x_{i-1} + 0,4; x_i < 5; i = 1, 2, \ldots, n, \tag{2.42}$$

gdzie: $\exp(x)$ to funkcja eksponencjalna. Listing 2.7 przedstawia program wykorzystujący omówione metody interpolacji do oszacowania wartości funkcji $f(x)$ w punktach $x_0 = 0; x_j = x_{j-1} + 0,1; x_j < 5; i = 1, 2, \ldots, n$.

Listing 2.7: Program testujący omówione metody interpolacji.

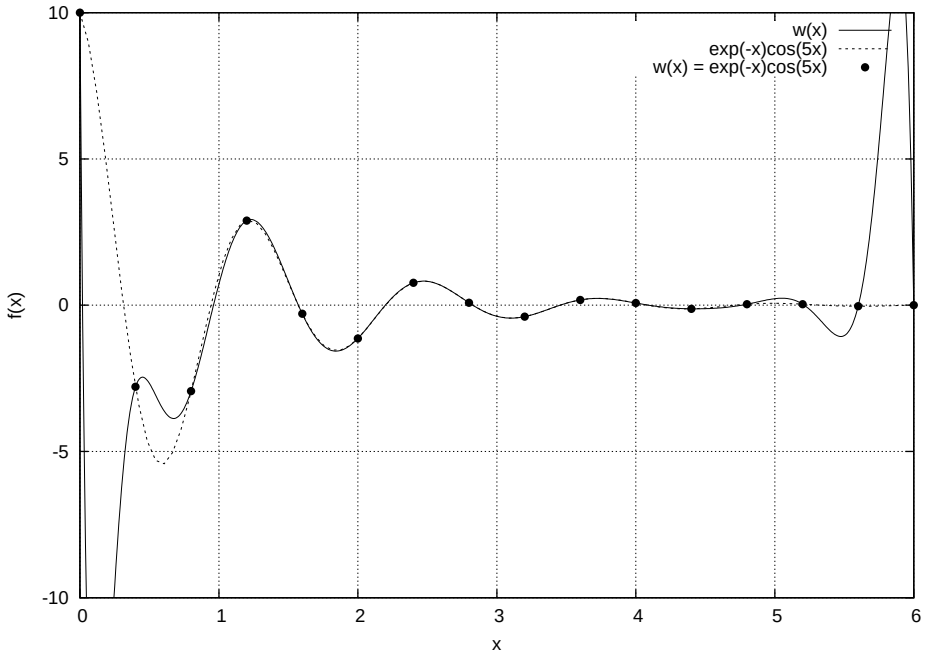
```
1 #include <iostream>
2 #include <cmath>
```

```
3  #include "interpolation.h"
4
5  const unsigned int size = 16;
6  const double step0 = 0.4;
7  const double step1 = 0.01;
8
9  int main()
10 {
11   XF *xf = new XF[size];
12   for (int i = 0; i < size; i++)
13     {
14       double x = i * step0;
15       xf[i].x = x;
16       xf[i].f = 10 * std::exp(-x) * std::cos(5 * x);
17     }
18
19   int i = 0;
20   for (double x = 0.0; x < (size - 1) * step0; x += step1)
21     {
22       std::cout << x << " "
23                 << lagrange_interpolation(xf, size, x) << " "
24                 << newton_interpolation(xf, size, x) << " "
25                 << spline_interpolation(xf, size, x);
26       if (i < size)
27         {
28           std::cout << " " << xf[i].x << " " << xf[i].f;
29           i++;
30         }
31       std::cout << std::endl;
32     }
33
34   delete [] xf;
35   return 0;
36 }
```

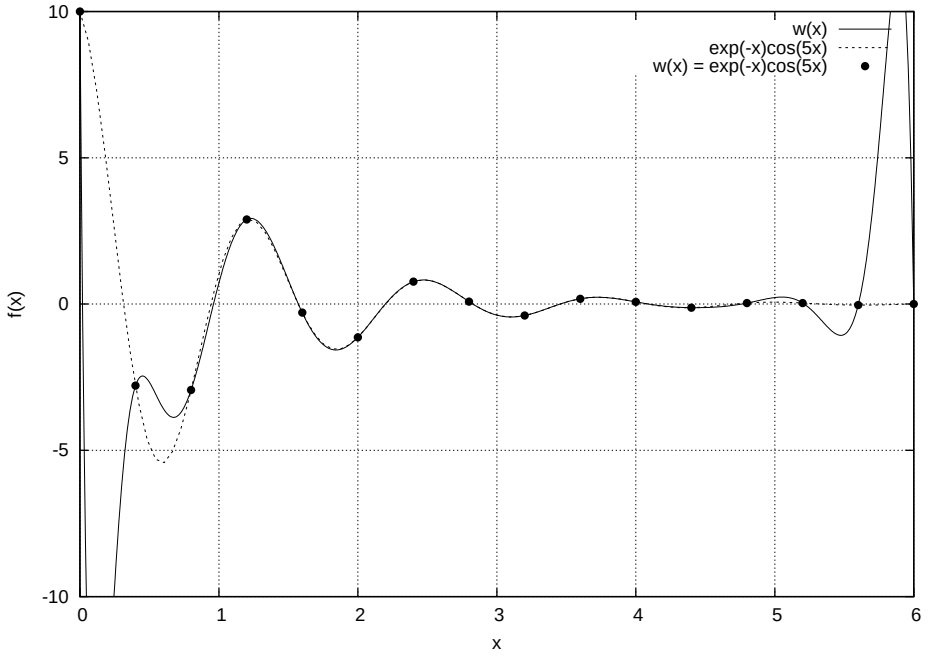
Na początku programu zdefiniowane zostały stałe: `size` – liczba węzłów, `step0` – różnica pomiędzy dwoma zmiennymi x_i, `step1` – różnica pomiędzy dwoma zmiennymi x_j. Program podzielony jest na dwie części. W pierwszej utworzona zostaje tablica struktur `XF` oraz przypisane zostają do niej wartości zgodnie z (2.42). Druga część wyświetla w sześciu kolumnach wyniki programu, odpowiednio x_j, $w_l(x_j)$, $w_n(x_j)$, $w_s(x_j)$, x_i, $f(x_i)$, gdzie: $w_l(x)$ – oszacowanie na podstawie interpolacji Lagrange'a, $w_n(x)$ – oszacowanie na podstawie interpolacji Newtona, $w_s(x)$ – oszacowanie na podstawie interpolacji funkcjami sklejanymi.

Na Rysunkach 2.5, 2.6, 2.7 przedstawione zostały wyniki programu z Listingu 2.7. Linią ciągłą zaznaczono funkcję interpolującą, linią przerywaną funkcję interpolowaną, punktami węzły.

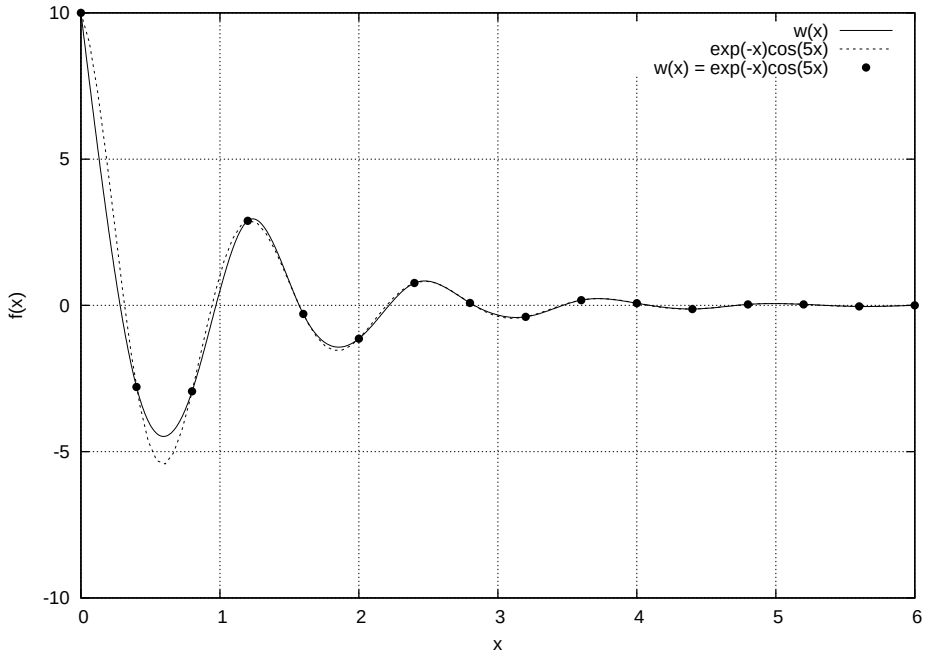
Jak widać na zamieszczonych rysunkach interpolacja wielomianami Lagrange'a i Newtona nie daje dobrych rezultatów przy krańcach przedziałów dla których określone są funkcje interpolowane. Jest to tzw. *efekt Rungego*, czyli pogorszenie interpolacji na krańcach przedziałów funkcji interpolowanej dla równoodległych węzłów. Efekt Rungego można uznać za paradoks w kontekście przytoczonego twierdzenia Weierstrassa, jednak wykazanie przyczyn takiego zjawiska wykracza poza ramy niniejszej pozycji. Rozwiązaniem tego problemu jest użycie innej metody interpolacji (np. funkcji sklejanych) bądź zagęszczenie ilości węzłów na krańcach przedziałów



Rysunek 2.5: Interpolacja Lagrange'a.



Rysunek 2.6: Interpolacja Newtona.



Rysunek 2.7: Interpolacja funkcjami sklejanymi.

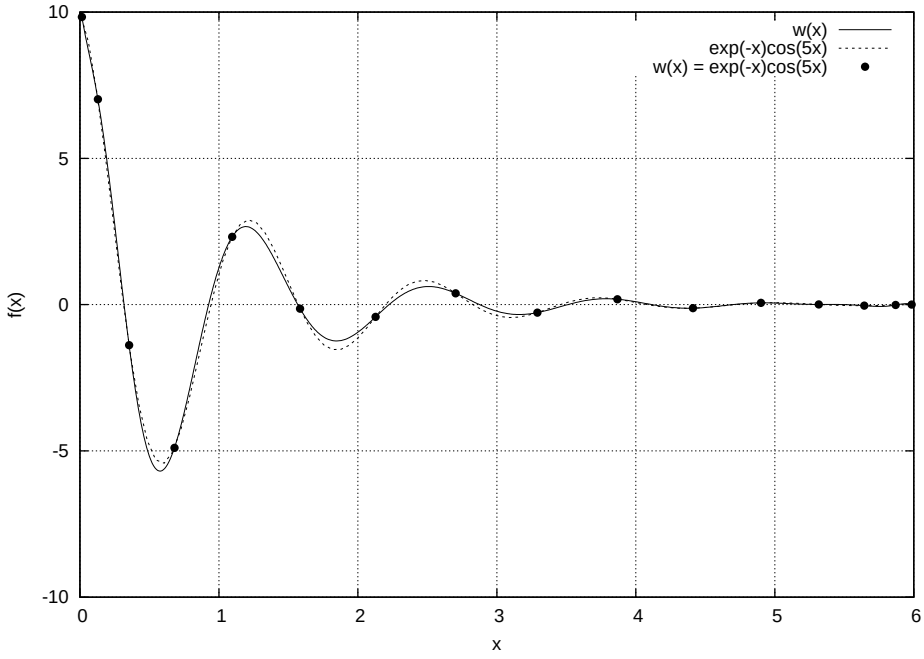
funkcji interpolowanej. Zagęszczenie ilości węzłów można uzyskać za pomocą omówionej interpolacji Czybyszewa. Jeśli w kodzie programu z Listingu 2.7 zamienić linię:

```
1   double x = i * step0;
```

na:

```
1   double x = -(6.0 - 0.0) / 2.0 * std::cos((2.0 * i + 1.0) / (2.0 * (
        size - 1) + 2.0) * 3.14) + (0.0 + 6.0) / 2.0;
```

to jako węzły interpolacji zostaną wygenerowane więzły Czybyszewa, których gęstość jest większa przy krańcach przedziałów a co za tym idzie nie są równo oddalone od siebie. Taki zabieg powoduje zniwelowanie efektu Rungego. Na Rysunku 2.8 zmieszczono interpolację funkcji (2.42) za pomocą wielomianów Lagrange'a i węzłów Czybyszewa.



Rysunek 2.8: Interpolacja Lagrange'a, węzły Czybyszewa.

Rozdział 3

Całkowanie numeryczne

3.1 Wstęp

Całkowanie numeryczne jest zbiorem metod pozwalających oszacować wartość zadanej całki oznaczonej. Mając daną funkcję $f(x)$, szukaną wartością jest:

$$\int_a^b f(x)dx,$$

gdzie a i b to przedziały całkowania. Funkcja $f(x)$ może być zadana w formie jawnego wzoru, bądź dyskretnych wartości zapisanych w tablicy. Intuicyjnie całkę oznaczoną funkcji $f(x)$ można rozumieć jako operator przypisujący funkcji $f(x)$ liczbę, która jest wartością reprezentującą pole powierzchni pod wykresem $f(x)$ i osi x. Taka interpretacja leży u podstaw metod całkowania numerycznego.

Większość metod całkowania korzysta z addytywności całki względem przedziałów całkowania:

$$\forall a \leqslant b \leqslant c : \int_a^c f(x)dx = \int_a^b f(x)dx + \int_b^c f(x)dx. \tag{3.1}$$

Własność ta pozwala podzielić przedział całkowania na mniejsze przedziały. Następnie dla każdego z przedziałów obliczana jest wartość całki. Wszystkie tak obliczone wartości są sumowane dając przybliżoną wartość szukanej całki.

3.2 Metoda prostokątów

Metoda prostokątów jest jedną z najprostszych metod służących do obliczania całek oznaczonych zadanej funkcji. Niech dana będzie całka oznaczona:

$$I = \int_a^b f(x)dx, \tag{3.2}$$

gdzie a i b to granice całkowania. Funkcja $f(x)$ jest interpolowana wielomianem stopnia zerowego (funkcją stałą), co prowadzi do:

$$I \approx (b-a)f(x^*), \tag{3.3}$$

gdzie: x^* to wartość z przedziału $[a;b]$. Nazwa „metoda prostokątów" bierze się stąd, że obliczane pole jest polem prostokąta. Zwykle wartość x^* jest punktem środkowym przedziału całkowania, a więc (metoda prostokątów zwana jest wtedy metodą punktu środkowego):

$$I \approx (b-a)f\left(\frac{a+b}{2}\right). \tag{3.4}$$

Wzór (3.4) należy zastosować do odpowiednio małych przedziałów tak, aby funkcja w każdym z tych przedziałów zmieniała się w niewielkim stopniu. Jeśli przedział całkowania zostanie podzielony na n równych części otrzymuje się wzór:

$$\int_a^b f(x)dx \approx \frac{b-a}{n}\sum_{i=0}^{n-1} f\left(a+\left(i+\frac{1}{2}\right)\frac{b-a}{n}\right). \tag{3.5}$$

Na Listingu 3.1 przedstawiony został kod funkcji `rectange_method` obliczającej wartość całki oznaczonej za pomocą metody prostokątów.

Listing 3.1: Metoda prostokątów.

```
1  double rectangle_method(double (*f)(double), double a, double b,
       unsigned int n)
2  {
3    if (n == 0)
4      return 0.0;
5    double sum = 0.0;
6    double h = (b - a) / n;
7    for (unsigned int i = 0; i < n; i++)
8      sum += f(a + (i + 0.5) * h);
9    return h * sum;
10 }
```

Funkcja `rectange_method` jako parametry przyjmuje:

- `f` – wskaźnik na funkcję podcałkową, która powinna być zakodowana jawnie;
- `a` – początek przedziału całkowania;
- `b` – koniec przedziału całkowania;
- `n` – nieujemna wartość reprezentująca liczbę, na jaką zostanie podzielony przedział całkowania.

Wartością zwracaną jest oszacowana wartość całki funkcji `f` w przedziale $[a, b]$.

3.3 Metoda trapezów

Bardziej dokładną metodą przybliżającą zadaną całkę jest metoda trapezów, która interpoluje funkcję podcałkową za pomocą wielomianu pierwszego stopnia wykorzystując punkty będące granicą całkowania:

$$\int_a^b f(x)dx \approx (b-a)\frac{f(a)+f(b)}{2}. \tag{3.6}$$

Obliczanym polem jest pole trapezu, stąd nazwa tej metody. Podobnie jak dla metody prostokątów, przedział całkowania należy podzielić na n części i dla każdej z tych części obliczyć całkę ze wzoru (3.6). Prowadzi to do następującej zależności:

$$\int_a^b f(x)dx \approx \frac{b-a}{2n}\sum_{i=0}^{n-1}\left(f\left(a+i\frac{b-a}{n}\right)+f\left(a+(i+1)\frac{b-a}{n}\right)\right). \tag{3.7}$$

Obliczenie wartości całki ze wzoru (3.7) wymaga w każdym kroku iteracji wyznaczenia dwa razy wartości funkcji podcałkowej, co wiąże się z dużym kosztem tego algorytmu – każdorazowe wywołanie funkcji może wymagać wykonania algorytmu o dużej złożoności. Algorytm ten można uprościć, niech $x_i = a + i(b-a)/n$:

$$\begin{aligned}\int_a^b f(x)dx &\approx \frac{b-a}{2n}(f(x_0)+f(x_1)+f(x_1)+f(x_2)+\dots\\ &\quad + f(x_{n-1})+f(x_n))\\ &= \frac{b-a}{2n}(f(x_0)+2f(x_1)+\dots+2f(x_{x-1})+f(x_n))\\ &= \frac{b-a}{n}\left(\frac{f(a)}{2}+\frac{f(b)}{2}+\sum_{i=1}^{n-1}f\left(a+i\frac{b-a}{n}\right)\right).\end{aligned} \tag{3.8}$$

Jak widać wykorzystanie zależności (3.8) wiąże się tylko z jednokrotnym wyznaczeniem wartości funkcji podcałkowej w każdym kroku iteracji.

Na listingu 3.2 zamieszczony został kod funkcji obliczającej wartość całki oznaczonej za pomocą metody trapezów.

Listing 3.2: Metoda trapezów.

```
1  double trapezoidal_method(double (*f)(double), double a, double b,
       unsigned int n)
2  {
3    if (n == 0)
4      return 0.0;
5    double sum = 0.0;
6    double h = (b - a) / n;
7    for (unsigned int i = 1; i < n; i++)
8      sum += f(a + i * h);
9    return h * (0.5 * f(a) + 0.5 * f(b) + sum);
10 }
```

Funkcja `trapezoidal_method` wymaga przekazania parametrów opisanych na następującej liście:

- `f` – wskaźnik na funkcję podcałkową, która powinna być zakodowana jawnie;
- `a` – początek przedziału całkowania;
- `b` – koniec przedziału całkowania;
- `n` – nieujemna wartość reprezentująca liczbę, na jaką zostanie podzielony przedział całkowania.

Wartością zwracaną jest oszacowana wartość całki funkcji `f` w przedziale $[a, b]$.

3.4 Metoda Simpsona

W metodzie Simpsona funkcję podcałkową $f(x)$ przybliża się wielomianem interpolacyjnym Lagrange'a, gdzie węzłami są punkty reprezentujące początek, środek i koniec przedziału całkowania. Odpowiednio x_0, x_1, x_2, gdzie $x_1 = (x_2 - x_0)/2$. Wielomian interpolacyjny można zapisać jak następuje:

$$f(x) \approx w(x) = f(x_0)\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + f(x_1)\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + f(x_2)\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} \tag{3.9}$$

Przybliżona wartość całki będzie więc równa:

$$\int_{x_0}^{x_2} f(x)dx \approx \int_{x_0}^{x_2} w(x)dx = \int_{x_0}^{x_2} \left(f(x_0)\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + f(x_1)\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + f(x_2)\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} \right) dx \tag{3.10}$$

Ponieważ odległości pomiędzy węzłami są sobie równe, wartość tą oznaczmy przez h, można zapisać $x = x_0 + th$, dla $t = 0, 1, 2$ oraz $x_i = x_0 + ih$ dla $i = 0, 1, 2$. Dokonując podstawienia oraz zmiany granicy całkowania otrzymuje się:

$$\begin{aligned}
\int_{x_0}^{x_2} f(x)dx &\approx \int_0^2 \left(f(0)\frac{(th-h)(th-2h)}{(-h)(-2h)} + f(1)\frac{th(th-2h)}{h(-h)} + f(2)\frac{th(th-h)}{(2h)(h)} \right) h\,dt \\
&= \frac{h}{2}\int_0^2 \Big(f(0)(t-1)(t-2) - 2f(1)t(t-2) + f(2)t(t-1) \Big) dt \\
&= \frac{h}{2}\int_0^2 \Big(f(0)(t^2-3t+2) - 2f(1)(t^2-2t) + f(2)(t^2-t) \Big) dt \\
&= \frac{h}{2}\left(f(0)\left(\frac{t^3}{3} - \frac{3t^2}{2} - 2t\right)\Big|_{t=0}^{t=2} - 2f(1)\left(\frac{t^3}{3} - \frac{2t^2}{2}\right)\Big|_{t=0}^{t=2} \right. \\
&\quad \left. + f(2)\left(\frac{t^3}{3} - \frac{t^2}{2}\right)\Big|_{t=0}^{t=2} \right)
\end{aligned}$$

$$= \frac{h}{2}\left(f(0)\frac{2}{3} - 2f(1)\frac{-4}{3} + f(2)\frac{2}{3}\right)$$
$$= \frac{h}{3}\left(f(0) + 4f(1) + f(2)\right) \tag{3.11}$$

Przyjmując, że $x_0 = a, x_2 = b$, oraz $h = (b-a)/2$, otrzymuje się zależność szacującą całkę funkcji $f(x)$ w przedziale $[a; b]$:

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right). \tag{3.12}$$

Podobnie jak dla omawianych wcześniej metod, aby zwiększyć dokładność szacowanej całki, należy przedział $[a; b]$ podzielić na n części. Dla $h = (b-a)/n$ otrzymuje się:

$$\begin{aligned}\int_a^b f(x)dx &\approx \sum_{i=0}^{n-1}\int_{a+ih}^{a+(i+1)h} f(x)dx \\ &= \sum_{i=0}^{n-1}\frac{h}{6}\left(f(a+ih) + 4f\left(\frac{2a+2ih+h}{2}\right) + f(a+(i+1)h)\right) \\ &= \frac{h}{6}\sum_{i=0}^{n-1}\left(f(a+ih) + 4f\left(a+ih+\frac{h}{2}\right) + f(a+ih+h)\right) \\ &= \frac{h}{6}\Bigg(f(a) + 4f\left(a+\frac{h}{2}\right) + f(a+h) + f(a+h) \\ &+ 4f\left(a+h+\frac{h}{2}\right) + f(a+2h) + \cdots + f(a+(n-1)h) \\ &+ 4f\left(a+(n-1)h+\frac{h}{2}\right) + f(a+nh)\Bigg)\end{aligned} \tag{3.13}$$

Co ostatecznie prowadzi do:

$$\int_a^b f(x)dx \approx \frac{h}{6}\left(f(a) + 4\sum_{i=0}^{n-1} f\left(a+ih+\frac{h}{2}\right) + 2\sum_{i=1}^{n-1} f(a+ih) + f(b)\right). \tag{3.14}$$

Kod funkcji obliczającej zadaną całkę metodą Simpsona zamieszczony został na Listingu 3.3.

Listing 3.3: Metoda Simpsona.

```
1 double simpson_method(double (*f)(double), double a, double b,
      unsigned int n)
2 {
3   if (n == 0)
4     return 0.0;
5   double sum2 = 0.0;
6   double sum4 = 0.0;
7   double h = (b - a) / n;
8   double h2 = h / 2.0;
9   for (unsigned int i = 0; i < n; i++)
```

```
10      sum4 += f(a + i * h + h2);
11    for (unsigned int i = 1; i < n; i++)
12      sum2 += f(a + i * h);
13    return h / 6.0 * (f(a) + 4.0 * sum4 + 2.0 * sum2 + f(b));
14  }
```

Funkcja z Listingu 3.3 wymaga przekazania następujących parametrów:

- `f` – wskaźnik na funkcję podcałkową, która powinna być zakodowana jawnie;
- `a` – początek przedziału całkowania;
- `b` – koniec przedziału całkowania;
- `n` – nieujemna wartość reprezentująca liczbę, na jaką zostanie podzielony przedział całkowania.

Wartością zwracaną jest oszacowana wartość całki funkcji `f` w przedziale $[a, b]$.

3.5 Porównanie omówionych metod całkowania

Jako przykład niech dany będzie wzór na pole koła o wartości promienia równej jedności, dokładana wartość pola tego koła równa jest liczbie π:

$$2\int_{-1}^{1}\sqrt{1-x^2}dx = \pi \approx 3,1415926535\ldots\,. \tag{3.15}$$

Program obliczający wartość liczby π na podstawie przytoczonego wzoru i poznanych metod zamieszczony został na Listingu 3.4.

Listing 3.4: Program testujący metodę prostokątów.

```
1  #include <iostream>
2  #include <iomanip>
3  #include <cmath>
4  #include "integration.h"
5
6  const unsigned int n = 1e+6;
7
8  double f(double);
9
10 int main()
11 {
12   for (unsigned int i = 10; i <= n; i *= 10)
13     std::cout << i << ", "
14               << std::setprecision(11) << 2 * rectangle_method(f, -1,
                       1, i) << ", "
15               << std::setprecision(11) << 2 * trapezoidal_method(f,
                       -1, 1, i) << ", "
16               << std::setprecision(11) << 2 * simpson_method(f, -1, 1,
                        i) << ", "
17               << std::endl;
18   return 0;
19 }
20
21 double f(double x)
```

```
22 {
23   return std::sqrt(1 - x * x);
24 }
```

Dane zebrane za pomocą programu z Listingu 3.4 zostały zestawione w Tabeli 3.1. Warto zwrócić uwagę, że dokładność obliczeń zależy do charakteru funkcji podcałkowej, im lepiej założenia metody wpasowują się w charakter owej funkcji, tym dokładniejszy wynik obliczanej całki.

Tabela 3.1: Dane zestawiające wyniki oszacowań całki (3.15) dla omówionych metod numerycznych.

i	Metoda prostokątów	Metoda trapezów	Metoda Simpsona
10	3,1719878236	3,0370488289	3,1270081587
100	3,1425655525	3,1382685111	3,1411332053
1000	3,1416234568	3,141487477	3,1415781302
10000	3,1415936278	3,1415893274	3,1415921943
100000	3,1415926844	3,1415925484	3,1415926391
1000000	3,1415926546	3,1415926503	3,1415926531

Rozdział 4

Różniczkowanie numeryczne

4.1 Wstęp

Różniczkowanie numeryczne to zbiór metod pozwalających oszacować wartość pochodnej funkcji w danym punkcie. Wszędzie tam, gdzie postać funkcji nie jest znana analitycznie lub jej postać jest bardzo skomplikowana oraz dla funkcji, której wartości znane są tylko dla dyskretnego zbioru argumentów, różniczkowanie numeryczne znajduje zastosowanie.

4.2 Metoda Newtona

Metoda Newtona jest najprostszym sposobem obliczenia różniczki funkcji $f(x)$. Wykorzystuje się tutaj definicję pochodnej funkcji, która przybliżana jest ilorazem różnicowym:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \approx \frac{f(x+h) - f(x)}{h}. \tag{4.1}$$

Dla dostatecznie małego h wartość ilorazu różnicowego przybliża wartość pochodnej w punkcie x.

Przybliżenie pochodnej ilorazem różnicowym (4.1) może powodować problemy. Wiąże się to z koniecznością znajomości wartości funkcji w punktach x i $x+h$ dla wartości pochodnej w punkcie x. Jeśli funkcja $f(x)$ nie jest dana jawnie, a wyliczana jest w procesie iteracyjnym, jej wartość w punkcie $x+h$ może nie być znana lub może wymagać dodatkowych zabiegów umożliwiających jej oszacowanie. Przy założeniu, że obliczając $f(x)$ znane są wartości dla wcześniejszych argumentów, warto rozważyć przybliżenie pochodnej ilorazem różnicowym w postaci:

$$\frac{df(x)}{dx} \approx \frac{f(x) - f(x-h)}{h}. \tag{4.2}$$

Zależność (4.1) określa się często *różniczkowaniem w przód*, natomiast (4.2) *różniczkowaniem w tył*. Podobnie pochodną można przybliżać ilorazem różnicowym

centralnym:

$$\frac{df(x)}{dx} \approx \frac{f(x + \frac{1}{2}h) - f(x - \frac{1}{2}h)}{h}. \tag{4.3}$$

Warto zwrócić uwagę, że dobór metody różniczkowania może mieć wpływ na stabilność rozwiązań.

Listing 4.1 zawiera kod funkcji obliczającej pochodną na podstawie zależności (4.1).

Listing 4.1: Różniczkowanie numeryczne metodą Newtona.

```
1 double newton_differentiation(double (*f)(double), double x, double h)
2 {
3   return (f(x + h) - f(x)) / h;
4 }
```

Argumenty funkcji `newton_differentiation` to:

- `f` – funkcja, dla której zostanie policzona pochodna;
- `x` – punkt, w którym zostanie policzona pochodna;
- `h` – krok różniczkowania.

Wartością zwracaną jest wartość pochodnej funkcji `f` w punkcie `x` dla kroku różniczkowania równego `h`. Tak prosta funkcja powinna mieć prototyp opatrzony słowem kluczowym **`inline`**, aby potencjalnie zwiększyć wydajność wywołań owej funkcji. W analogiczny sposób można zakodować metody (4.2) i (4.3).

Rozdział 5

Równania różniczkowe zwyczajne

5.1 Wstęp

Równania różniczkowe odgrywają dużą rolę w opisie zjawisk i procesów fizycznych. Wszędzie tam, gdzie istnieje potrzeba modelowania i symulacji układów i zjawisk dynamicznych, równania różniczkowe znajdują zastosowanie. Niestety w wielu przypadkach analityczne rozwiązanie równania różniczkowego nie jest znane, stąd metody numeryczne są niemal nieodłączną częścią teorii równań różniczkowych. W metodach numerycznych szacujących rozwiązanie równania różniczkowego zakłada się, że zmienna niezależna występuje w postaci dyskretnych wartości. Oczywiście, konsekwencją tego faktu jest otrzymywanie przybliżonych rozwiązań równań różniczkowych.

Równanie różniczkowe zwyczajne jest równaniem wyznaczającym zależność pomiędzy nieznaną funkcją i pochodnymi tej funkcji. Rozwiązanie polega na znalezieniu wspomnianej funkcji. Metody numeryczne stosowane do rozwiązania równania różniczkowego zależą w głównej mierze od typu równania. Równaniem różniczkowym zwyczajnym będziemy nazywać równanie typu:

$$F\left(t, x(t), \frac{dx(t)}{dt}, \frac{d^2x(t)}{dt^2}, \cdots, \frac{d^nx(t)}{dt^n}\right) = 0, \tag{5.1}$$

gdzie: $x(t) : \mathbb{R} \to \mathbb{R}^k$ — szukana funkcja (w ogólności funkcja wektorowa); $F : \mathbb{R}^{(n+1)k+1} \to \mathbb{R}^k$. Równanie (5.1) nazywa się równaniem różniczkowym n-tego rzędu. Takie równanie często da się rozwikłać względem $d^nx(t)/dt^n$, otrzymując postać jawną (równanie (5.1) to postać niejawna):

$$\frac{d^nx(t)}{dt^n} = f\left(t, x(t), \frac{dx(t)}{dt}, \frac{d^2x(t)}{dt^2}, \ldots, \frac{d^{n-1}x(t)}{dt^{n-1}}\right), \tag{5.2}$$

którą łatwiej rozwiązać w sposób numeryczny.

Jak zostało wspomniane, użycie metody numerycznej zależy od równania róż-

niczkowego, które należy rozwiązać. Jednym z kryteriów jest stopień równania. Warto zwrócić uwagę, że każde równanie stopnia n da się sprowadzić do układu n równań różniczkowych:

$$\begin{aligned} x(t) &= x_1, \\ \frac{dx_1}{dt} &= x_2, \\ \frac{dx_2}{dt} &= x_3, \\ &\vdots \\ \frac{dx_{n-1}}{dt} &= x_n, \\ \frac{dx_n}{dt} &= f(t, x_1, x_2, \ldots, x_n), \end{aligned}$$

gdzie $f : \mathbb{R} \times \mathbb{R}^k \to \mathbb{R}^k$ jest znaną funkcją. Dzięki temu dysponując metodą rozwiązującą równanie różniczkowe pierwszego rzędu, można rozwiązać równanie różniczkowe n-tego rzędu, sprowadzając je do układu równań różniczkowych pierwszego rzędu. Dla potrzeb dalszych rozważań niech $n = 1$, a więc:

$$\frac{dx(t)}{dt} = f\Big(t, x(t)\Big). \tag{5.3}$$

Rozwiązaniem równania (5.3) jest dowolna funkcja $\psi(t)$ spełniająca zależność:

$$\frac{d\psi(t)}{dt} = f\Big(t, \psi(t)\Big). \tag{5.4}$$

Rozwiązując równania różniczkowe często znana jest wartość szukanej funkcji dla pewnego argumentu. Argument ten zwany jest zwykle początkowym, a więc otrzymuje się *równanie różniczkowe z warunkiem początkowym*. Jeśli dla równania (5.3) znany jest warunek początkowy, $x(t_0) = x_0$ to taki problem zwie się zagadnieniem Cauchy'ego:

$$\begin{cases} x(t_0) &= x_0 \\ \dfrac{dx(t)}{dt} &= f\Big(t, x(t)\Big) \end{cases}. \tag{5.5}$$

Jeśli rozwiązanie równania różniczkowego $\psi(t)$ spełnia dodatkowo warunek $\psi(t_0) = x_0$, to jest ono również rozwiązaniem zagadnienia Cauchy'ego. Aby rozwiązać równanie różniczkowe za pomocą większości metod numerycznych, potrzebna jest znajomość warunku początkowego.

Metodę numeryczną nazywa się *samo-startującą*, jeśli mając daną tylko jedną wartość początkową (zagadnienie Cauchy'ego) można za jej pomocą rozwiązać równanie różniczkowe. Przeciwieństwem są metody samo-niestartujące, które potrzebują więcej niż jednej wartości początkowej aby rozwiązać równanie.

5.2 Metoda Eulera

Niech rozważaniom poddane zostanie zagadnienie Cauchy'ego pierwszego rzędu (5.5). Korzystając z definicji pochodnej można zapisać:

$$\frac{dx(t)}{dt} \equiv \lim_{h\to 0}\frac{x(t+h)-x(t)}{h} \approx \frac{x(t+h)-x(t)}{h}. \tag{5.6}$$

Przekształcając to wyrażenie otrzymuje się wzór na kolejne wartości przybliżające funkcję $x(t)$ (dokładność przybliżenia będzie wzrastać dla $h \to 0$):

$$x(t+\Delta t) = x(t) + \Delta t \cdot f\Big(t, x(t)\Big), \tag{5.7}$$

gdzie $f(t, x(t)) = dx(t)/dt$ oraz $h = \Delta t$. Znając rozwiązanie w t_0: $x(t_0) = x_0$ można wyznaczyć kolejne przybliżenia szukanej funkcji w punktach $\{t_0 + n \cdot \Delta t : n = 1, 2, \dots\}$. Metoda Eulera jest samo-startująca.

Metoda Eulera jest najprostszym sposobem na numeryczne rozwiązanie równania różniczkowego pierwszego rzędu, ceną jaką trzeba zapłacić za prostotę jest dokładność obliczeń. Metoda ta generuje znaczne błędy przy przybliżaniu szukanej funkcji w porównaniu z innymi metodami.

Funkcję rozwiązującą zadane równanie różniczkowe za pomocą metody Eulera przedstawiono na Listingu 5.1.

Listing 5.1: Implementacja metody Eulera.

```
1 void euler(double (*f)(double t, double x), double *data, unsigned int
        data_size, double t0, double dt, double x0)
2 {
3   if (data_size < 1)
4     return;
5   data[0] = x0;
6   double t = t0;
7   for (int i = 1; i < data_size; i++, t += dt)
8     data[i] = data[i - 1] + dt * f(t, data[i - 1]);
9 }
```

Na początku funkcji sprawdzane jest czy przekazana tablica mająca stanowić rozwiązanie zadanego równania jest większa lub równa jedności – pierwszy element otrzymuje wartość początkową, dlatego prowadzenie obliczeń ma sens, jeśli wielkość owej tablicy jest większa lub równa jedności. W kolejnych krokach realizowany jest algorytm (5.7). Argumenty dla funkcji `euler` należy interpretować jak następuje:

- `f` – wskaźnik na pierwszą pochodną szukanej funkcji;
- `data` – wskaźnik na tablicę mającą stanowić rozwiązanie zadanego równania (kolejne wartości szukanej funkcji);
- `data_size` – rozmiar tablicy `data`;
- `t0` – początkowy czas;
- `dt` – krok całkowania;
- `x0` – początkowa wartość szukanej funkcji.

5.3 Metoda punktu środkowego

Metoda punktu środkowego jest modyfikacją metody Eulera. W metodzie Eulera mając daną wartość $x(t)$ szacuje się wartość $x(t+\Delta t)$ wyliczając pochodną w punkcie t. W metodzie punktu środkowego, aby zwiększyć dokładność, pochodną wylicza się w punkcie środkowym przedziału $[t; t+\Delta t]$. Korzystając z definicji pochodnej za pomocą ilorazu różnicowego centralnego:

$$\begin{aligned}\frac{dx}{dt}\left(t+\frac{1}{2}\Delta t\right) &\approx \frac{x(t+\frac{1}{2}\Delta t+\frac{1}{2}\Delta t)-x(t+\frac{1}{2}\Delta t-\frac{1}{2}\Delta t)}{\Delta t}\\ &= \frac{x(t+\Delta t)-x(t)}{\Delta t},\end{aligned} \tag{5.8}$$

można oszacować wartość $x(t+\Delta t)$:

$$x(t+\Delta t) \approx x(t)+\Delta t\cdot f\left(t+\frac{1}{2}\Delta t, x\left(t+\frac{1}{2}\Delta t\right)\right), \tag{5.9}$$

gdzie: $f(t+1/2\ \Delta t, x(t+1/2\ \Delta t)) = dx(t+1/2\ \Delta t)/dt$. Równanie (5.9) nie określa jeszcze wartości $x(t+\Delta t)$, ponieważ wartość funkcji $x(t+1/2\ \Delta t)$ nie jest znana. Korzystając z definicji pochodnej można zapisać:

$$\frac{dx(t)}{dt} \approx \frac{x(t+\frac{1}{2}\Delta t)-x(t)}{\frac{1}{2}\Delta t}, \tag{5.10}$$

przekształcając to równanie tak aby otrzymać $x(t+1/2\ \Delta t)$ oraz wstawiając otrzymaną wartość do (5.9) otrzymuje się:

$$x(t+\Delta t) \approx x(t)+\Delta t\cdot f\left(t+\frac{1}{2}\Delta t, x(t)+\frac{1}{2}\Delta t\cdot f\Big(t, x(t)\Big)\right), \tag{5.11}$$

gdzie: $f(t, x(t)) = dx(t)/dx$.

Na Listingu 5.2 zamieszczono kod funkcji rozwiązującej równania różniczkowe pierwszego rzędu za pomocą metody punktu środkowego.

Listing 5.2: Funkcja implementująca metodę punktu środkowego.

```
1 void midpoint(double (*f)(double t, double x), double *data, unsigned
      int data_size, double t0, double dt, double x0)
2 {
3   if (data_size < 1)
4     return;
5   data[0] = x0;
6   double t = t0;
7   for (int i = 1; i < data_size; i++, t += dt)
8     data[i] = data[i - 1] + dt * f(t + 0.5 * dt, data[i - 1] + 0.5 *
          dt * f(t, data[i - 1]));
9 }
```

Zamieszczona funkcja wymaga przekazania parametrów opisanych na następującej liście:

- `f` – wskaźnik na pierwszą pochodną szukanej funkcji;
- `data` – wskaźnik na tablicę mającą stanowić rozwiązanie zadanego równania (kolejne wartości szukanej funkcji);
- `data_size` – rozmiar tablicy `data`;
- `t0` – początkowy czas;
- `dt` – krok całkowania;
- `x0` – początkowa wartość szukanej funkcji.

5.4 Metoda Rungego-Kutty

Wzory typu Rungego-Kutty są najczęściej używane do rozwiązywania równań różniczkowych zwyczajnych. Zalety rozważanej metody są następujące:

- jest łatwa do zakodowania,
- jest zazwyczaj stabilna,
- jest samo-startująca.

Podobnie jak w metodzie Eulera, zakładamy że znamy $x(t)$ i chcemy wyznaczyć przybliżoną wartość $x(t+\Delta t)$. Metoda Rungego-Kutty polega na obliczeniu wartości $f(t, x(t))$ w pewnych szczególnie dobranych punktach leżących w pobliżu krzywej rozwiązania w przedziale $(t; t+\Delta t)$ i zastosowaniu odpowiednio dobranego wzoru zbudowanego z kombinacji tych wartości, tak aby dobrze oszacować przyrost $x(t+\Delta t) - x(t)$.

Wyróżnia się metody n-tego rzędu, gdzie rząd określa ile razy obliczamy funkcję $f(t, x(t))$. Najczęściej używana jest formuła Runge-Kutty czwartego rzędu. Wzory metody Rungego-Kutty czwartego rzędu mają postać:

$$\begin{aligned} k_1 &= \Delta t \cdot f(t, x(t)), \\ k_2 &= \Delta t \cdot f\left(t + \frac{1}{2}\Delta t, x(t) + \frac{1}{2}k_1\right), \\ k_3 &= \Delta t \cdot f\left(t + \frac{1}{2}\Delta t, x(t) + \frac{1}{2}k_2\right), \\ k_4 &= \Delta t \cdot f\left(t + \Delta t, x(t) + k_3\right), \\ x(t+\Delta t) &= x(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \end{aligned} \tag{5.12}$$

gdzie: Δt jest krokiem iteracji.

Przedstawioną metodę Eulera i punktu środkowego można uważać za metodę Rungego-Kutty odpowiednio pierwszego i drugiego rzędu. Metodę pierwszego rzędu można zapisać jako:

$$x(t+\Delta t) = x(t) + k_1, \tag{5.13}$$

odpowiednio metodę drugiego rzędu:

$$x(t + \Delta t) = x(t) + k_2, \tag{5.14}$$

dla współczynników k_1 i k_2 zdefiniowanych w (5.12).

Funkcja rozwiązująca równanie różniczkowe metodą Rungego-Kutty czwartego rzędu ma postać przedstawioną na Listingu 5.3.

Listing 5.3: Funkcja implementująca metodę Rungego-Kutty.

```
1  void rk4(double (*f)(double t, double x), double *data, unsigned int
       data_size, double t0, double dt, double x0)
2  {
3    if (data_size < 1)
4      return;
5    data[0] = x0;
6    double t = t0;
7    for (int i = 1; i < data_size; i++, t += dt)
8      {
9        double k1 = dt * f(t, data[i - 1]);
10       double k2 = dt * f(t + 0.5 * dt, data[i - 1] + 0.5 * k1);
11       double k3 = dt * f(t + 0.5 * dt, data[i - 1] + 0.5 * k2);
12       double k4 = dt * f(t + dt, data[i - 1] + k3);
13       data[i] = data[i - 1] + 1.0 / 6.0 * (k1 + 2 * k2 + 2 * k3 + k4);
14     }
15 }
```

Funkcja `rk4` określona jest dla następujących parametrów:

- `f` – wskaźnik na pierwszą pochodną szukanej funkcji;
- `data` – wskaźnik na tablicę mającą stanowić rozwiązanie zadanego równania (kolejne wartości szukanej funkcji);
- `data_size` – rozmiar tablicy `data`;
- `t0` – początkowy czas;
- `dt` – krok całkowania;
- `x0` – początkowa wartość szukanej funkcji.

5.5 Porównanie metod: Eulera, punktu środkowego i Rungego-Kutty

Jako przykład niech dane będzie równianie różniczkowe pierwszego rzędu określone jak następuje:

$$\frac{dx(t)}{dt} = x(t) + t. \tag{5.15}$$

Rozwiązaniem równania (5.15) jest funkcja:

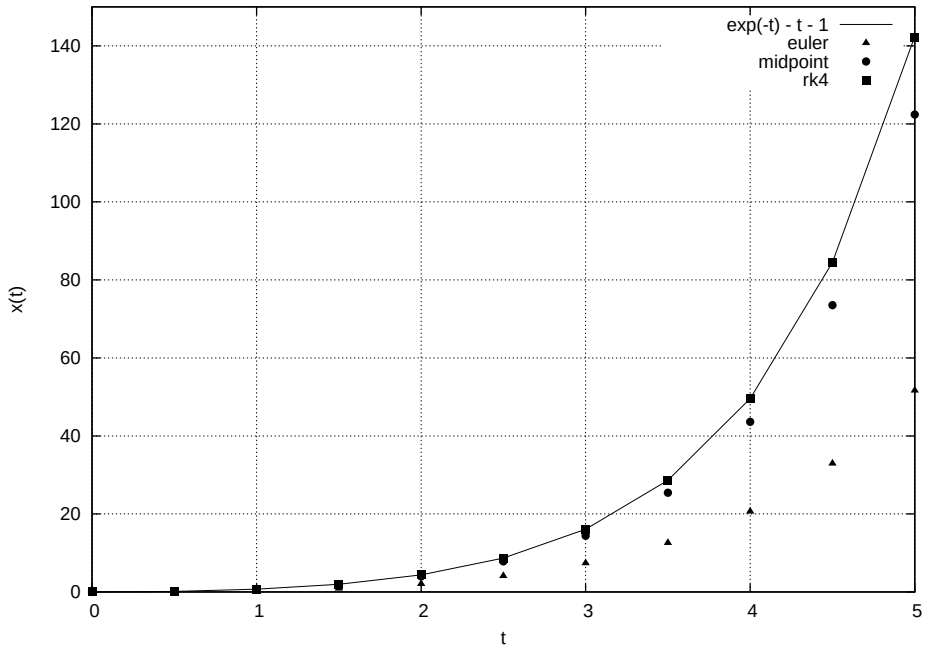
$$x(t) = ce^t - t - 1, \tag{5.16}$$

gdzie c to stała. Program rozwiązujący równanie (5.15) za pomocą metody Eulera, punktu środkowego i metody Rungego-Kutty rzędu czwartego zaprezentowany jest na Listingu 5.4:

Listing 5.4: Program testujący omówione metody.

```
#include <iostream>
#include <cmath>
#include "ode.h"

const int data_size = 50;

double df(double t, double x);
double f(double t);

int main()
{
  double *euler_data = new double[data_size];
  double *midpoint_data = new double[data_size];
  double *rk4_data = new double[data_size];
  double t0 = 0;
  double dt = 0.5;
  double x0 = f(t0);
  euler(df, euler_data, data_size, t0, dt, x0);
  midpoint(df, midpoint_data, data_size, t0, dt, x0);
  rk4(df, rk4_data, data_size, t0, dt, x0);
  for (int i = 0; i < data_size; i++, t0 += dt)
    std::cout << t0 << " "
              << f(t0) << " "
              << euler_data[i] << " "
              << midpoint_data[i] << " "
              << rk4_data[i] << std::endl;
  delete [] rk4_data;
  delete [] midpoint_data;
  delete [] euler_data;
}

double df(double t, double x)
{
  return x + t;
}

double f(double t)
{
  return std::exp(t) - t - 1;
}
```

W zaprezentowanym programie przyjęto, że stała $c = 1$. Rysunek 5.1 przedstawia wykresy funkcji stanowiących rozwiązania numeryczne i analityczne. Jak widać na wspomnianym rysunku, prostota metody Eulera ma swoje konsekwencje w dokładności rozwiązania. Aby uzyskać zadowalające wyniki, krok całkowania dla tej metody powinien być relatywnie mały (mniejszy niż w innych metodach). Metoda punktu środkowego daje znaczne lepsze oszacowanie szukanej funkcji przy niewiele większym koszcie implementacji. Metoda Rungego-Kutty czwartego rzędu daje bardzo dobre przybliżenie rozwiązania analitycznego nawet przy relatywnie dużym kroku całkowania ($\Delta t = 0,5$).



Rysunek 5.1: Rozwiązanie analityczne – linia ciągła, metoda Eulera – trójkąty, metoda punktu środkowego – koła, metoda Rungego-Kutty czwartego rzędu – kwadraty.

5.6 Metoda Verleta

5.6.1 Wiadomości wstępne

Metoda Verleta jest algorytmem służącym do rozwiązywania równań różniczkowych zwyczajnych drugiego rzędu. Metoda ta jest zwykle stosowana w fizyce do rozwiązywania Newtonowskich równań ruchu, czyli do szacowania położeń i prędkości ciał w funkcji czasu. W algorytmie Verleta zakłada się, że położenie ciała nie zależy od prędkości, przy tym założeniu równanie ruchu Newtona można zapisać:

$$\frac{d^2x(t)}{dt^2} = a\Big(t, x(t)\Big), \tag{5.17}$$

gdzie: $a(\cdot)$ to funkcja, która jest zwykle interpretowana jako przyspieszenie ciała w chwili t i o położeniu $x(t)$.

5.6.2 Podstawowy algorytm Verleta

Równanie (5.17) może być przybliżone przez iloraz różnicowy centralny, jak następuje:

$$\frac{d^2x(t)}{dt^2} = \frac{d}{dt}\frac{dx(t)}{dt} \approx \frac{d}{dt}\frac{x(t+\frac{1}{2}\Delta t) - x(t-\frac{1}{2}\Delta t)}{\Delta t}$$

$$\approx \frac{\frac{x(t+\Delta t)-x(t)}{\Delta t} - \frac{x(t)-x(t-\Delta t)}{\Delta t}}{\Delta t}$$
$$\approx \frac{x(t+\Delta t)-2x(t)+x(t-\Delta t)}{\Delta t^2}. \tag{5.18}$$

Korzystając z (5.17) i (5.18) otrzymuje się wzór na kolejną wartość położenia w chwili $t+\Delta t$:

$$x(t+\Delta t) \approx 2x(t) - x(t-\Delta t) + a\Big(t, x(t)\Big)\Delta t^2. \tag{5.19}$$

Jak widać podstawowy algorytm Verleta nie jest samo-startujący, wymaga on wartości $x(t-\Delta t)$ i $x(t)$ aby obliczyć $x(t+\Delta t)$. Wartość $x(t_1)$ może być oszacowana przez dowolną metodę samo-startującą, np. metodę Eulera lub można ją wyznaczyć z szeregu Taylora, tak jak w tym przypadku:

$$x(t_1) \approx x(t_0) + v(t_0)\Delta t + \frac{1}{2}a\Big(t_0, x(t_0)\Big)\Delta t^2, \tag{5.20}$$

gdzie $t_1 = t_0 + \Delta t$. Rozwiązując równania ruchu często należy wyznaczyć również prędkość poruszającego się ciała. Dla podstawowego algorytmu Verleta można to zrobić przy pomocy twierdzenia Lagrange'a o wartości średniej:[1]

$$v(t) = \frac{x(t+\Delta t) - x(t-\Delta t)}{2\Delta t}. \tag{5.21}$$

Warto zwrócić uwagę, że aby obliczyć prędkość w chwili t, potrzebna jest wartość położenia w chwili $t+\Delta t$, co może być problematyczne przy implementacji takiego algorytmu.

5.6.3 Prędkościowy algorytm Verleta

Znacznie częściej niż podstawowy algorytm Verleta stosowana jest jego modyfikacja – prędkościowy algorytm Verleta. Zależność określającą $x(t+\Delta t)$ można wyznaczyć z szeregu Taylora, otrzymując:

$$x(t+\Delta t) \approx x(t) + v(t)\Delta t + \frac{1}{2}a\Big(t, x(t)\Big)\Delta t^2. \tag{5.22}$$

Korzystając z (5.21) i (5.19) otrzymuje się:

$$v(t+\Delta t) \approx \frac{x(t+2\Delta t) - x(t)}{2\Delta t}$$
$$\approx \frac{2x(t+\Delta t) - x(t) + a\Big(t+\Delta t, x(t+\Delta t)\Big)\Delta t^2 - x(t)}{2\Delta t}$$

[1] $$\forall a \leqslant c \leqslant b : f(b) - f(a) = f'(c)(b-a).$$

$$\approx \frac{x(t+\Delta t)-x(t)}{\Delta t} + \frac{1}{2}a\Big(t+\Delta t, x(t+\Delta t)\Big)\Delta t. \tag{5.23}$$

Ostatecznie wstawiając (5.22) do (5.23) otrzymuje się:

$$v(t+\Delta t) \approx v(t) + \frac{1}{2}\Big(a\Big(t, x(t)\Big) + a\Big(t+\Delta t, x(t+\Delta t)\Big)\Big)\Delta t. \tag{5.24}$$

Prędkościowy algorytm Verleta jest samo-startujący oraz znacznie prostszy w zakodowaniu, kolejne wartości położenia i prędkości otrzymuje się w tym samym kroku iteracji.

5.6.4 Implementacja w języku C++

Funkcje rozwiązujące równanie różniczkowe drugiego rzędu za pomocą algorytmów Verleta posiadają interfejs przedstawiony na Listingu 5.5.

Listing 5.5: Definicja funkcji basic_verlet oraz velocity_verlet.

```
1 typedef struct
2 {
3   double x, v;
4 } XV;
5
6 void basic_verlet(double (*a)(double t, double x), XV *data, unsigned
      int data_size, double t0, double dt, XV xv0);
7 void velocity_verlet(double (*a)(double t, double x), XV *data,
      unsigned int data_size, double t0, double dt, XV xv0);
```

W przedstawionym pliku zdefiniowany został typ strukturalny `XV` zawierający dwa pola typu `double`: `x` oraz `v`. Typ ten reprezentuje odpowiednio położenie oraz prędkość ciała, którego równanie ruchu całkujemy. Funkcją całkującą równanie ruchu zgodnie z podstawowym algorytmem Verleta jest `basic_verlet`, prędkościowy algorytm Verleta realizowany jest natomiast przez funkcję `velocity_verlet`. Argumenty jakie przyjmuje funkcja to:

- `a` – wskaźnik na funkcję reprezentującą przyspieszenie ciała w chwili `t` i położeniu `x`; w ogólnym przypadku przyspieszenie ciała może zależeć także od prędkości, jednak algorytm Verleta zakłada, że tak nie jest;
- `data` – wskaźnik na wynikową tablicę danych stanowiących rozwiązanie równania różniczkowego zadanego przez funkcję `a`;
- `data_size` – rozmiar tablicy `data`;
- `t0` – wartość reprezentująca początkową chwilę w czasie, od której zostanie rozwiązane zadane równanie różniczkowe;
- `dt` – krok całkowania;
- `xv0` – wartości stanowiące wartości początkowe położenia oraz prędkości.

Podstawowy algorytm Verleta

Implementacja funkcji `basic_verlet` przedstawiona została na Listingu 5.6. Funkcja `basic_verlet` na początku sprawdza, czy ilość elementów w przekazanej tablicy mającej stanowić przybliżone rozwiązanie zadanego równania ruchu jest większa od dwóch. Podstawowy algorytm Verleta nie jest samo-startujący. Aby obliczyć wartość położenia w chwili $t + \Delta t$, należy znać położenie w chwilach t i $t - \Delta t$. Pierwszy element tablicy `data` zawsze otrzymuje wartości początkowe, wartość położenia dla drugiego elementu obliczana jest z zależności (5.20). Jako taki algorytm stosowany jest dopiero na dalszych elementach tablicy, dlatego wywołanie funkcji `basic_verlet` ma sens, jeśli rozmiar tablicy `data` jest większy bądź równy dwa. Jeśli rozmiar tablicy `data` jest mniejszy od dwóch, funkcja kończy działanie. W kolejnym kroku sterowanie przekazane jest do pętli, która kolejno na podstawie algorytmu Verleta (5.19), (5.21) oblicza położenia oraz prędkości odpowiednio w chwili $t_i + \Delta t : i \in [2; \text{data\_size}]$ oraz $t_i : i \in [1; \text{data\_size} - 1]$. Obliczone wartości położenia i prędkości zostają przypisane do elementów tablicy o indeksach i oraz $i - 1$. Dzieje się tak ponieważ aby obliczyć prędkość w chwili t_i, należy znać położenie w chwili $t_i + \Delta t$. Po zakończeniu pętli, tablica `data` zawiera `data_size` wartości przybliżających położenie oraz `data_size`-1 wartości przybliżających prędkość, dlatego też poza pętlą algorytm oblicza prędkość dla ostatniego elementu tablicy.

Listing 5.6: Definicja funkcji basic_verlet.

```
1  void basic_verlet(double (*a)(double t, double x), XV *data, unsigned
       int data_size, double t0, double dt, XV xv0)
2  {
3    if (data_size < 2)
4      return;
5    data[0].x = xv0.x;
6    data[0].v = xv0.v;
7    data[1].x = xv0.x + xv0.v * dt + 0.5 * a(t0, xv0.x) * dt * dt;
8    double t = t0 + dt;
9    for (int i = 2; i < data_size; i++, t += dt)
10     {
11       data[i].x = 2.0 * data[i - 1].x - data[i - 2].x + a(t - dt, data
             [i - 1].x) * dt * dt;
12       data[i - 1].v = (data[i].x - data[i - 2].x) / (2.0 * dt);
13     }
14   data[data_size - 1].v = ((2.0 * data[data_size - 1].x - data[
         data_size - 2].x + a(t, data[data_size - 1].x) * dt * dt) -
         data[data_size - 2].x) / (2.0 * dt);
15 }
```

Prędkościowy algorytm Verleta

Kod funkcji realizującej prędkościowy algorytm Verleta zamieszczony został na Listingu 5.7. W porównaniu do podstawowego algorytmu Verleta, algorytm prędkościowy jest mniej problematyczny w implementacji. Funkcja na początku sprawdza, czy rozmiar przekazanej tablicy jest większy od jedności. Pierwszy element taj tablicy zawsze otrzymuje wartości początkowe, więc sens prowadzenia obliczeń jest tylko wtedy, gdy rozmiar tej tablicy jest większy od jedności. Kolejny krok algorytmu to pętla obliczająca położenie i prędkość zgodnie ze wzorami (5.22) (5.24).

Listing 5.7: Definicja funkcji basic_verlet.

```
void velocity_verlet(double (*a)(double t, double x), XV *data,
      unsigned int data_size, double t0, double dt, XV xv0)
{
  if (data_size < 1)
    return;
  data[0].x = xv0.x;
  data[0].v = xv0.v;
  double t = t0 + dt;
  for (int i = 1; i < data_size; i++, t += dt)
    {
      data[i].x = data[i - 1].x + data[i - 1].v * dt + 0.5 * a(t - dt,
            data[i - 1].x) * dt * dt;
      data[i].v = data[i - 1].v + 0.5 * (a(t - dt, data[i - 1].x) + a(
           t, data[i].x)) * dt;
    }
}
```

5.7 Przykład - metoda Verleta

Chcąc przetestować podstawowy algorytm Verleta rozważmy równianie różniczkowe jednowymiarowego oscylatora harmonicznego:

$$\frac{d^2x}{dt^2} + \omega_0^2 x = 0, \tag{5.25}$$

gdzie: x – funkcja zależna od czasu reprezentująca położenie ciała w danej chwili czasu; d^2x/dt^2 – druga pochodna funkcji x po czasie; ω_0 – częstotliwość drgań oscylatora. Rozwiązaniem równania (5.25) jest funkcja typu:

$$x(t) = A\cos(\omega_0 t + \phi), \tag{5.26}$$

gdzie: A – amplituda drgań oscylatora, ω_0 – częstotliwość drgań oscylatora, ϕ – przesunięcie fazowe zależne od warunków początkowych. Prędkość oscylującego ciała na podstawie wzoru (5.25) dana jest zależnością $dv/dt = d^2x/dt^2 = -\omega_0^2 x$. Rozwiązaniem tego równania jest funkcja typu:

$$v(t) = -A\omega_0 \sin(\omega_0 t + \phi), \tag{5.27}$$

gdzie zmienne należy interpretować tak samo jak dla równania (5.26).

Przykładowy program rozwiązujący przytoczone równanie oscylatora harmonicznego za pomocą podstawowego i prędkościowego algorytmu Verleta zaprezentowany jest na Listingu 5.8.

Listing 5.8: Przykładowy program testujący podstawowy algorytm Verleta.

```
#include <iostream>
#include <cmath>
#include "ode.h"

const double omega = 3.0;
const double A = 2.0;
```

```
7  const int data_size = 100;
8
9  double d2f(double t, double x);
10 double df(double t);
11 double f(double t);
12
13 int main()
14 {
15   XV *bv_data = new XV[data_size];
16   XV *vv_data = new XV[data_size];
17   XV xv0 = {A, 0.0};
18   double t0 = 0.0;
19   double dt = 0.1;
20   basic_verlet(d2f, bv_data, data_size, t0, dt, xv0);
21   velocity_verlet(d2f, vv_data, data_size, t0, dt, xv0);
22   for (int i = 0; i < data_size; i++, t0 += dt)
23     std::cout << t0 << " "
24         << bv_data[i].x << " "<< bv_data[i].v << " "
25         << vv_data[i].x << " "<< vv_data[i].v << " "
26         << f(t0) << " " << df(t0) << std::endl;
27   delete [] bv_data;
28   delete [] vv_data;
29   return 0;
30 }
31
32 double d2f(double t, double x)
33 {
34   return -omega*omega * x;
35 }
36
37 double df(double t)
38 {
39   return -A * omega * std::sin(omega * t);
40 }
41
42 double f(double t)
43 {
44   return A * std::cos(omega * t);
45 }
```

Na początku program definiuje stałe oraz deklaruje funkcje:

- `omega = 3.0` – częstotliwość drgań oscylatora;
- `A = 2.0` – amplituda drgań oscylatora;
- `data_size = 100` – rozmiar tablicy danych stanowiącej przybliżone rozwiązanie równania (5.25);
- `d2f` – funkcja opisująca przyspieszenie oscylatora, przekształcając równanie (5.25) otrzymuje się zależność $a(t) = -\omega_0^2 x(t)$; w tym konkretnym przypadku przyspieszenie zależy tylko do położenia, jednak w ogólności może również zależeć od czasu, stąd parametrami tej funkcji są czas i położenie;
- `df` – funkcja opisująca prędkość oscylatora harmonicznego, równanie (5.27);
- `f` – funkcja stanowiąca jawne rozwiązanie równania oscylatora harmonicznego, czyli równanie (5.26).

Zarówno w funkcji `df`, jak i `d2f` założone jest, że przesunięcie fazowe równe jest zero.

W funkcji `main` tworzone są dynamicznie tablice mające stanowić przybliżone rozwiązanie równania (5.25), zdefiniowane zostają wartości początkowe położenia i prędkości, początkowy czas symulacji oraz krok całkowania. Przy początkowym (i nie tylko) wychyleniu oscylatora równym amplitudzie drgań, prędkość równa jest zero. Kolejny krok to wywołanie funkcji `basic_verlet` i `velocity_verlet` z parametrami opisanymi na początku sekcji. Wyniki obliczeń oraz wyniki jawnego rozwiązania funkcji zostają wyświetlone na ekranie. Finalnie zostaje zwolniona pamięć.

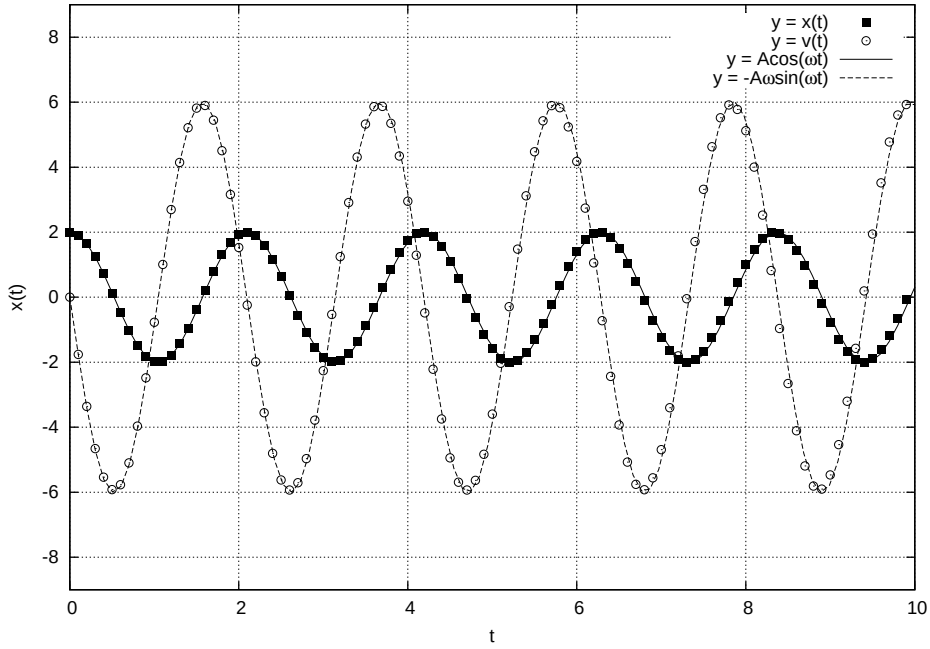
W Tabeli 5.1 przedstawione zostało porównanie rozwiązania numerycznego za pomocą podstawowego i prędkościowego algorytmu Verleta oraz rozwiązanie analityczne równania (5.25). Warunki początkowe oraz inne parametry są takie same, jak w programie na Listingu 5.8. Warto zwrócić uwagę, że rozwiązanie za pomocą

Tabela 5.1: Wynik działania programu z listingu 5.8. Symbole w nagłówku oznaczają odpowiednio: czas, położenie (podstawowy algorytm Verleta), prędkość (podstawowy algorytm Verleta), położenie (prędkościowy algorytm Verleta), prędkość (prędkościowy algorytm Verleta), położenie (rozwiązanie analityczne), prędkość (rozwiązanie analityczne).

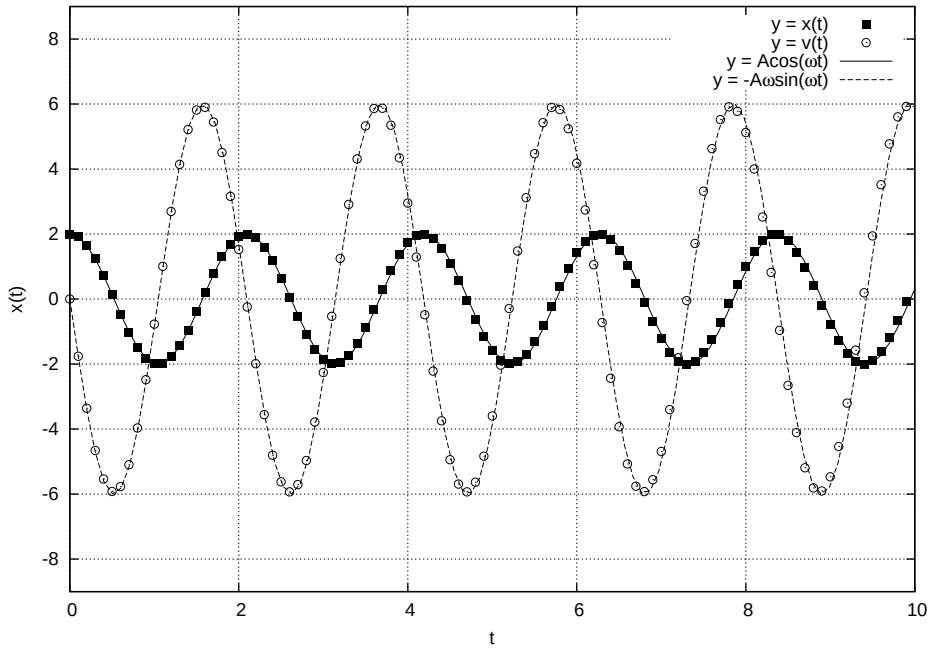
t	$x_{bv}(t)$	$v_{bv}(t)$	$x_{vv}(t)$	$v_{vv}(t)$	$x_a(t)$	$v_a(t)$
0	2	0	2	0	2	0
0,1	1,91	-1,7595	1,91	-1,7595	1,91067	-1,77312
0,2	1,6481	-3,36064	1,6481	-3,36064	1,65067	-3,38785
0,3	1,23787	-4,65933	1,23787	-4,65933	1,24322	-4,69996
0,4	0,716234	-5,53868	0,716234	-5,53868	0,724716	-5,59223
0,5	0,130135	-5,91954	0,130135	-5,91954	0,141474	-5,98497
0,6	-0,467675	-5,76765	-0,467675	-5,76765	-0,454404	-5,84309
0,7	-1,0234	-5,09667	-1,0234	-5,09667	-1,00969	-5,17926
0,8	-1,48701	-3,96699	-1,48701	-3,96699	-1,47479	-4,05278
0,9	-1,81679	-2,48028	-1,81679	-2,48028	-1,80814	-2,56428
...	...	...	...	...	...	...

podstawowego i prędkościowego algorytmu daje takie same wyniki, co nie powinno dziwić, gdyż obydwa algorytmy są sobie równoważne. Jednak jak zostało pokazane, algorytm prędkościowy jest prostszy w implementacji.

Wykresy przedstawiające rozwiązanie zamieszczone zostały na Rysunku 5.2 i 5.3. Linią ciągłą zaznaczono wychylenie oscylującego ciała z położenia równowagi, linią przerywaną prędkość oscylującego ciała, wartości zostały otrzymane na podstawie rozwiązania analitycznego. Kwadratami zaznaczono wartości wychylenia ciała z położenia równowagi, kołami natomiast prędkości tego ciała, otrzymane na podstawie algorytmów Verleta. Warto zwrócić uwagę, że zgodnie z oczekiwaniami, prędkość ciała równa jest zero dla maksymalnego wychylenia i przyjmuje wartość maksymalną dla zerowego wychylenia.



Rysunek 5.2: Podstawowy algorytm Verleta.



Rysunek 5.3: Prędkościowy algorytm Verleta.

Rozdział 6

Równania nieliniowe

6.1 Wstęp

W niniejszym rozdziale będą omówione proste metody rozwiązywania równań następującej postaci:

$$f(x) = 0. \tag{6.1}$$

Tego typu równania, których przykładem mogą być równania kwadratowe, trygonometryczne czy wykładnicze, są omawiane w szkołach. Te równania rozwiązywane są metodami dokładnymi (analitycznie). Zazwyczaj poszukiwanie pierwiastków równania polega na znajdowaniu wartości x, która spełnia jakąś relację, na przykład:

$$ax^3 + bx^2 = cx + d. \tag{6.2}$$

W zależności od problemu, wartość x może być liczbą rzeczywistą lub zespoloną. Procedura znajdowania pierwiastków równania polega na uporządkowaniu równania:

$$ax^3 + bx^2 - cx - d = 0. \tag{6.3}$$

Dla dowolnej wartości x, różnej od pierwiastka, to równanie nie może być spełnione, tak więc w ogólnym przypadku mamy:

$$ax^3 + bx^2 - cx - d = f(x). \tag{6.4}$$

Znajdowanie pierwiastków równań jest równoważne znajdowaniu wartości x dla których $f(x) = 0$. Niestety, ilość równań, które mogą być rozwiązane dokładnie jest niewielka. Równania typu (6.1) muszą być w większości przypadków rozwiązywane numerycznie. Wykonując odpowiednie wykresy, możemy mieć informacje o pierwiastkach równania (6.1). Równanie wyjściowe (6.1) zawsze można sprowadzić do postaci równoważnej:

$$g(x) = f(x),$$

A następnie narysować wykresy obu funkcji:

$$y = f(x),$$

$$y = g(x).$$

Rzeczywistymi pierwiastkami równania (6.1) są odcięte punktów przecięcia wykreślonych krzywych.

Zastosowanie omówionej geometrycznej metody szukania pierwiastków równania (6.1) zilustrujemy przykładem. Mamy równanie postaci:

$$2\sin(x) - x = 0. \tag{6.5}$$

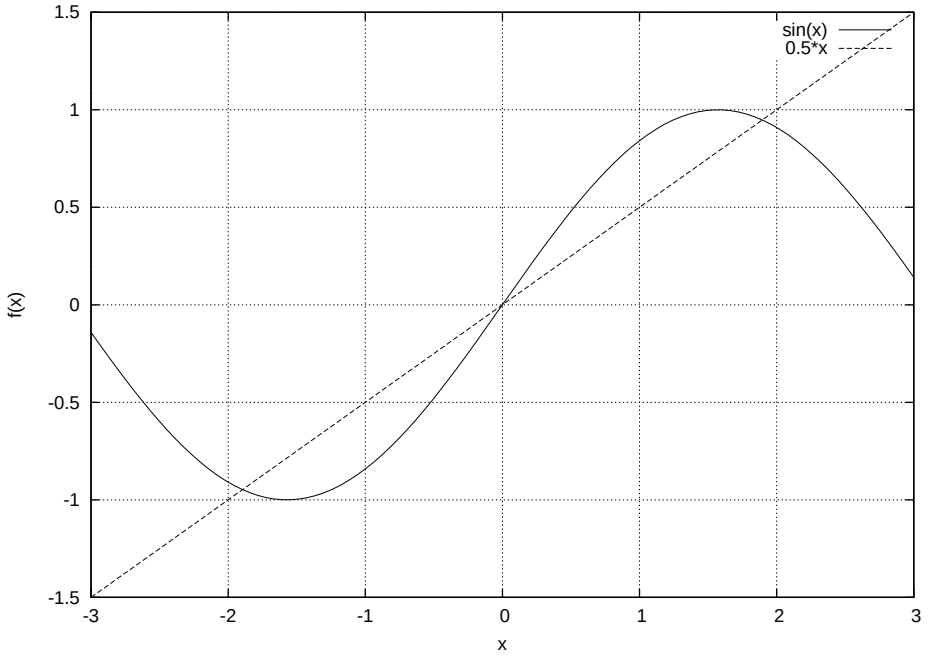
To równanie można przekształcić do postaci:

$$\sin(x) = \frac{1}{2}x. \tag{6.6}$$

A następnie wykonać wykresy dwóch funkcji:

$$y = \sin(x),$$
$$y = \frac{1}{2}x.$$

Wykresy tych funkcji pokazane są na Rysunku 6.1. Z wykresu widzimy, że oma-

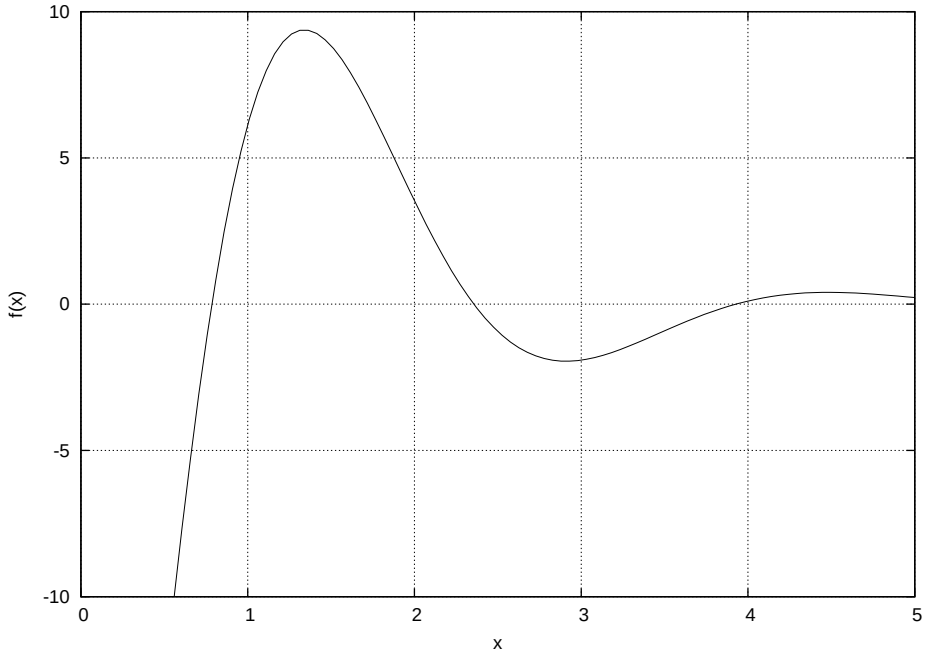


Rysunek 6.1: Interpretacja geometryczna równania $f(x) = 0$.

wiane równanie ma trzy pierwiastki. Jeden pierwiastek $x = 0$, dwa inne to około $-1,9$ i $1,9$. W numerycznych metodach poszukiwania pierwiastków równania (6.1) pomocne są dwa twierdzenia.

Twierdzenie 6.1 *Jeżeli funkcja $f(x)$ jest ciągła w przedziale $[a, b]$ i wartość funkcji na końcach przedziału przyjmuje różne znaki, to w przedziale tym znajduje się co najmniej jeden pierwiastek równania $f(x) = 0$. Na Rysunku 6.2 pokazano ilustracje tego twierdzenia.*

Twierdzenie 6.2 *Jeżeli w przedziale $[a, b]$ są spełnione warunki Twierdzenia 6.1 i znak pochodnej funkcji w tym przedziale jest stały, to w przedziale tym funkcja ma dokładnie jeden pierwiastek. Na Rysunku 6.3 pokazano ilustracje Twierdzenia 6.2.*

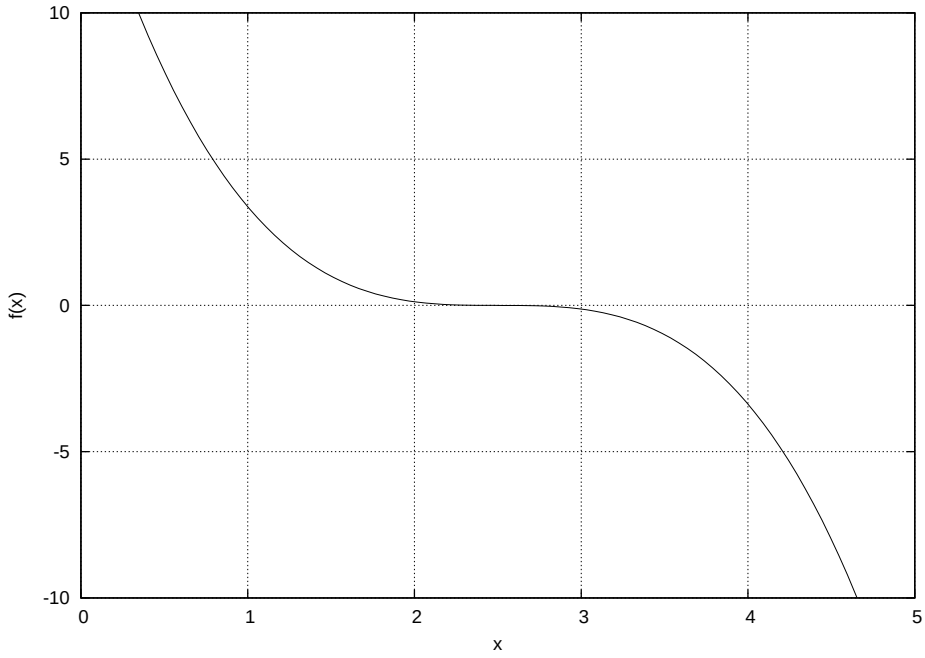


Rysunek 6.2: Geometryczna interpretacja Twierdzenia 6.1.

6.2 Metoda bisekcji

Metoda bisekcji (inna nazwa – połowienia) zaliczana jest to tzw. metod siłowych (ang. *brute force technique*), nie jest zbyt finezyjna, ale idealnie nadaje się do obliczeń komputerowych. Rozpatrzmy najprostszy przypadek. Niech funkcja $f(x)$ ma jeden rzeczywisty pierwiastek w przedziale $a < x < b$. Taka funkcja pokazana jest na Rysunku 6.3. Pierwiastek może być znaleziony z wystarczającą dokładnością wykorzystując następującą strategię. Przepoławiamy przedział $[a, b]$, znajdując środek przedziału (ang. *midpoint*):

$$x_m = \frac{a + b}{2}.$$



Rysunek 6.3: Geometryczna interpretacja Twierdzenia 6.2.

Wyliczamy iloczyn:

$$f(x_m) \cdot f(b)$$

Mamy trzy możliwości, iloczyn może być dodatni, ujemny lub równy 0. W ostatnim przypadku x_m jest pierwiastkiem równania $f(x) = 0$. Gdy iloczyn jest ujemny, pierwiastek leży w przedziale $x_m < x < b$. Jeżeli iloczyn jest dodatni, to $f(x)$ nie przecięła osi pomiędzy x_m i b, wobec tego pierwiastek leży w przedziale $a < x < x_m$. Należy wybrać przedział, w którym leży pierwiastek, przepołowić go i powtarzać opisana procedurę. Proces jest powtarzany tak długo, aż pierwiastek zostanie zlokalizowany z wystarczającą dokładnością. Jeżeli przerwiemy procedurę i uznamy, że pierwiastek znaleziono dla ostatnio wyznaczonego punktu środkowego podprzedziału, to maksymalny błąd nie będzie większy, niż połowa tego podprzedziału.

Na Listingu 6.1 zamieszczono kod funkcji obliczającej pierwiastek równania za pomocą algorytmu bisekcji.

Listing 6.1: Algorytm bisekcji.

```
1 bool bisection(double (*f)(double), double a, double b, double e,
      double & x0)
2 {
3   while (std::fabs(b - a) > e)
4     {
5       double m = (a + b) / 2;
6       double f_a = f(a);
7       double f_b = f(b);
8       double f_m = f(m);
9       if (f_a == 0.0)
```

```
10          {
11            x0 = a;
12            return true;
13          }
14        if (f_b == 0.0)
15          {
16            x0 = b;
17            return true;
18          }
19        if (f_m == 0.0)
20          {
21            x0 = m;
22            return true;
23          }
24        if (f_a * f_m < 0.0)
25          b = m;
26        else if (f_b * f_m < 0.0)
27          a = m;
28        else return false;
29      }
30    x0 = (a + b) / 2;
31    return true;
32  }
```

Argumenty funkcji `bisection` to:

- `f` – wskaźnik na jawnie zakodowaną funkcję definiującą równanie $f(x) = 0$;
- `a` – początek przedziału w którym szukane będzie rozwiązanie;
- `b` – koniec przedziału w którym szukane będzie rozwiązanie;
- `e` – zmienna określająca pożądaną dokładność wyniku;
- `x0` – oszacowana wartość pierwiastka funkcji `f`.

Jeśli istnieje pierwiastek równania w zadanym przedziale funkcja `bisection` zwraca wartość logiczną, która równa jest prawdzie (**`true`**), jeśli nie, zwracany jest fałsz logiczny (**`false`**). Wartość `x0` jest nieokreślona jeśli funkcja zwróci logiczny fałsz. Algorytm w każdej iteracji sprawdza czy punkty będące granicą przedziału oraz punkt środkowy nie są pierwiastkiem przekazanej funkcji, jeśli tak to natychmiastowo zwracany jest wynik.

6.3 Metoda siecznych

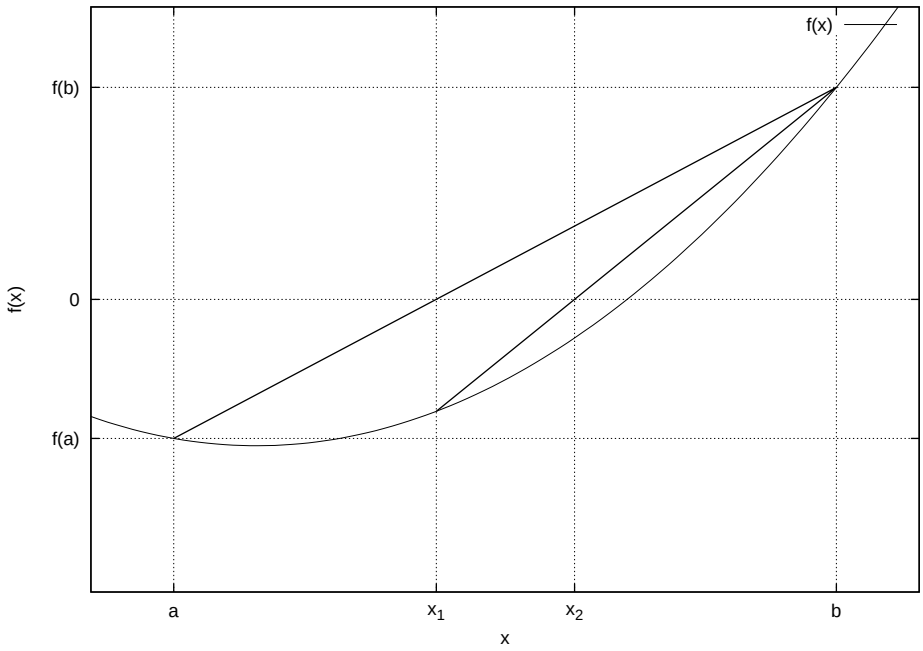
Rozwinięte przez I. Newtona metody szukania pierwiastków równania $f(x) = 0$ wymagały obliczania pochodnych funkcji. Bardzo często nie jest łatwo otrzymać pochodne funkcji, ale można zastąpić pochodną jej aproksymacją:

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}. \tag{6.7}$$

Metoda siecznych jest następująca. Po wyborze przybliżeń początkowych x_0 oraz x_1 tworzony jest rekurencyjny ciąg $x_2, x_3, x_4, \ldots$ na podstawie wzoru:

$$\begin{cases} x_0 = a, \\ x_1 = b, \\ x_i = x_{i-1} - f(x_{i-1}) \dfrac{x_{i-1} - x_{x-2}}{f(x_{i-1}) - f(x_{i-2})}, i = 2, 3, \ldots, n. \end{cases} \tag{6.8}$$

Interpretacja geometryczna metody jest następująca. Wartość x_i wyznaczana jest jako odcięta punktu przecięcia siecznej (cięciwy) przechodzącej przez punkty $(x_{i-2}, f(x_{i-2}))$ i $(x_{i-1}, f(x_{i-1}))$ z osią x-ów (pokazana jest to na Rysunku 6.4). Metoda



Rysunek 6.4: Metoda siecznych.

siecznych wymaga dwóch przybliżeń początkowych, w każdym kroku oblicza się tylko jedną nową wartość funkcji. Jak widać z Rysunku 6.4 kolejne przybliżenia w metodzie siecznych mają postać:

$$\begin{aligned} x_2 &= b - f(b) \frac{b - a}{f(b) - f(a)} \\ x_3 &= x_2 - f(x_2) \frac{x_2 - b}{f(x_2) - f(b)} \\ &\vdots \\ x_i &= x_{i-1} - f(x_{i-1}) \frac{x_{i-1} - x_{x-2}}{f(x_{i-1}) - f(x_{i-2})}, \end{aligned}$$

Metodę siecznych można zilustrować przykładem (A. Bjork, G. Dahlquist). Mamy równanie postaci:

$$f(x) = \sin(x) - \left(\frac{1}{2}x\right)^2. \tag{6.9}$$

Wybieramy $x_0 = 1$, $x_1 = 2$. Naszym zadaniem jest znalezienia pierwiastka α tego równania. Żądamy dokładności pięciu cyfr po przecinku. Kolejne wyniki iteracji pokazane są w Tabeli 6.1.

Tabela 6.1: Wyniki kolejnych obliczeń wyznaczania pierwiastka równania (6.9).

n	x_i	$f(x_i)$	$x_i - \alpha$
0	1	0,59147	-0,93
1	2	-0,090703	0,066
2	1,86704	0,084980	-0,067
3	1,93135	0,003177	-0,0024
4	1,93384	-0,000114	0,00009
5	1,93375	0,000005	$< 0,5 \cdot 10^{-5}$

Listing 6.2 zawiera kod funkcji szacującej pierwiastek równania metodą siecznych.

Listing 6.2: Metoda siecznych.

```
1  double secant_method(double (*f)(double), double a, double b, unsigned
       int n)
2  {
3    double x0 = 0.0;
4    for (unsigned int i = 0; i < n; i++)
5      {
6        double f_a = f(a);
7        double f_b = f(b);
8        if (f_a == f_b)
9          break;
10       x0 = b - f_b * (b - a) / (f_b - f_a);
11       a = b;
12       b = x0;
13     }
14   return x0;
15 }
```

Przedstawiona funkcja przyjmuje następujące parametry:

- `f` – wskaźnik na jawnie zakodowaną funkcję definiującą równanie $f(x) = 0$;
- `a` – początek przedziału w którym szukane będzie rozwiązanie;
- `b` – koniec przedziału w którym szukane będzie rozwiązanie;
- `n` – liczba iteracji algorytmu.

Wartością zwracaną jest obliczony pierwiastek równania $f(x) = 0$. Ze względu na skończoną dokładność z jaką reprezentowane są liczby w komputerach, algorytm zawiera warunek, który kończy iterację jeśli wartości funkcji dla x_{i-1} i x_{i-2} są sobie równe. Przeciwdziała to dzieleniu przez zero.

6.4 Metoda Newtona

Metoda Newtona (zwana także metodą stycznych, lub też metodą Newtona-Raphsona) jest elegancką metodą poszukiwania pierwiastka równania $f(x) = 0$, wymaga jednak znajomości pierwszej pochodnej funkcji $f(x)$. Jeżeli założymy, że punkt x_0 leży dostatecznie blisko pierwiastka równania $f(x)$, to możemy rozwinąć $f(x)$ w szereg Taylora w punkcie x_0:

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0) + \dots . \tag{6.10}$$

Jeżeli $f(x)$ jest przyrównane do zera, to x musi być pierwiastkiem równania. Jeżeli uwzględnimy ten fakt i z nieskończonego rozwinięcia szeregu Taylora weźmiemy tylko dwa pierwsze czynniki to mamy równość:

$$0 = f(x_0) + (x - x_0)f'(x_0). \tag{6.11}$$

Rozwiązanie ze względu na x ma postać:

$$x = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{6.12}$$

lub

$$x - x_0 = \delta = -\frac{f(x_0)}{f'(x_0)}. \tag{6.13}$$

Teraz x reprezentuje poprawione oszacowanie pierwiastka równania. Ta wartość może zastąpić x_0 w równaniu (6.11) dając lepsze oszacowanie w kolejnym kroku iteracji. Ogólna formuła we wzorze Newtona może być zapisana następująco:

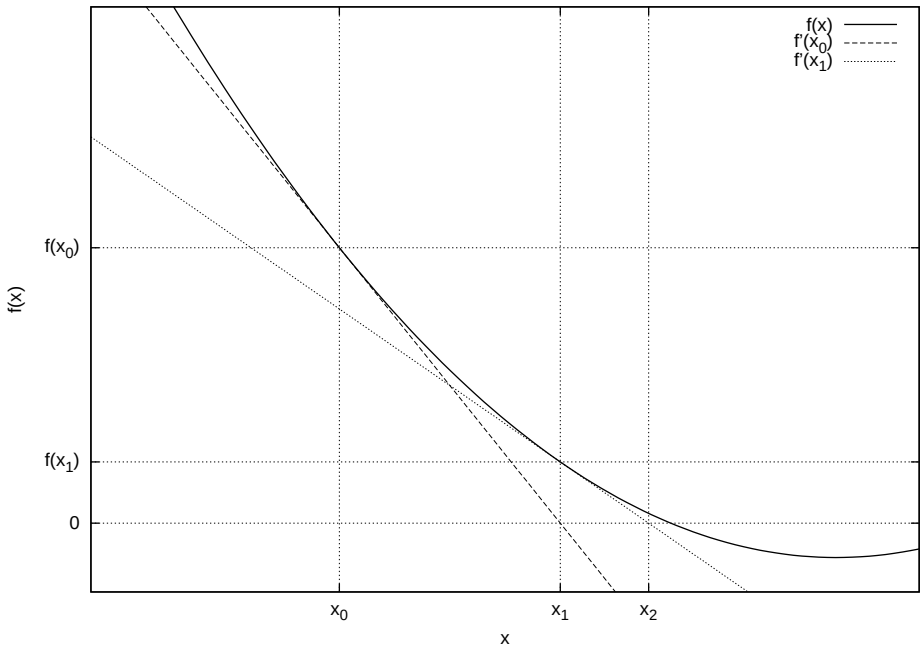
$$x_i - x_{i-1} = \delta_i = -\frac{f(x_{i-1})}{f'(x_{i-1})}, i = 1, 2, \dots, n. \tag{6.14}$$

Procedura, gdy jest zbieżna jest bardzo szybka. Obliczenia są zatrzymane, gdy wyliczona zmiana w wartości pierwiastka δ jest mniejsze niż założona wartość, którą oznaczamy, jako ε (dokładność). W opisanej metodzie kluczowe znaczenie ma wybór punktu startowego x_0. Można pokazać, że niewłaściwy wybór punktu startowego określa kolejny punkt iteracji, który będzie prowadził do procesu rozbieżnego (mówimy, że mamy niestabilny proces obliczeniowy). W praktyce często jako punkt startowy obiera się 0 lub 1.

W interpretacji geometrycznej metody Newtona przybliżamy pierwiastek równania $f(x)$ tworząc ciąg liczb utworzonych przez miejsca przecięcia się stycznych funkcji z osią x (Rysunek 6.5). Tangens nachylenia krzywej ma postać:

$$\tan\theta = \frac{f(x_0)}{x_1 - x_0} = f'(x_0). \tag{6.15}$$

Na Listingu 6.3 zamieszczony został kod funkcji obliczającej pierwiastek funkcji za pomocą metody Newtona.



Rysunek 6.5: Graficzna interpretacja metody Newtona.

Listing 6.3: Metoda Newtona.

```
1  double newton_method(double (*f)(double), double e, double start,
       double h)
2  {
3    double x0 = start;
4    double f_x0 = f(x0);
5    while (std::fabs(f_x0) > e)
6      {
7        double df = (f(x0 + h) - f_x0) / h;
8        x0 -= f_x0 / df;
9        f_x0 = f(x0);
10     }
11   return x0;
12 }
```

Funkcja `newton_method` zdefiniowana jest dla następujących parametrów:

- `f` – wskaźnik na jawnie zakodowaną funkcję definiującą równanie $f(x) = 0$;
- `e` – zmienna określająca pożądaną dokładność wyniku;
- `start` – wartość określająca punkt startowy;
- `h` – wielkość kroku różniczkowania (wykorzystywana przy obliczeniu pochodnej).

Wartością zwracaną przez funkcję `newton_method` jest oszacowana wartość pierwiastka funkcji `f`.

6.5 Porównanie omówionych metod

Niech dane będzie nieliniowe równanie $f(x) = 0$, posiadające jedno rozwiązanie i określone jak następuje:

$$f(x) = \frac{1}{1 + \exp(-x)} - \frac{1}{2} = 0, \tag{6.16}$$

gdzie $\exp(\cdot)$ to funkcja eksponencjalna. Rozwiązaniem równania jest $x = 0$. Na Listingu 6.4 zamieszczono kod programu rozwiązującego równanie (6.16) za pomocą omówionych metod.

Listing 6.4: Program testujący omówione metody rozwiązywania równań nieliniowych.

```
#include <iostream>
#include <cmath>
#include "nonlinear_equations.h"

double f(double);

int main()
{
  for (int i = 1; i < 10; i++)
    {
      double x0;
      bisection(f, -7, 6, 1.0 / std::pow(10, i), x0);
      std::cout << x0 << " "
                << secant_method(f, -7, 6, i) << " "
                << newton_method(f, 1.0 / std::pow(10, i), 1, 0.001)
                << std::endl;
    }
  return 0;
}

double f(double x)
{
  return 1 / (1 + std::exp(-x)) - 0.5;
}
```

W przedstawionym programie rozwiązanie poszukiwane jest w przedziale $[-7; 6]$, wyniki programu zostały umieszczone na Tabeli 6.2.

Tabela 6.2: Oszacowane rozwiązania równania (6.16) dla omówionych metod numerycznych.

i	Metoda bisekcji	Metoda siecznych	Metoda Newtona
1	-0.0175781	-0.489815	-0.175473
2	-0.00170898	0.77184	0.000894174
3	0.000274658	0.00849765	0.000894174
4	2.67029e-05	-0.000422561	-3.93557e-10
5	-1.19209e-06	2.41632e-09	-3.93557e-10
6	-2.98023e-08	-5.19519e-17	-3.93557e-10
7	1.86265e-08	-5.19519e-17	-3.93557e-10
8	-2.56114e-09	-5.19519e-17	-3.93557e-10
9	8.73115e-11	-5.19519e-17	-3.93557e-10

Rozdział 7

Układ równań liniowych

7.1 Wstęp

Układ równań liniowych można zdefiniować jako zestaw n równań z m niewiadomymi tak, że rozwiązaniem jest zbiór m liczb spełniających wszystkie równania z owego zestawu. Oczywiście taki układ równań może nie mieć rozwiązania, nazywamy go wtedy sprzecznym lub mieć nieskończenie wiele rozwiązań. W formie symbolicznej układ równań liniowych można zapisać jak następuje:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1m}x_m & = b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2m}x_m & = b_2, \\ & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nm}x_m & = b_n, \end{cases} \tag{7.1}$$

gdzie: zmienne a_{ij} zwane są współczynnikami, x_j to niewiadome, b_i to wyrazy wolne, dla $i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, m$. Taki układ równań można zapisać w formie macierzowo-wektorowej:

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1m} \\ a_{21} & a_{22} & \ldots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nm} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}. \tag{7.2}$$

Co można zapisać jako:

$$Ax = b, \tag{7.3}$$

gdzie: $A = [a_{ij}]$, $x = [x_j]$, $b = [b_i]$, dla $i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, m$. Jak zostało wspomniane taki układ może:

- posiadać jedno rozwiązanie, jest to układ *oznaczony*;
- posiadać nieskończenie wiele rozwiązań, jest to układ *nieoznaczony*;
- nie posiadać rozwiązań, jest to układ *sprzeczny*.

Jeśli założymy $n = m$ to rozwiązanie układu równań można wyznaczyć z zależności:

$$x = A^{-1}b, \tag{7.4}$$

gdzie A^{-1} to macierz odwrotna macierzy A. Odwracalność macierzy A oznacza, że układ równań posiada jedno rozwiązanie – jest oznaczony. Taki sposób rozwiązywania układu równań liniowych wymaga znajomości algorytmu odwracania macierzy.

Innym sposobem na rozwiązanie układu równań liniowych jest skorzystanie z wzorów Cramera:

$$x_1 = \frac{\det A_1}{\det A}, \tag{7.5}$$

$$x_2 = \frac{\det A_2}{\det A}, \tag{7.6}$$

$$\vdots$$

$$x_m = \frac{\det A_m}{\det A}, \tag{7.7}$$

gdzie macierz $A_i : i = 1, 2, \ldots, m$ to macierz powstała z macierzy A w której zastąpiono i-tą kolumnę wektorem b. Wzory Cramera oczywiście mają sens dla $\det A \neq 0$.

7.2 Metoda Gaussa

Metoda Gaussa zwana także *metodą eliminacji Gaussa* jest algorytmem służącym do rozwiązywania układu równań nieliniowych, typu $Ax = b$, gdzie A jest macierzą $n \times n$ natomiast x i b wektorami o rozmiarze n elementów każdy. Rozwiązywany układ równań należy sprowadzić do *trójkątnego układu równań* za pomocą operacji elementarnych, do których można zaliczyć:

- zamianę równań miejscami,
- pomnożenie równania przez stałą różną od zera,
- dodanie bądź odjęcie od jednego równania innego.

A więc układ równań:

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix},$$

powinno się sprowadzić do równoważnego układu typu:

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ 0 & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix},$$

za pomocą wymienionych operacji elementarnych. Taki układ równań można łatwo rozwiązać zaczynając od ostatniego równania aż do pierwszego. Macierz układu równań liniowych zapisuje się często w formie rozszerzonej o wektor wyrazów wolnych. Aby uzyskać macierz trójkątną należy zastosować przekształcenie $A^{k+1} = A^k$ takie, że:

$$a_{ij}^{k+1} = a_{ij}^k - \frac{a_{ik}^k a_{kj}^k}{a_{kk}^k}, \quad \begin{matrix} k & = 1, 2, \ldots, n, \\ i & = k+1, k+2, \ldots, n, \\ j & = k+1, k+2, \ldots, n+1, \end{matrix} \tag{7.8}$$

gdzie $j = n+1$ oznacza odwołanie do wektora wyrazów wolnych. W kolejnym kroku zakłada się że $x = b$ i przekształca wektor x zgodnie z zależnością:

$$x_i = \frac{x_i - a_{ij} x_j}{a_{ii}}, \quad \begin{matrix} i & = n, n-1, \ldots, 1, \\ j & = i+1, i+2, \ldots, n. \end{matrix} \tag{7.9}$$

We wzorach (7.8) i (7.9) zakłada się, że współczynniki leżące na głównej przekątnej są niezerowe. Po wykonaniu przekształceń (7.8) i (7.9) wektor x będzie zawierał rozwiązanie zadanego układu równań liniowych.

Listing 7.1 zawiera kod funkcji rozwiązującej liniowy układ równań metodą eliminacji Gaussa.

Listing 7.1: Metoda eliminacji Gaussa.

```
1  void gaussian_elimination(double **a, double *x, double *b, unsigned
       int n)
2  {
3    for (int k = 0; k < n - 1; k++)
4      {
5        for (int i = k + 1; i < n; i++)
6          {
7            a[i][k] /= a[k][k];
8            for (int j = k + 1; j < n + 1; j++)
9              {
10               if (j == n)
11                 b[i] -= a[i][k] * b[k];
12               else a[i][j] -= a[i][k] * a[k][j];
13             }
14           a[i][k] = 0;
15         }
16     }
17   for (int i = n - 1; i >= 0; i--)
18     {
19       x[i] = b[i];
20       for (int j = i + 1; j < n; j++)
21         {
22           if (j == n)
23             x[i] -= b[i] * x[j];
24           else x[i] -= a[i][j] * x[j];
25         }
26       x[i] /= a[i][i];
27     }
28 }
```

Funkcja `gaussian_elimination` wymaga przekazania parametrów opisanych na następującej liście:

- `a` – dwuwymiarowa dynamicznie alokowana tablica reprezentująca macierz współczynników układu równań (bez dodatkowej kolumny);
- `x` – wektor reprezentujący wyrazy niewiadome, w owym wektorze umieszczone zostanie rozwiązanie układu równań;
- `b` – wektor wyrazów wolnych;
- `n` – rozmiar macierzy `a` (powinna to być macierz kwadratowa) oraz wektorów `x` i `b`;

Warto zwrócić uwagę, że pamięć dla wektora `x` powinna być zadeklarowana przed wywołaniem funkcji oraz, że elementy macierzy `a[i][k]` nie muszą być zerowane, gdyż nie wpływa to na rozwiązanie układu równań. Po początkowych przekształceniach wartości te nie są już wykorzystywane, jednak macierz przekazana jest przez wskaźnik co oznacza, że jej wartości będą dostępne dla programisty po wywołaniu rozważanej funkcji. Zerowanie elementów (`a[i][k] = 0`) sprawia, że macierz po wykonaniu funkcji jest trójkątna i równoważna macierzy przekazanej.

7.3 Przykład - Metoda Gaussa

Jako przykład niech dany będzie układ trzech równań zdefiniowany jak następuje:

$$\begin{cases} x_1 + 2x_2 + 5x_3 & = 1, \\ 2x_1 + 5x_2 + 3x_3 & = 5, \\ -2x_1 - 4x_2 + 5x_5 & = 4. \end{cases} \tag{7.10}$$

Macierz tego układu wraz z dodatkową kolumną to:

$$\left[\begin{array}{ccc|c} 1 & 2 & 5 & 1 \\ 2 & 5 & 3 & 5 \\ -2 & -4 & 5 & 4 \end{array}\right]. \tag{7.11}$$

Aby sprowadzić tą macierz do postaci schodkowej należy do ostatniego wiersza dodać oraz od drugiego odjąć dwukrotność pierwszego, przez co otrzymuje się:

$$\left[\begin{array}{ccc|c} 1 & 2 & 5 & 1 \\ 0 & 1 & -7 & 3 \\ 0 & 0 & 15 & 6 \end{array}\right]. \tag{7.12}$$

Tak zapisaną macierz należy zapisać w formie układu równań:

$$\begin{cases} x_1 + 2x_2 + 5x_3 & = 1, \\ x_2 - 7x_3 & = 3, \\ 15x_3 & = 6, \end{cases} \tag{7.13}$$

który należy rozwiązać od ostatniego, a więc:

$$x_3 = 0,4, \tag{7.14}$$

$$x_2 = 3 + 7\frac{6}{15} = \frac{45}{15} + \frac{42}{15} = \frac{87}{15} = 5,8, \tag{7.15}$$

$$x_1 = 1 - 2 \cdot \frac{87}{15} - 5 \cdot \frac{6}{15} = 1 - \frac{174}{15} - \frac{30}{15} = -12,6. \tag{7.16}$$

Na Listingu 7.1 zamieszczono kod programu, który rozwiązuje liniowy układ równań (7.10) za pomocą eliminacji Gaussa.

Listing 7.2: Program wykorzystujący funkcję `gaussian_elimination` do rozwiązania układu równań liniowych.

```
#include <iostream>
#include "linear_equations.h"

const unsigned int N = 3;

int main()
{
  double **a = new double*[N];
  for (int i = 0; i < N; i++)
    a[i] = new double[N];
  double *b = new double[N];
  double *x = new double[N];
  a[0][0] = 1; a[0][1] = 2; a[0][2] = 5;
  a[1][0] = 2; a[1][1] = 5; a[1][2] = 3;
  a[2][0] = -2; a[2][1] = -4; a[2][2] = 5;
  b[0] = 1; b[1] = 5; b[2] = 4;
  gaussian_elimination(a, x, b, N);
  for (int i = 0; i < N; i++)
    std::cout << x[i] << " ";
  std::cout << std::endl;
  delete [] x;
  delete [] b;
  for (int i = 0; i < N; i++)
    delete [] a[i];
  delete [] a;
  return 0;
}
```

Wyjście programu, a więc rozwiązanie zadanego układu równań liniowych, jest zgodne z przeprowadzonym wcześniej rozwiązaniem analitycznym, czyli: `-12.6 5.8 0.4`.

7.4 Metoda Thomasa

Metoda eliminacji Gaussa jest algorytmem o złożoności, którą można uznać za dużą, $O(n^3)$. Jeśli układ równań, który należy rozwiązać ma specyficzną strukturę, można spróbować uprościć metodę Gaussa, tak aby jej złożoność była mniejsza. Przykładem tak uproszczonego algorytmu może być metoda Thomasa, która służy do rozwiązywania *układów równań trójprzekątniowych* i jej złożoność jest rzędu $O(n)$.

Wspomniany, trójprzekątniowy układ równań liniowych można zdefiniować jak następuje:

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & \dots & 0 \\ 0 & a_3 & b_3 & c_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & 0 & 0 & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix}. \tag{7.17}$$

Taki układ równań można zapisać jako:

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i, \tag{7.18}$$

gdzie: $a_1 = c_n = 0$ oraz $i = 1, 2, \dots, n$. Algorytm Thomasa sprowadza się do wyzerowania współczynników $a_i : i = 2, 3, \dots, n$, tak aby ostatnie równanie było postaci $b_n x_n = d_n$, co pozwoli wyznaczyć x_n. Mając dane x_n można wyznaczyć x_{n-1}, oczywiście przy założeniu $a_{n-1} = 0$, tak postępując można rozwiązać cały układ równań. Rozważmy równanie (7.18) dla $i = 1$:

$$b_1 x_1 + c_1 x_2 = d_1,$$

równanie to można podzielić przez b_1, a następnie pomnożyć przez a_2 i odjąć od równania drugiego:

$$(a_2 x_1 + b_2 x_2 + c_2 x_3 = d_2) - \left(a_2 x_1 + \frac{a_2}{b_1} c_1 x_2 = \frac{a_2}{b_1} d_1 \right).$$

Równanie drugie można więc zapisać jak następuje (współczynnik a_2 został wyeliminowany):

$$\left(b_2 - \frac{a_2}{b_1} c_1 \right) x_2 + c_2 x_3 = d_2 - \frac{a_2}{b_1} d_1. \tag{7.19}$$

Równanie (7.19) można podzielić przez $b_2 - (a_2/b_1)c_1$, pomnożyć przez a_3 i tak uzyskane równanie odjąć od równania trzeciego:

$$(a_3 x_2 + b_3 x_3 + c_3 x_4 = d_3) - \left(a_3 x_2 + \frac{a_3}{b_2 - \frac{a_2}{b_1} c_1} c_2 x_3 = \frac{d_2 - \frac{a_2}{b_1} d_1}{b_2 - \frac{a_2}{b_1} c_1} \right),$$

otrzymując trzecie równanie postaci:

$$\left(b_3 - \frac{a_3}{b_2 - \frac{a_2}{b_1} c_1} c_2 \right) x_3 + c_3 x_4 = d_3 - \frac{d_2 - \frac{a_2}{b_1} d_1}{b_2 - \frac{a_2}{b_1} c_1}. \tag{7.20}$$

Na podstawie indukcji matematycznej można zapisać rekurencyjną zależność wyznaczającą współczynniki b_i i d_i każdego z równań w rozważanym układzie, tak

aby współczynniki a_i, były równe zero:

$$b_i = b_i - \frac{a_i}{b_{i-1}} c_{i-1} : i = 2, 3, \ldots, n, \tag{7.21}$$

$$d_i = d_i - \frac{a_i}{b_{i-1}} d_{i-1} : i = 2, 3, \ldots, n. \tag{7.22}$$

Przy zerowej wartości współczynników a_i, można zapisać rekurencyjną zależność wyznaczającą niewiadome rozważanego układu:

$$\begin{cases} x_n = \dfrac{d_n}{b_n}, \\ x_i = \dfrac{d_i - c_i x_{i+1}}{b_i} : i = n-1, n-2, \ldots, 1. \end{cases} \tag{7.23}$$

Na Listingu 7.3 zamieszczono kod funkcji rozwiązującej trójprzekątniowy układ równań liniowych. Warto zwrócić uwagę, że złożoność tego algorytmu równa jest $O(n)$. Poza tym macierz takiego układu równań może być przechowywana w pamięci komputera za pomocą trzech tablic jednowymiarowych, co dodatkowo zmniejsza zużycie pamięci.

Listing 7.3: Metoda Thomasa.

```
1  void thomas_method(double *a, double *b, double *c, double *x, double
       *d, unsigned int n)
2  {
3    for (int i = 1; i < n; i++)
4      {
5        double l = a[i] / b[i - 1];
6        a[i] = 0.0;
7        b[i] -= l * c[i - 1];
8        d[i] -= l * d[i - 1];
9      }
10
11   x[n - 1] = d[n - 1] / b[n - 1];
12   for (int i = n - 2; i >= 0; i--)
13     x[i] = (d[i] - c[i] * x[i + 1]) / b[i];
14 }
```

Parametry funkcji z Listingu 7.3 określone są jak następuje:

- `a` – wskaźnik na tablicę zawierającą wartości współczynników a,
- `b` – wskaźnik na tablicę zawierającą wartości współczynników b,
- `c` – wskaźnik na tablicę zawierającą wartości współczynników c,
- `x` – wskaźnik na tablicę niewiadomych,
- `d` – wskaźnik na tablicę wyrazów wolnych
- `n` – rozmiar tablic `a`, `b`, `c`, `x` i `d`.

Warto zwrócić uwagę, że pamięć dla tablicy `x` powinna być zaalokowana przed wywołaniem funkcji `thomas_method`. Mimo iż tablice `a` i `c` mogą być rozmiaru `n - 1`,

algorytm nie zakłada takiego rozwiązania, zysk w postaci miejsca w pamięci dla dwóch zmiennych typu **double** byłby opłacony skomplikowaniem kodu. W algorytmie zmienne `a[0]` i `c[n - 1]` nie są używane i mogą mieć wartości dowolne.

Przykład wykorzystania algorytmu Thomasa można znaleźć w Rozdziale 2. gdzie algorytm użyty jest do rozwiązania układu równań przy interpolacji metodą funkcji sklejanych.

Rozdział 8

Metoda najmniejszych kwadratów

8.1 Wstęp

Rozważmy ponownie zagadnienie interpolacji funkcji. W Rozdziale 2. przedstawione zostały metody, których celem jest oszacowanie wartości funkcji w punktach leżących pomiędzy węzłami funkcji, tak dokładnie jak to możliwe. Co więcej tego typu interpolacja wymagała zachowania równości funkcji interpolacyjnej z funkcją interpolowaną w węzłach. Takie podejście nie zawsze jest słuszne, jeśli rozważymy funkcję zdefiniowaną przez pary punktów z dziedziny i przeciwdziedziny, których wartości uzyskane zostały na drodze eksperymentu. To takie dane zawierają zwykle tzw. szum związany z samym pomiarem i innymi czynnikami mającymi wpływ na przebieg eksperymentu.

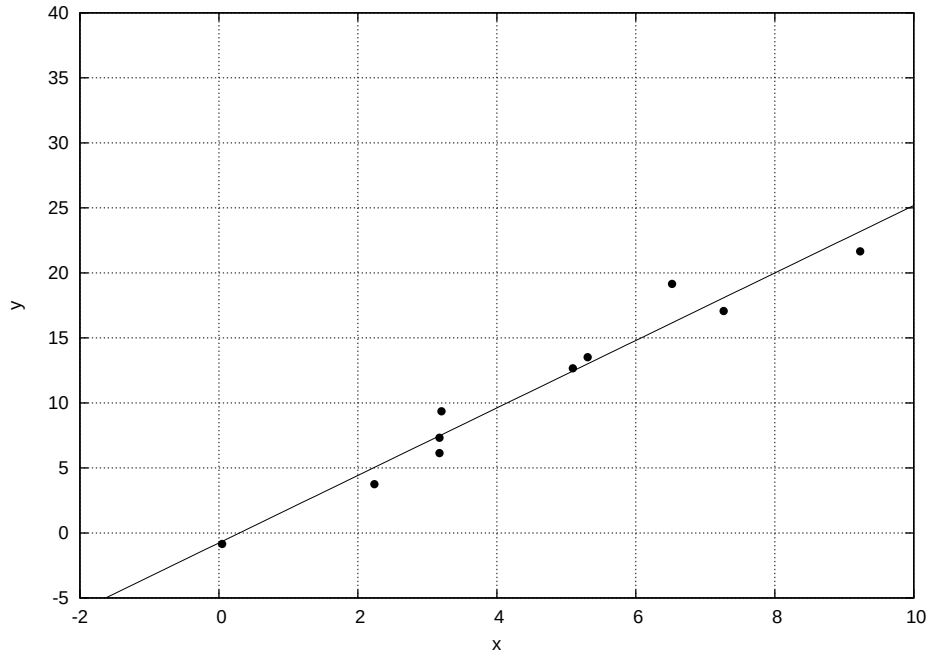
Na Rysunku 8.1 przedstawiono przykład danych zawierających szum, gdzie linia prosta to wartości dokładne. Korzystając z poznanych już metod interpolacji można by spróbować skonstruować wielomian interpolacyjny pasujący dokładnie do tych punktów (np. za pomocą interpolacji Lagrange'a). Jednak taki wielomian zawierałby szum będący częścią tych danych oraz byłby wielomianem wysokiego stopnia z dużą liczbą oscylacji. Najprawdopodobniej wartości tego wielomianu nie będące węzłami odbiegałyby znacząco od wartości dokładnej interpolowanych danych.

Znacznie lepszą aproksymacją takiej funkcji byłaby metoda pozwalająca skonstruować wielomian małego stopnia, który jednocześnie redukowałby szum. Jeśli założymy, że szum ma charakter losowy, to przybliżenie takich danych będzie linią prostą przechodzącą przez „środek" tych punktów.

Jeśli aproksymowaną funkcję oznaczymy przez $f(x)$, a funkcję aproksymującą przez $g(x)$, to lokalna odległość pomiędzy tymi funkcjami może być zdefiniowana jak następuje:

$$d(x) = |f(x) - g(x)|. \tag{8.1}$$

Jeśli zmienna x zadana jest przez dyskretne wartości $x_i : i = 1, 2, \ldots, n$, to można żądać minimalizacji funkcji $d(x)$ w sensie *najmniejszych kwadratów*, a więc mini-



Rysunek 8.1: Przykład danych zawierających szum.

malizacji funkcji:

$$E = \sum_{i=1}^{n} d^2(x_i). \tag{8.2}$$

Jeśli założymy że funkcja $g(x)$ dana jest przez wielomian m-tego stopnia:

$$g(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m, \tag{8.3}$$

to można zapisać:

$$\begin{aligned} E &= \sum_{i=1}^{n} |f(x_i) - g(x_i)|^2 = \sum_{i=1}^{n} |g(x_i) - f(x_i)|^2 = \sum_{i=1}^{n} (g(x_i) - f(x_i))^2 \\ &= \sum_{i=1}^{n} (a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_m x_i^m - f(x_i))^2. \end{aligned} \tag{8.4}$$

Minimalizacja funkcji E może być przeprowadzona przez przyrównanie do zera pochodnych cząstkowych funkcji E względem współczynników $a_i : i = 0, 1, \ldots, m$:

$$\frac{\partial E}{\partial a_0} = 0,$$
$$\frac{\partial E}{\partial a_1} = 0,$$

$$\vdots$$

$$\frac{\partial E}{\partial a_m} = 0. \tag{8.5}$$

Dowód prawdziwości tego stwierdzenia wykracza poza ramy niniejszej pozycji, więc nie zostanie przytoczony. Wzory (8.5) dają $m+1$ równań z $m+1$ niewiadomymi $a_i : i = 0, 1, \ldots, m$. Rozważając pierwsze równanie (8.5) można zapisać:

$$\begin{aligned}
\frac{\partial E}{\partial a_0} &= \frac{\partial}{\partial a_0} \sum_{i=1}^{n} (a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_m x_i^m - f(x_i))^2 \\
&= \sum_{i=1}^{n} \frac{\partial}{\partial a_0} (a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_m x_i^m - f(x_i))^2 \\
&= \sum_{i=1}^{n} 2(a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_m x_i^m - f(x_i)) \\
&\left(\frac{\partial}{\partial a_0} (a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_m x_i^m - f(x_i)) \right) \\
&= \sum_{i=1}^{n} 2(a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_m x_i^m - f(x_i))(1) \\
&= n a_0 + \left(\sum_{i=1}^{n} x_i \right) a_1 + \left(\sum_{i=1}^{n} x_i^2 \right) a_2 + \cdots + \left(\sum_{i=1}^{n} x_i^m \right) a_m = \sum_{i=1}^{n} f(x_i).
\end{aligned} \tag{8.6}$$

W podobny sposób można wyprowadzić równanie $\partial E / \partial a_1$, otrzymując:

$$\left(\sum_{i=1}^{n} x_i \right) a_0 + \left(\sum_{i=1}^{n} x_i^2 \right) a_1 + \cdots + \left(\sum_{i=1}^{n} x_i^{m+1} \right) = \sum_{i=1}^{n} x_i f(x_i). \tag{8.7}$$

Rozwiązując w podobny sposób pozostałe równania otrzymuje się układ równań:

$$\begin{bmatrix}
n & \psi(x_i) & \psi(x_i^2) & \ldots & \psi(x_i^m) \\
\psi(x_i) & \psi(x_i^2) & \psi(x_i^3) & \ldots & \psi(x_i^{m+1}) \\
\psi(x_i^2) & \psi(x_i^3) & \psi(x_i^4) & \ldots & \psi(x_i^{m+2}) \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\psi(x_i^m) & \psi(x_i^{m+1}) & \psi(x_i^{m+2}) & \ldots & \psi(x_i^{2m})
\end{bmatrix}
\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix}
=
\begin{bmatrix} \psi(f(x_i)) \\ \psi(x_i f(x_i)) \\ \psi(x_i^2 f(x_i)) \\ \vdots \\ \psi(x_i^m f(x_i)) \end{bmatrix}, \tag{8.8}$$

gdzie: $\psi(\cdot) = \sum_{i=1}^{n} (\cdot)$. Taki układ równań może być rozwiązany przy użyciu metody eliminacji Gaussa.

8.2 Dopasowanie funkcji liniowej

Jak już zostało wspomniane wielomiany relatywnie niskiego stopnia są najbardziej użyteczne przy aproksymacji funkcji. Okazuje się, że najczęściej używaną

funkcją do aproksymacji jest linia prosta. Nawet jeśli charakterystyka danych nie jest liniowa, zawsze można przeskalować wykres tak aby uzyskać liniową charakterystykę wyznaczanej funkcji (np. za pomocą skali logarytmicznej).

Niech $m = 1$ oraz $\psi(\cdot) = \sum_{i=1}^{n}(\cdot)$, równanie (8.8) przyjmuje postać:

$$\begin{bmatrix} n & \psi(x_i) \\ \psi(x_i) & \psi(x_i^2) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} \psi(f(x_i)) \\ \psi(x_i f(x_i)) \end{bmatrix}. \tag{8.9}$$

Mnożąc strony równania przez macierz odwrotną oraz korzystając z algorytmu obliczającego macierz odwrotną[1] otrzymuje się:

$$\begin{aligned} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} &= \begin{bmatrix} n & \psi(x_i) \\ \psi(x_i) & \psi(x_i^2) \end{bmatrix}^{-1} \begin{bmatrix} \psi(f(x_i)) \\ \psi(x_i f(x_i)) \end{bmatrix} \\ &= \frac{1}{n\psi(x_i^2) - (\psi x_i)^2} \begin{bmatrix} \psi(x_i^2) & -\psi(x_i) \\ -\psi(x_i) & n \end{bmatrix} \begin{bmatrix} \psi(f(x_i)) \\ \psi(x_i f(x_i)) \end{bmatrix} \\ &= \frac{1}{n\psi(x_i^2) - (\psi(x_i))^2} \begin{bmatrix} \psi(x_i^2)\psi(f(x_i)) - \psi(x_i)\psi(x_i f(x_i)) \\ n\psi(x_i f(x_i)) - \psi(x_i)\psi(f(x_i)) \end{bmatrix}. \end{aligned} \tag{8.10}$$

Co ostatecznie prowadzi do:

$$a_0 = \frac{(\sum_{i=1}^{n} x_i^2)(\sum_{i=1}^{n} f(x_i)) - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} x_i f(x_i))}{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2}, \tag{8.11}$$

$$a_1 = \frac{n\sum_{i=1}^{n} x_i f(x_i) - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} f(x_i))}{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2}. \tag{8.12}$$

Uzyskane w ten sposób zmienne a_0 i a_1 to współczynniki prostej $y = a_1 x + a_0$ aproksymującej funkcję $f(x)$ w sensie najmniejszych kwadratów.

Na Listingach 8.1 i 8.2 zamieszczone zostały definicje typów strukturalnych oraz funkcja, obliczająca za pomocą metody najmniejszych kwadratów współczynniki prostej $y = a_1 x + a_0$. Warto zwrócić uwagę, że struktura `F` i `XF` są niemal identyczne, co mogło by sugerować, że można je zastąpić jedną strukturą. Jednak w takiej sytuacji kod programu musiałby być opatrzony znaczną ilością komentarzy, tak by można rozróżnić czy struktura przechowuje dane dotyczące współczynników prostej lub punktów. Co więcej, takie rozwiązanie mogło by generować sytuacje w których można by porównywać dane określające punkt i prostą. Oczywiście nie ma to większego sensu, dlatego zasadne jest napisanie dwóch struktur danych, które są niemalże takie same lecz ich przeznaczenie jest inne. Dzięki temu kod jest bardziej czytelny i łatwiejszy w utrzymaniu.

Listing 8.1: Definicja struktury F i XF.

```
1 typedef struct
2 {
3   double a0, a1;
4 } F;
5
```

[1] $$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix},\ A^{-1} = \frac{1}{|A|} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

```
6 typedef struct
7 {
8   double x, f;
9 } XF;
```

Listing 8.2: Metoda najmniejszych kwadratów.

```
1 F least_squares(const XF *xf, unsigned int n)
2 {
3   double sum_x = 0.0;
4   double sum_f = 0.0;
5   double sum_xf = 0.0;
6   double sum_x2 = 0.0;
7   for (int i = 0; i < n; i++)
8     {
9       sum_x += xf[i].x;
10      sum_f += xf[i].f;
11      sum_xf += xf[i].x * xf[i].f;
12      sum_x2 += xf[i].x * xf[i].x;
13    }
14  F f;
15  f.a0 = (sum_x2 * sum_f - sum_x * sum_xf) / (n * sum_x2 - sum_x *
        sum_x);
16  f.a1 = (n * sum_xf - sum_x * sum_f) / (n * sum_x2 - sum_x * sum_x);
17  return f;
18 }
```

Funkcja `least_squares` przyjmuje parametry określone jako:

- `xf` – tablica struktur `XF` (Listing 8.1);
- `n` – rozmiar tablicy `xf`.

W rozważanej funkcji, w pętli, obliczane są sumy i na ich podstawie, poza pętlą, obliczane są współczynniki prostej regresji (8.11) i (8.12). Wartości te zwracane są w formie typu strukturalnego `F`, którego definicja znajduje się na Listingu 8.1.

8.3 Przykład - dopasowane funkcji liniowej

Jako przykład niech dane będą punkty $(x_i, y_i), i = 0, 1, \ldots, n$. Zdefiniowane jak następuje:

$$\begin{cases} x_i = \text{rand}(-4, 4) + i \\ y_i = \text{rand}(-4, 4) + 2,5i \end{cases}, i = 0, 1, \ldots, n, \tag{8.13}$$

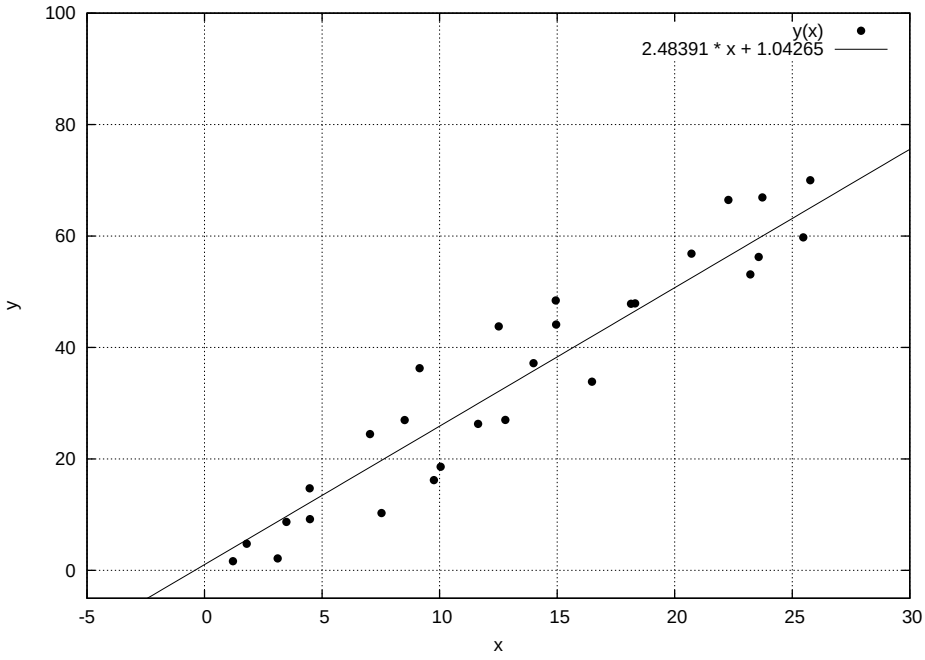
gdzie $\text{rand}(a, b)$ zwraca liczbę losową o rozkładzie równomiernym z przedziału $[a; b]$. Na Listingu 8.3 zamieszczono program, który generuje punkty w oparciu o (8.13), dla $n = 30$, oraz wykorzystuje funkcję `least_squares` do wyznaczenia współczynników prostej najlepiej dopasowującej się do tych danych w sensie najmniejszych kwadratów.

Listing 8.3: Program testujący funkcję z Listingu 8.3.

```
1 #include <iostream>
2 #include <cstdlib>
```

```
3  #include <ctime>
4  #include "least_squares.h"
5
6  const unsigned int N = 30;
7  const double E = 8;
8
9  int main()
10 {
11   std::srand(std::time(0));
12   XF *xf = new XF[N];
13   for (int i = 0; i < N; i++)
14     {
15       xf[i].x = std::rand() / (double)RAND_MAX * E - 0.5 * E + i;
16       xf[i].f = std::rand() / (double)RAND_MAX * E - 0.5 * E + 2.5 * i
               ;
17       std::cout << xf[i].x << " " << xf[i].f << std::endl;
18     }
19   F f = least_squares(xf, N);
20   std::cout << std::endl << f.a0 << " " << f.a1 << std::endl;
21   delete [] xf;
22   return 0;
23 }
```

Na Rysunku 8.2 zamieszczono jedne z wyników programu z Listingu 8.3. Otrzymane współczynniki to: $a_0 = 1,04265$, $a_1 = 2,48391$. Oczywiście każde wykonanie programu da inny wynik ze względu na losowy algorytm generowania punktów (x_i, y_i).



Rysunek 8.2: Regresja liniowa za pomocą metody najmniejszych kwadratów.

Rozdział 9

Metody Monte Carlo

9.1 Wstęp

Rozwój technologii informatycznych powoduje ciągłe powiększanie się obszaru zastosowań obliczeń komputerowych – rozwijane są techniki znane od dziesięcioleci (na przykład metody statystyczne) lub tworzone nowe techniki (na przykład zastosowania logiki rozmytej, algorytmów genetycznych, algorytmów mrówkowych, czy metod falkowych). Znacząco rozwinęła się technologia zwana symulacją komputerową. Symulacje komputerowe stosowane są coraz częściej, jako metody pozwalające budować lub weryfikować teorie na podstawie danych doświadczalnych.

Istnieje duża klasa problemów, które są dobrze opisane teoretycznie, ale ich rozwiązanie nie jest możliwe w rzeczywistym czasie. Przykładem może być problem komiwojażera czy zagadnienie energii w ciele stałym (na przykład obliczenie energii powierzchniowej wymaga uwzględnienia funkcji falowych przynajmniej 10^{15} atomów tworzących powierzchnię). Aby otrzymać jakiekolwiek rozwiązanie, do modeli teoretycznych musimy wprowadzać daleko idące uproszczenia.

Metody symulacji komputerowych pozwalają na badanie modeli dużo mniej uproszczonych.

Technika symulacji komputerowych rozwinęła się obecnie tak, że w wielu przypadkach rezygnuje się z wykonywania kosztownych doświadczeń na rzecz tzw. eksperymentów komputerowych. Przykładem mogą być symulacje zachowań się samolotu podczas lotu. Zamiast budować kosztowne modele i wykonywać badania w tunelach aerodynamicznych, istotne dane techniczne otrzymuje się wyłącznie na podstawie symulacji komputerowych.

Symulacja komputerowa jest algorytmiczną metodą wykonywania eksperymentów przy pomocy komputerów na modelach dynamicznych rozważanych systemów.

System definiujemy, jako zbiór powiązanych ze sobą obiektów, każdy obiekt scharakteryzowany jest wybranymi wielkościami. Każda interesująca nas wielkość charakteryzowana jest skończonym zbiorem zmiennych. Wartości zmiennych w symulacjach komputerowych tworzą skończony i dyskretny zbiór. W mechanice klasycznej operujemy funkcjami ciągłymi i nieskończonymi, w symulacjach komputerowych zawsze mamy do czynienia z funkcjami dyskretnym i skończonymi. Konkrety stan rozważanego systemu określamy, jako zbiór wartości wybranych zmiennych.

Zmianę wybranego stanu systemu symulujemy zmieniając zbiór wartości wybranych zmiennych. Zmianę stanu systemu nazywamy zdarzeniem.

W celu wykonania symulacji komputerowych postępujemy w następujący sposób:

- Określamy problem do rozwiązania.
- Ustalamy system.
- Budujemy model.
- Przygotowujemy odpowiednie dane.
- Tworzymy algorytm i kodujemy program komputerowy.
- Wykonujemy serię eksperymentów komputerowych.
- Interpretujemy otrzymane wyniki.

Istnieje wiele metod symulacji komputerowych. Ten rozdział naszego skryptu w całości poświęcony będzie popularnej metodzie symulacji za jaką uważa się technikę (metodę) Monte Carlo. *Metodę Monte Carlo określamy, jako metodę polegającą na generowaniu zmiennych losowych w celu oszacowania parametrów ich rozkładu.* Z historycznego punktu widzenia, warto powołać się na nowatorską pracę A. Halla („On an experiment determination of π", Messeng. Math. (1873), nr 2, str. 113-114). Autor tej pracy uważany jest za twórcę idei wykorzystania zjawisk losowych do wykonania obliczeń. W cytowanej pracy Hall zaproponował sposób szacowania wartości liczby π. Istotą metody był zaprojektowany odpowiednio eksperyment: z określonej wysokości upuszczano igłę na płaszczyznę kartki papieru odpowiednio poliniowanej. Upadająca igła mogła przeciąć linie lub nie. Eksperymentator musiał policzyć ile razy igła przetnie linie. Prawdopodobieństwo wystąpienia zdarzenia wyraża się za pomocą liczby π, należy, zatem oszacować to prawdopodobieństwo. W czasie publikacji artykułu, eksperyment można było wykonać tylko ręcznie. Obecnie, opisany eksperyment możemy symulować, korzystając z pomocy komputera. W literaturze przedmiotu opisany eksperyment Halla nosi nazwę zadania Buffona. W 1944 roku John von Neumann zaproponował wykorzystanie rachunku prawdopodobieństwa i metod statystyki matematycznej do rozwiązywania zadań matematycznych wykorzystując komputery.

Formalnie za twórców metody Monte Carlo uważa się Metropolisa i Ulama, którzy opublikowali pionierską pracę na temat tej metody (N. Metropolis, S. Ulam, „The Monte Carlo Method", J. Amer. Stat. Assoc. (1949), nr 247, str. 335-341). Stanisław Ulam był polskim matematykiem, tuż przed drugą wojną światową wyemigrował do Ameryki. Praktyczne stosowanie metody Monte Carlo polega na wykonaniu następujących zadań:

- Należy wygenerować zmienna losową o danym rozkładzie prawdopodobieństwa.
- Należy skonstruować model probabilistyczny dla realnych systemów.
- Należy przeprowadzić estymację statystyczną.

9.2 Generatory Monte Carlo

Liczby losowe są bardzo przydatne w symulacjach komputerowych, testowaniu programów, w tworzeniu gier. Języki programowania, takie jak na przykład język C czy język C++ dostarczają odpowiednich narzędzi do tworzenia liczb, które zachowują się jak liczby losowe. Prawdziwe liczby losowe otrzymamy rzucając kostką sześcienną, w komputerze, korzystając z odpowiednich algorytmów możemy tworzyć jedynie tzw. *liczby pseudolosowe*. Funkcje, dzięki którym otrzymujemy liczby pseudolosowe noszą nazwę *generatorów liczb pseudolosowych*.

Istnieje wiele algorytmów realizacji programowej generatorów liczb pseudolosowych. Najczęściej badane i używane to:

- Generatory liniowe kongruentne (ang. *linear congruential generators*).
- Generatory multiplikatywne liniowe kongruentne (ang. *multiplicative linear congruential generators*).
- Generatory Fibonacciego (ang. *Fibonacci generators*).
- Generatory oparte na rejestrach przesuwnych (ang. *shift register generators*).
- Generatory kombinowane (ang. *combination generators*).

Generatory Monte Carlo produkują ciągi liczb losowych. W większości przypadków kolejna liczba pseudolosowa x_{i+1} jest funkcją wcześniej wygenerowanej liczby pseudolosowej:

$$x_{i+1} = f(x_i, x_{i-1}, \ldots, x_1). \tag{9.1}$$

Bardzo popularny w zastosowaniach jest generator liniowy kongruentny. Jest to generator postaci:

$$x_{i+1} = (a_i x_i + a_2 x_{i-1} + \cdots + a_k x_{i-k+1} + c) \mod m. \tag{9.2}$$

Gdzie współczynniki a, c i m są ustalonymi liczbami całkowitymi. Występujący we wzorze symbol $a \mod b$ czytamy: *a modulo b*, oznacza on resztę z dzielenia a przez b. Jeżeli $k = 1$, to mamy następujący wzór:

$$x_{i+1} = (ax_i + c) \mod m \tag{9.3}$$

Wszystkie wartości w powyższym wzorze przybierają wartości całkowite. Tego typu generator został zaproponowany w 1951 roku przez Lehmera, w skrócie oznaczany jest, jako generator LCG. Znając wartości początkowe ciągu liczb pseudolosowych x_0 oraz a i c mamy wyznaczony ciąg nieskończony. W praktyce generowany komputerowo ciąg liczb jest okresowy. Maksymalna długość okresu tego generatora wynosi m. Jeżeli w generowanym ciągu jakaś liczba pojawi się drugi raz, to dalszy ciąg będzie powtórzeniem poprzedniego. Okres generowanych liczb może być różny, nie może być on jednak zbyt krotki. Jeżeli okres jest krótki, to wtedy liczby:

$$U_i = \frac{x_i}{m},$$

będą zbyt rzadko wypełniały przedział $(0, 1)$ i taki algorytm nie może być wykorzystywany jako generator liczb pseudolosowych o rozkładzie równomiernym $u(0, 1)$. Generator LCG jest dość dobrze zbadany, znamy twierdzenie o jego maksymalnym okresie. Niech generator LCG ma określone wartości m, a, c i x_0. Generowany ciąg liczbowy ma okres równy m wtedy i tylko wtedy, gdy:

- c i m nie maja wspólnych dzielników,
- $b = a - 1$ jest wielokrotnością każdej liczby pierwszej p, która jest dzielnikiem liczby m,
- b jest wielokrotnością 4, o ile m jest też wielokrotnością 4.

Zestawem dobrych parametrów generatora są następujące wartości:

$$a = 69069, c = 1, m = 2^{32}.$$

Generatory liczb pseudolosowych zawsze wyprodukują ciąg liczb okresowy. Długość takiego okresu zależy od implementacji. Należy jednak pamiętać, że do obliczeń i symulacji wybierać należy krótkie, w porównaniu z długością okresu, części całego ciągu.

Konstrukcja generatora liczb pseudolosowych jest prosta. Należy pamiętać jedynie o zakresach typów użytych do przechowywania liczb. Najczęściej generuje się liczby całkowite, stosujemy wtedy typ `unsigned long int`. Przenośna wersja generatora liczb pseudolosowych może mieć następującą postać przedstawioną na Listingu 9.1.

Listing 9.1: Prosty generator LCG.

```
1  #include<iostream>
2  #include<cstdio>
3
4  const int N = 10;  // ilosc wygenerownych liczb
5
6  unsigned long int psrand();
7
8  int main()
9  {
10   for (int i = 0; i < N; ++i)
11     std::cout << psrand() << std::endl;
12   return 0;
13 }
14
15 unsigned long int psrand()
16 {
17   static unsigned long int seed = 1;
18   seed = seed * 1103515245 + 12345;
19   return (seed/65536) % 32768;
20 }
```

Funkcja `psrand()` generuje liczby pseudolosowe. Kolejna liczba pseudolosowa generowana jest z poprzedniej. Wartość początkowa musi być podana przez użytkownika. W naszym przypadku jest to wartość równa jeden. Ta liczba początkowa nosi nazwę „ziarno" (ang. *seed*). Ponieważ ziarno musi być pamiętane, należy użyć zmiennej statycznej:

```
1   static unsigned int seed = 1;
```

Funkcja `psrand()` zwraca liczbę z zakresu od 0 do 32767. Uruchomieniu programu daje następujący wynik:

```
1   16838
2   5758
3   10113
4   17515
5   31051
6   5627
7   23010
8   7419
9   16212
10  4086
```

Na pierwszy rzut oka liczby te wyglądają na przypadkowe. Ponowne uruchomienie generatora powoduje otrzymanie takich samych liczb. Jest to oczywiste, ponieważ liczby otrzymywane są na podstawie operacji arytmetycznych:

```
1   static unsigned long int seed = 1;
2   seed = seed * 1103515245 + 12345;
3   return (seed/65536) % 32768;
```

Łatwo zauważyć, że otrzymamy inne wyniki, gdy zmienimy wartość początkową ziarna, np. zmieniając jeden na pięć:

```
1   static unsigned int seed = 5;
```

Po uruchomieniu generatora, początkowe liczby wyglądają następująco:

```
1   18655
2   8457
3   10616
4   31877
5   10193
6   25964
7   18104
8   23667
9   32572
10  19560
```

Generatory liczb pseudolosowych mają wbudowane mechanizmy pozwalające nadać ziarnu dowolną wartość. W poważnych symulacjach wymagane są generatory liczb pseudolosowych o doskonałych parametrach, szybkie i o długim okresie. Istnieje bogata literatura na ten temat. Do celów testowych każda implementacja języka C/C++ zawiera funkcje biblioteczne służące do generacji liczb pseudolosowych. Często zamiast generatora LCG korzystamy z generatora multiplikatywnego liniowego kongruentnego, zwanego w skrócie MLCG. Jest on znacznie szybszy niż

generator LCG. Należy pamiętać, że generator MLCG nie produkuje zera. Okresy generatorów MLCG w typowych przypadkach są krótsze niż generatorów LCG. Jeżeli we wzorze (9.3) parametr $c = 0$, to otrzymamy wzór na kolejną liczbę pseudolosową:

$$x_i = (ax_i) \mod m, \tag{9.4}$$

Okres tego generatora nie może przekraczać m. Jeżeli na przykład:

$$m = 2^L,$$

to okres tego generatora nie przekracza wartości 2^{L-2}. W przypadku gdy m jest liczbą pierwszą to maksymalny okres generatora multiplikatywnego jest równy $m-1$. Jeżeli stała $m = 2L$, generator osiąga maksymalny okres wtedy, gdy x_0 jest liczbą nieparzystą oraz:

$$a = 3 \mod 8 \text{ lub } a = 5 \mod 8.$$

Historyczny generator *RANDU* (jeden z pierwszych) implementowany na maszynach IBM360/370 osiągał maksymalny okres dla następujących parametrów:

$$a = 2^{16} + 3, m = 2^{31} \tag{9.5}$$

W symulacjach komputerowych chcemy, aby okresy generatorów było wystarczająco długie. Aby zwiększyć okres generatorów liczb pseudolosowych korzystamy z następującego wzoru obliczeniowego:

$$x_i = (a_1 x_{i-1} + \cdots + a_r x_{i-r}) \mod m, \tag{9.6}$$

Gdzie $r > 1, a_r \neq 0$. Okresem generatora jest najmniejsza liczba dodatnia, dla której mamy:

$$(x_0, \ldots, x_{r-1}) = (x_\lambda, \ldots, x_{\lambda+r-1}), \tag{9.7}$$

Jeżeli $r = 2, a_1 = a_2 = 1$, wtedy otrzymujemy tzw. generator Fibonacciego:

$$x_i = (x_{i-1} + x_{i-2}) \mod m. \tag{9.8}$$

Widzimy, że we wzorze nie ma mnożenia, generator tego typu jest bardzo szybki. Przypominamy, że ciąg rekurencyjny postaci:

$$\begin{cases} f_0 = f_1 = 1 : i \leqslant 2, \\ f_i = f_{i-1} + f_{i-2} : i > 2, \end{cases} \tag{9.9}$$

opublikował włoski matematyk Fibonacci w 1202 roku, od jego nazwiska nazywamy opisany generator.

Jak już mówiliśmy, generatory zazwyczaj produkują liczby całkowite x_i. W zastosowaniach standardem jest generator o rozkładzie równomiernym, to znaczy liczby pseudolosowe znajdują się w przedziale $(0, 1)$. Typowo, taki ciąg możemy otrzymać przy pomocy normalizacji:

$$u_n = \frac{x_n}{m} \tag{9.10}$$

Liczby pseudolosowe możemy wygenerować w dowolnym przedziale wykonując skalowanie generatora $U(0,1)$:

$$RN_{scal} = (MAX - MIN)RN + MIN \tag{9.11}$$

MAX oznacza maksymalną żądaną liczbę, MIN oznacza minimalną żądaną liczbę, RN_{scal} jest skalowaną liczbą pseudolosową, RN jest liczba pseudolosową produkowaną przez generator równomierny $U(0,1)$. W zasadzie nie dysponujemy pełną teorią generatorów liczb pseudolosowych, stąd wybór najlepszych parametrów jest oparty na wieloletnim doświadczeniu i intuicji. W Tabeli 9.1 pokazano rekomendowane parametry generatorów liniowych. W literaturze przedmiotu znajdziemy

Tabela 9.1: Polecane parametry generatorów liniowych.

a	b	c
69069	1	2^{32}
16807	0	$2^{31}-1$
630360016	0	$2^{31}-1$
397204094	0	$2^{31}-1$
410092949	0	2^{32}
742938285	0	$2^{31}-1$
1099087573	0	2^{32}
40692	0	$2^{31}-249$

polecane generatory, posiadające dobre właściwości:

- $x_n = (1176x_{n-1} + 1476x_{n-2} + 1776x_{n-3}) mod (2^{32} - 5)$,
- $x_n = 213(x_{n-1} + x_{n-2} + x_{n-3}) mod (2^{32} - 5)$,
- $x_n = (1995x_{n-1} + 1998x_{n-2} + 2001x_{n-3}) mod (2^{35} - 849)$

Wymienione generatory posiadają długi okres rzędu $m^3 - 1$.

W poważnych symulacjach wymagane są generatory liczb pseudolosowych o doskonałych parametrach, szybkie i o długim okresie. Istnieje bogata literatura na ten temat. Do celów testowych każda implementacja języka C/C++ zawiera funkcje biblioteczne służące do generacji liczb pseudolosowych. W pliku nagłówkowym `cstdlib` znajdują się funkcje, makra i stałe związane z produkowaniem liczb pseudolosowych:

- `rand()`,
- `srand()`,
- `RAND_MAX`.

9.2.1 Generator liczb pseudolosowych rand()

Bardzo wygodnym i bardzo często używanym do najrozmaitszych zadań generatorem liczb pseudolosowych jest biblioteczny generator `rand()`. Deklaracja generatora jest następująca:

```
1  int rand();
```

W wyniku wywołania generatora zwracana jest liczba pseudolosowa z zakresu od 0 do `RAND_MAX`. Stała `RAND_MAX` jest zdefiniowana w pliku `cstdlib` i jej wartość jest zależna od implementacji, jednak zwykle jest równa `INT_MAX` (największej możliwej liczbie typu `int`).

Listing 9.2: Generator liczb pseudolosowych – `rand()`

```
1 #include<cstdlib>
2 #include<iostream>
3
4 int main()
5 {
6   for(int i = 0; i < 10; i++)
7     std::cout << std::rand() << std::endl;
8   return 0;
9 }
```

Wynik programu zależy od kompilatora i komputera na którym zostanie uruchomiony. Po uruchomieniu programu na komputerze autorów mamy następujący wynik:

```
1   1804289383
2   846930886
3   1681692777
4   1714636915
5   1957747793
6   424238335
7   719885386
8   1649760492
9   596516649
10  1189641421
```

Widzimy, że liczbę losową otrzymamy przez proste wywołanie funkcji:

```
1  std::rand();
```

a wartości wytwarzane bezpośrednio przez `rand()` są zawsze w zasięgu:

$$0 \leqslant \texttt{std::rand()} \leqslant \texttt{RAND\_MAX}$$

Bardzo często potrzebujemy innego zakresu generowanych liczb, np. przy symulacji rzutów kostką sześcienną otrzymujemy tylko liczby 1, 2, 3, 4, 5 i 6. Aby wygenerować zadany zakres liczb, należy dokonać skalowania:

```
1  a + std::rand() % b;
```

`a` jest wartością przesunięcia, `b` jest czynnikiem skalowania (jest równy szerokości zakresu kolejnych liczb całkowitych). Aby symulować rzut kostką sześcienną nasze skalowanie ma postać:

```
1 1 + std::rand() % 6;
```

Gdy chcemy mieć zakres liczb pseudolosowych od 0 do 99 to używamy instrukcji:

```
1 std::rand() % 100;
```

9.2.2 Inicjalizacja generatora liczb pseudolosowych srand()

Jak już wiemy, proste uruchamianie generatora liczb pseudolosowych daje w wyniku taki sam ciąg liczb. Dzieje się tak, dlatego, że generator startuje zawsze z tej samej wartości ziarna. Jeżeli będziemy zmieniali ziarno, będziemy mogli otrzymywać różne ciągi liczbowe. W języku C/C++ jest specjalny mechanizm do zmiany ziarna generatora `rand()`. Służy do tego funkcja `srand()`. Deklaracja funkcji `srand()` jest następująca:

```
1 void srand(unsigned int seed);
```

Funkcja `srand()` potrzebuje odpowiedniego parametru. W tym celu wykorzystano zegar systemowy. Biblioteka C++ zawiera funkcję `time()`, która zwraca czas systemowy. Zwracana wartość jest liczbą i jej wartość zmienia się w czasie. W programie symulującym rzuty kostką sześcienną przez dwóch graczy wykorzystaliśmy funkcję `srand()` i zegar systemowy.

Listing 9.3: Liczby pseudolosowe.

```
1  #include<cstdlib>
2  #include<ctime>
3  #include<iostream>
4
5  int main()
6  {
7    std::srand(std::time(NULL));
8    for(int i = 0; i < 4; i++)
9      {
10       std::cout << "rzut " << i + 1 << ": "
11                 << 1 + std::rand() % 6 << " "
12                 << 1 + std::rand() % 6 << std::endl;
13     }
14   return 0;
15 }
```

Po uruchomieniu programu mamy następujący wynik:

```
1 rzut 1: 1 4
2 rzut 2: 2 1
3 rzut 3: 6 1
4 rzut 4: 6 4
```

Pętla **for** powtarzana jest cztery razy, za każdym razem losowane są dwie liczby pseudolosowe z zakresu od 1 do 6. Każdorazowo po uruchomieniu programu otrzymamy inny wynik – mamy rzeczywistą symulację gry w kości. Za każdym razem zmienia się ziarno w generatorze `rand()`. Tą sytuację powoduje użycie wyrażenia takiego jak:

```
1   std::srand(std::time(NULL));
```

Funkcja `srand()` jest wywoływana tylko raz w programie, aby dać pożądany rezultat losowy. Nie ma potrzeby wywoływania jej wielokrotnie. Do programu należy włączyć plik nagłówkowy `ctime`, w którym umieszczona jest funkcja `time()`.

9.2.3 Ustalanie zakresu generowanych liczb pseudolosowych

W wielu przypadkach potrzebujemy liczb pseudolosowych z konkretnego zakresu. Jak już wiemy najbardziej uniwersalnym sposobem generowania takich liczb jest użycie funkcji `rand()`. Funkcja ta dostarcza liczbę całkowitą z przedziału [0;`RAND_MAX`]. Stała `RAND_MAX` powinna być nie mniejsza niż maksymalna wartość liczby typu **int**, czyli 32767. Aby otrzymać liczbę pseudolosową z przedziału $[0;1]$ należy wygenerowaną przez `rand()` liczbę podzielić przez `RAND_MAX + 1`. Taką liczbę należy pomnożyć przez $n+1$, aby otrzymać liczbę z przedziału $[0;n]$. Mamy trzy przypadki:

- `rand()` $\rightarrow [0;$`RAND_MAX`$]$,
- `rand()/ (RAND_MAX + 1)` $\rightarrow [0;1]$,
- `rand()/ (RAND_MAX + 1)(n + 1)` $\rightarrow [0;n]$.

Praktyczna realizacja generatora liczb pseudolosowych z zakresu $[0;1]$ pokazana jest na Listingu 9.4. Generowanie liczb pseudolosowych wykonywane jest za pomocą funkcji `rg_1()`. Funkcja `rg_1()` jest typu **double** i nie ma parametrów wejściowych – zakres generowanych liczb zawsze jest w przedziale $[0;1]$. Ta funkcja wykorzystuje funkcję `srand()`, aby losowanie było zmienne w czasie (ziarno obliczane jest na podstawie wskazań zegara systemowego).

Listing 9.4: Liczby pseudolosowych z zakresu $[0;1]$.

```
1  #include<cstdlib>
2  #include<iostream>
3  #include<ctime>
4
5  double rg_1();
6
7  int main()
8  {
9    for (int i = 0; i < 20; i++)
10     std::cout << rg_1() << std::endl;
11   return 0;
12 }
13 double rg_1()
14 {
15   static int flag = 1;
16   if (flag)
17     {
```

```
18        std::srand(time(NULL));
19        flag = 0;
20      }
21    return std::rand() / (double)(RAND_MAX + 1.0);
22  }
```

Praktyczna realizacja generatora liczb pseudolosowych z zakresu $[0; n]$ pokazana jest na Listingu 9.5. Generowanie liczb pseudolosowych wykonywane jest za pomocą funkcji `rg_n()`. Funkcja `rg_n()` ma jeden parametr – zakres generowanych liczb. Ta funkcja wykorzystuje funkcję `srand()`, aby losowanie było zmienne w czasie (ziarno obliczane jest na podstawie wskazań zegara systemowego). Program wyświetla 20 liczb pseudolosowych z zakresu podanego przez użytkownika.

Listing 9.5: Liczby pseudolosowych z zakresu $[0; n]$.

```
1  #include<cstdlib>
2  #include<iostream>
3  #include<ctime>
4
5  const int N = 10;
6
7  int rg_n(int);
8
9  int main()
10 {
11   for (int i = 0; i < 20; i++)
12     std::cout << rg_n(N) << std::endl;
13   return 0;
14 }
15
16 int rg_n(int n)
17 {
18   static int flag = 1;
19   if (flag)
20     {
21       std::srand(std::time(NULL));
22       flag = 0;
23     }
24   return rand() / (RAND_MAX + 1.0) * (n + 1);
25 }
```

9.3 Testowanie generatorów Monte Carlo

Głównym zadaniem w konstruowaniu generatorów Monte Carlo jest otrzymanie sekwencji liczb, które są losowe. Powstaje pytanie jak sprawdzić, czy wyprodukowany przez generator Monte Carlo ciąg liczbowy zachowuje się jak ciąg liczb losowych. Przyjmuje się, że generator jest uznany jako dobry, gdy pomyślnie przejdzie odpowiednio skonstruowane testy.

Wygenerowana sekwencja liczb aby była użyteczna, musi mieć odpowiednią długość, należy zatem sprawdzić długość (okres) wygenerowanego ciągu.

Jedynym sposobem ustalenia czy generator dobrze produkuje liczby pseudolosowe jest poddanie go serii testów. Jednym z ważniejszych testów jest test widmowy.

Jest to bardzo prosty test. Z wygenerowanego ciągu liczb wybieramy kolejne pary sąsiednich liczb:

$$(x_n, x_{n+1}) : n = 0, 1, \ldots, k-1,$$

gdzie liczba k jest równa długości okresu. Następnie pary liczb (x_k, x_{k+1}) traktujemy jako współrzędne punktu na płaszczyźnie. Wyświetlenie tych punktów na przykład na ekranie monitora, pozwala stwierdzić czy, nie występuje zjawisko grupowania się punktów, co świadczy o złej jakości generatora. Podstawowymi testami są testy statystyczne. Znamy wiele takich testów, najważniejsze z nich to:

- Test średniej arytmetycznej.
- Test częstości.
- Test przerw.
- Test serii.
- Test kombinatoryczny (test pokerowy).
- Różne zadania kontrolne.

Jeżeli mamy ciąg liczbowy:

$$x_1, x_2, \ldots, x_n,$$

to średnia z tego ciągu wyraża się wzorem:

$$\bar{x}_n = \frac{1}{n} \sum_{i=1}^{n} x_i. \tag{9.12}$$

Jeżeli wszystkie wartości liczb pseudolosowych są niezależne oraz mają rozkład równomierny na przedziale $(0, 1)$, wtedy wartość oczekiwana średniej jest równa 0,5.

Zgodnie z centralnym twierdzeniem granicznym, dla dużych n, zmienna losowa jaką jest wartość średnia $\bar{x}$ ma rozkład normalny:

$$N\left(\frac{1}{2}, \frac{1}{\sqrt{12n}}\right)$$

Statystyka:

$$U = (\bar{x} - 0,5)\sqrt{12n}, \tag{9.13}$$

ma rozkład normalny $N(0, 1)$. Postępując zgodnie z zasadami testowania hipotez statystycznych, należy z tablic rozkładu normalnego $N(0, 1)$ wyznaczyć taką wartość u_α aby dla założonego poziomu istotności α zachodziła równość:

$$P\{|U| \geqslant u_\alpha\} = \alpha. \tag{9.14}$$

Mamy do czynienia z dwustronnym obszarem krytycznym, zatem wartość u_α określa wartość U takich, że mamy warunek:

$$|U| \geqslant u_\alpha. \tag{9.15}$$

Jeżeli przyjmiemy, że $\alpha = 0,05$ to z tablic rozkładu normalnego mamy wartość $u_\alpha = 1,96$. Obszar krytyczny jest, zatem określony następująco:

$$(-\infty; -1,96] \cup [1,96, \infty).$$

Aby sprawdzić hipotezę o wartości średniej należy obliczyć statystykę u i sprawdzić, czy znajduje się ona w obszarze krytycznym, tzn. czy:

$$|u| \geqslant u_\alpha.$$

Weryfikując hipotezy często posługujemy się testem chi-kwadrat zgodności rozkładu. W testach zgodności chi-kwadrat, hipoteza styczna ma postać: *zmienna losowa X ma rozkład prawdopodobieństwa o dystrybuancie F*. Niech a i b będą takimi liczbami, że:

$$F(a) = 0 \wedge F(b) = 1. \tag{9.16}$$

Oznaczmy symbolem a rozbicie zbioru wartości zmiennej losowej X i niech:

$$a = a_1 < a_2 < \cdots < a_k = b. \tag{9.17}$$

Oznaczmy symbolem p prawdopodobieństwa, takie, że:

$$P = \{a_{i-1} < X < a_i\}, i = 1, 2, \dots , \tag{9.18}$$

Następnie oznaczmy przez n_i liczbę takich elementów X ciągu $X_1, X_2, \dots, X_n$, które spełniają warunek:

$$a_{i-1} < X < a_i. \tag{9.19}$$

Statystyką testu nazywanego chi-kwadrat (χ^2) jest:

$$\chi^2_{k-1} = \sum_{i=1}^{k} \frac{(n_i - np_i)^2}{np_i}, \tag{9.20}$$

gdzie k jest liczbą klas, p_i jest prawdopodobieństwem przyjęcia przez zmienną losową X wartości z i-tej klasy, n_i jest liczebnością i-tej klasy. Dla dużych n statystyka (9.20) ma rozkład chi-kwadrat o $k-1$ stopniach swobody. Zerowa wartość χ^2 oznacza doskonały generator, duża wartość oznacza, że nasza hipoteza nie jest prawdziwa.

Weryfikację hipotezy o równomierności rozkładu liczb pseudolosowych otrzymanych z badanego generatora można przeprowadzić sprawdzając równomierność rozkładu cyfr na poszczególnych pozycjach. Do testów należy wyznaczyć wartość statystyki:

$$\chi^2_9 = \frac{10}{n} \sum_{i=1}^{10} (l_i - \frac{n}{10})^2. \tag{9.21}$$

W tym wzorze l_i jest liczbą wystąpień cyfry i na danej pozycji. Otrzymaną na podstawie powyższego wzoru wartość należy porównać z wartością krytyczną otrzymaną z tablic rozkładu χ^2 (dla ustalonej liczby swobody $k-1$ i poziomu istotności α). W omawianym przykładzie, wartość krytyczna wynosi 16,92 (dla $k = 9$ i $\alpha = 5\%$).

Test częstości jest bardzo dobrym testem, pozwala na przykład określić ile razy każda cyfra dziesiętna $(0, 1, 2, \ldots, 9)$ występuje na zakładanej pozycji. Możemy na przykład postawić pytanie ile razy cyfra 3 pojawia się na drugim miejscu. Możemy także postawić pytanie – ile liczb wyprodukował generator w zadanym przedziale. Jeżeli mamy generator o rozkładzie równomiernym na przedziale $(0, 1)$ to możemy zapytać się ile liczb znalazło się w przedziale 0.0 – 0.1, ile w przedziale 0.1 – 0.2, itd.

Omówimy praktycznie taki test. Pewien generator wyprodukował 1000 liczb (7 cyfr znaczących). Analiza częstościowa dała wynik zamieszczony w Tabeli 9.2. W przypadku idealnego generatora, w każdym przedziale powinno być 100 po-

Tabela 9.2: Przykładowy wynik testu częstości.

Przedział	Ilość liczb
0,0 – 0,1	99
0,1 – 0,2	114
0,2 – 0,3	100
0,3 – 0,4	126
0,4 – 0,5	95
0,5 – 0,6	85
0,6 – 0,7	100
0,7 – 0,8	89
0,8 – 0,9	82
0,9 – 1,0	110

wtórzeń. Ponieważ liczby są losowe, musimy przeprowadzić analizę statystyczną otrzymanych wyników. Zastosujemy test χ^2. Statystyka ma postać:

$$\chi^2 = \sum_{i=1}^{10} \frac{(F(i) - 100)^2}{100}. \tag{9.22}$$

Wykonując powyższe sumowanie otrzymujemy następujący wynik:

$$\chi^2 = 16,680. \tag{9.23}$$

Mamy 9 stopni swobody. Z tabel statystycznych mamy, że dla 9 stopni swobody i 5% przedziału ufności χ^2 ma wartość 16,92. Ponieważ obliczona wartość statystyki dla liczb wyprodukowanych z naszego generatora ma wartość 16,68, czyli jest mniejsza niż wartość odczytanej z tabel, hipoteza o rozkładzie jednorodnym jest zaakceptowana.

Jakość generatorów liczb pseudolosowych sprawdzamy także przy pomocy testów kombinatorycznych. Jednym z takich testów jest test pokerowy. Ten test należy do grupy testów niezależności.

Test generatora równomiernego na przedziale $(0, 1)$ rozpoczynamy, od przekształcenia ciągu liczb pseudolosowych $\{x_i\}$ na ciąg $\{y_i\}$ w taki sposób, że y_i jest pierwszą cyfrą po przecinku liczby x_i. Zatem nasz nowy ciąg składa się wyłącznie

z cyfr $0, 1, 2, \ldots, 9$. Tak otrzymany ciąg dzielimy na grupy 5 cyfr:

$$(y_0, y_1, y_2, y_3, y_4, y_5), (y_6, y_7, y_8, y_9, y_{10}), \ldots \ . \tag{9.24}$$

Możemy wyróżnić kombinacje zaprezentowane w Tabeli 9.3 (w nawiązaniu do popularnej gry w karty o nazwie poker). Test polega na sprawdzeniu, czy występujące

Tabela 9.3: Wyróżnione kombinacje.

Nr	Sekwencja	Opis	P (sekwencja)
1	abcde	Każda liczba jest inna (bust).	0,3024
2	aabcd	Dwie liczby identyczne, pozostałe różne (para).	0,5040
3	aabbc	Dwie pary identycznych liczby (dwie pary).	0,1080
4	aaabc	Trzy identyczne liczby, pozostałe są różne (trójka).	0,072
5	aaabb	Trójka i para.	0,009
6	aaaab	Cztery spośród pięciu liczb są identyczne (czwórka).	0,0045
7	aaaaa	Wszystkie liczby są identyczne (piątka).	0,0001

kombinacje tworzone z liczb pseudolosowych nie odbiegają znacznie od teoretycznych wartości – musimy sprawdzić hipotezę o rozkładzie.

Rozkład równomierny na przedziale $(0, 1)$ jest podstawowym rozkładem stosowanym w obliczeniach komputerowych. W symulacjach komputerowych zjawisk fizycznych, zjawisk chemicznych, badaniach ekonomicznych i innych praktycznie potrzebujemy generatorów liczb pseudolosowych o innych rozkładach niż równomierny. W teorii pomiarów fizycznych ważną rolę odgrywa rozkład normalny (Gaussa), mierzone wartości będą skupiały się wokół wartości średniej. W analizie układów telekomunikacyjnych ważną rolę odgrywa rozkład Poissona. Znamy wiele sposobów generowania liczb pseudolosowych o zadanym rozkładzie, najważniejsze z nich to:

- Metoda odwracanej dystrybuanty.
- Metoda von Neumanna (akceptacja i odrzucanie).
- Metoda addytywna.
- Inne metody (np. metoda ciągów monotonicznych).

W metodzie odwracanej dystrybuanty korzystamy z faktu, że jeżeli zmienna losowa R ma rozkład równomierny na przedziale $(0, 1)$ oraz jest dana dystrybuanta $F(x)$ to zmienna losowa:

$$X = F^{-1}(R),$$

ma rozkład o dystrybuancie F (zapis F^{-1} oznacza funkcje odwrotną do F).

Opisaną metodę zilustrujemy algorytmem generowania liczb pseudolosowych o rozkładzie wykładniczym. Jest to ważny rozkład, wykorzystywany jest on na przykład do opisu rozpadu promieniotwórczego. Do symulacji rozpadu promienio-

twórczego potrzebne są liczby pseudolosowe opisywane gęstością prawdopodobieństwa postaci:

$$\begin{cases} g(t) = 0 : t < 0, \\ g(t) = \dfrac{1}{\tau} \exp\left(\dfrac{-t}{\tau}\right) : t \geqslant 0. \end{cases} \tag{9.25}$$

We wzorze τ oznacza średni czas życia jądra promieniotwórczego, t – oznacza upływający czas. Dystrybuanta rozkładu ma postać:

$$x = G(t) = \frac{1}{\tau} \int_{t'=0}^{t} g(t')dt' = 1 - \exp\left(\frac{-t}{\tau}\right). \tag{9.26}$$

Funkcja odwrotna ma postać:

$$G^{-1}(x) = -\tau \ln(1 - x). \tag{9.27}$$

Algorytm generowania liczb pseudolosowych RN_w o rozkładzie wykładniczym ma następującą postać:

1. Generujemy pierwszą liczbę pseudolosową U_l z generatora równomiernego $(0, 1)$.
2. Obliczamy RN_w: $RN_w = -\tau \ln U_l$.

Generowanie liczb pseudolosowych metodą odwracanej dystrybuanty jest proste, niestety istnieje niewiele rozkładów, dla których potrafimy obliczyć funkcje odwrotne.

Von Neumann zaproponował następujący algorytm generowania liczb pseudolosowych X o rozkładzie danym funkcją $f(x)$:

1. Generujemy liczbę pseudolosową r_1 z rozkładu równomiernego na przedziale (a, b).
2. Generujemy liczbę pseudolosową r_2 z rozkładu równomiernego na przedziale $(0, c)$.
3. Jeżeli $r_2 \leqslant f(r_1)$ przyjmujemy r_1 jako liczbę pseudolosową o żądanym rozkładzie.
4. Jeżeli jest przeciwnie, idziemy do punktu pierwszego.

W kolejnej metodzie zwanej metodą addytywną wyznaczenie liczby pseudolosowej o danym rozkładzie F polega na utworzeniu odpowiedniej sumy niezależnych liczb pseudolosowych o rozkładzie równomiernym. Jako przykład rozważmy generowanie liczb pseudolosowych o rozkładzie normalnym. Rozkład normalny (Gaussa) z wartością średnią równą zero i odchyleniem standardowym σ definiujemy następująco:

$$\varphi(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma}\right). \tag{9.28}$$

Gdy wartość średnia rozkładu ma wartość μ, gęstość prawdopodobieństwa ma postać $\varphi(x-\mu)$. Standardowe odchylenie wyraża się wzorem

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} x(i)^2}{N}}. \tag{9.29}$$

Liczby pseudolosowe o rozkładzie normalnym mogą być otrzymane przez sumowanie N liczb pseudolosowych o rozkładzie równomiernym. Aby otrzymać liczby pseudolosowe o rozkładzie normalnym z wartością średnią równą zero i wariancją równą 1, wartość $N\mu$ musi być odjęta od sumy i różnica podzielona przez pierwiastek kwadratowy z $N\sigma^2$. Dla rozkładu równomiernego na przedziale $(0,1)$ wartość średnia $\mu = 0,5$. Wariancja jest obliczana ze wzoru:

$$\mathrm{var}(x) = \sigma^2 = E[x^2] - \mu^2. \tag{9.30}$$

A oczekiwana wartość E dana jest wzorem:

$$E[x^2] = \int_0^1 x^2 f(x) dx. \tag{9.31}$$

Dla rozkładu równomiernego na przedziale $(0,1)$, $f(x) = 1$. Wobec tego mamy:

$$E[x^2] = \int_0^1 x^2 dx = \frac{1}{3}. \tag{9.32}$$

Możemy także obliczyć wariancję:

$$\mathrm{var}(x) = \sigma^2 = \frac{1}{3} - \left(\frac{1}{2}\right)^2 = \frac{1}{12}. \tag{9.33}$$

Liczba pseudolosowa RN_{normal} o rozkładzie normalnym generowana jest następująco:

$$RN_{normal} = \frac{\sum_{i=1}^{n} RN(i) - N\mu}{\sqrt{N\sigma^2}}. \tag{9.34}$$

Jeżeli ustalimy, że $N = 20$, to wzór na generowanie liczb pseudolosowych o rozkładzie normalnym ma postać:

$$RN_{normal} = \frac{\sum_{i=1}^{20} RN(i) - 10 \cdot 0,5}{\sqrt{20/12}}. \tag{9.35}$$

Do wyprodukowania jednej liczby pseudolosowej o rozkładzie normalnym potrzebujemy w tym przykładzie 20 liczb pseudolosowych o rozkładzie równomiernym. Możemy zauważyć, że całkiem dobry generator liczb pseudolosowych o rozkładzie normalnym otrzymamy korzystając tylko z sumowania 112 liczb pseudolosowych o rozkładzie równomiernym. W tej sytuacji wzór (9.35) upraszcza się do:

$$RN_{normal} = r_1 + r_2 + \cdots + r_{112} - 6. \tag{9.36}$$

9.4 Całkowanie metodą Monte Carlo

Istnieje wiele doskonałych metod całkowania numerycznego. Jedną z takich metod jest metoda Monte Carlo. W zasadzie metoda Monte Carlo polecana jest do rozwiązywania całek wielokrotnych oraz całek ze skomplikowanych funkcji o trudno określonych w formie analitycznej granicach całkowania.

Metody Monte Carlo nie dają ścisłych wyników, trudno jest oszacować błąd metody, ale czasami lepiej jest mieć jakikolwiek wynik niż żaden.

W niniejszym skrypcie omówimy dwie metody:

- Metodę prostego próbkowania.
- Metodę próbkowania średniej.

9.4.1 Metoda prostego próbkowania

Idea całkowania za pomocą metody próbkowania prostego jest nieskomplikowana. Aby przybliżyć ideę tej metody posłużymy się opisem eksperymentu. Niech na tarczy o znanych wymiarach (na przykład kwadrat o boku 1 metr) będzie narysowana dowolna figura (na przykład elipsa) o nieznanej powierzchni. Naszym zadaniem jest oszacowanie pola tej figury. Proponuje się następujące rozwiązanie. Wykonuje się serię strzałów do tarczy. Jeżeli założymy, że pociski będą uderzały w tarczę równomiernie, to znając liczbę trafień w figurę n oraz całkowitą liczbę N oddanych strzałów możemy obliczyć stosunek n/N. W ten sposób oszacujemy pole powierzchni figury.

Naszym zadaniem jest oszacowanie metodą Monte Carlo następującej całki:

$$J = \int_a^b f(x)dx. \tag{9.37}$$

W naszych rozważaniach opieramy się na geometrycznej interpretacji całki oznaczonej – traktujemy całkę jako pole powierzchni pod krzywą $f(x)$. Na płaszczyźnie mamy zaznaczony prostokąt o wysokości H i szerokości $(b-a)$ oraz wykres funkcji $f(x)$. Spełniony jest warunek:

$$f(x) < H : x \in [a; b]. \tag{9.38}$$

Na Rysunku 9.1 zacienione pole odpowiada wartości całki (9.37). Jeżeli zostaną wygenerowane w n-tym losowaniu punkty o współrzędnych (x_n, y_n) i liczby pseudolosowe o rozkładzie równomiernym x_n i y_n spełniają warunek:

$$a < x_n < b \wedge 0 < y_n < H, \tag{9.39}$$

to możemy oszacować całkę (9.37):

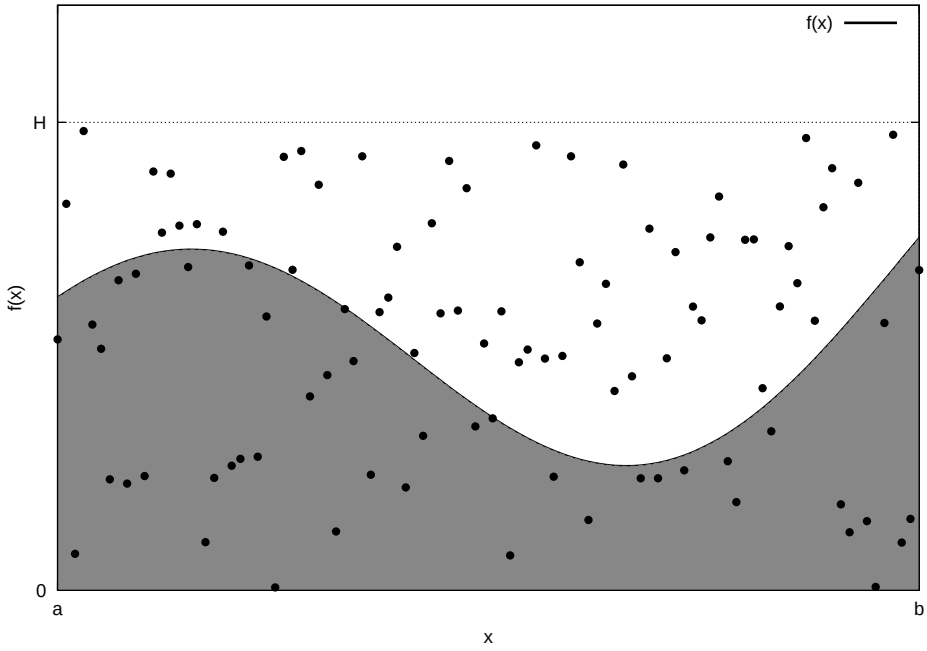
$$I = H(b-a)\frac{n}{N}, \tag{9.40}$$

gdzie I oznacza przybliżenie całki J, N oznacza liczbę wygenerowanych punktów próbkujących zdefiniowaną płaszczyznę, a n jest liczbą punktów, które spełniają

warunek:

$$y_i < f(x_i). \tag{9.41}$$

Bardzo często aby uprościć algorytmy szacowania całek metodą Monte Carlo wpro-



Rysunek 9.1: Ilustracja całkowania metodą prostego próbkowania.

wadza się dodatkowe ograniczenia. Możemy założyć dodatkowo, że funkcja podcałkowa spełnia warunek:

$$0 \leqslant f(x) \leqslant 1. \tag{9.42}$$

Nie jest to istotne ograniczenie. Jeżeli funkcja $f(x)$ nie spełnia tego warunku, ale jest ograniczona, to przy pomocy przekształcenia:

$$\frac{f(x) - f_0}{f_1 - f_0}, \tag{9.43}$$

gdzie f_0 i f_1 są takimi liczbami, że dla każdego $0 \leqslant x \leqslant 1$ mamy $f_0 \leqslant f(x) \leqslant f_1$. Kolejnym uproszczeniem metody jest ograniczenie się do całkowania typu:

$$J = \int_0^1 f(x)dx. \tag{9.44}$$

Rozwiązywanie całki J po przedziale $(0, 1)$ nie jest istotnym ograniczeniem. Jeżeli mamy obliczyć całkę postaci:

$$\int_a^b g(u)du.$$

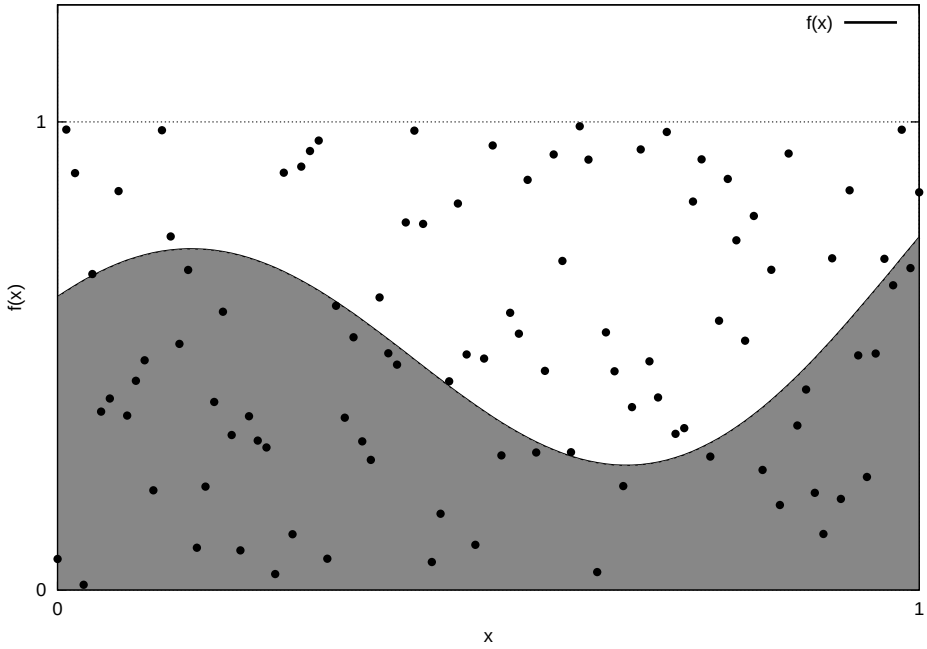
To przy pomocy podstawienia:

$$x = \frac{u-a}{b-a}$$

oraz wykorzystując podstawienie:

$$f(x) = ((b-a)g\Big((b-a)x + a\Big), \tag{9.45}$$

otrzymujemy żądaną postać całki J. Po tych wszystkich uproszczeniach możemy szacować całkę w kwadracie jednostkowym (Rysunek 9.2).



Rysunek 9.2: Ilustracja całkowania metodą prostego próbkowania.

Możemy sformułować następujące pytanie: *Niech w kwadrat* $(0 \leqslant x \leqslant 1)$, $(0 \leqslant y \leqslant 1)$ *trafia przypadkowy punkt, którego współrzędne o rozkładzie równomiernym są niezależne w przedziale* $(0, 1)$. *Jakie jest prawdopodobieństwo* P *tego, że punkt trafi w obszar leżący pod krzywą* $f(x)$*?* Dla dowolnego punktu (X, Y) leżącego w kwadracie jednostkowym jest oczywiste, że prawdopodobieństwo P jest równe polu S pod krzywą $f(X)$, czyli jest to oszacowanie metodą Monte Carlo całki (9.37). Praktyczny algorytm szacowania całki ma postać:

1. Uruchamiamy generator liczb pseudolosowych o rozkładzie równomiernym na przedziale $(0, 1)$.
2. Według tego rozkładu losujemy N punktów o współrzędnych (X_N, Y_N).
3. Dla każdej pary współrzędnych X_N i Y_N sprawdzamy, czy spełniony jest warunek $f(X) > Y$.

4. Po wylosowaniu N par i ustaleniu, że powyższy warunek został spełniony M razy obliczamy stosunek M/N.

5. Za przybliżoną wartość całki J przyjmujemy wielkość M/N.

Dla zilustrowania opisanej metody oszacujemy dwie znane całki:

$$J_1 = \int_0^1 \frac{dx}{1+x^2} = \frac{\pi}{4} = 0,78540,$$
$$J_2 = \int_0^1 \frac{dx}{\sqrt{2\pi}} \exp\left(\frac{-x^2}{2}\right) = 0,34135.$$

Na listingu 9.6 pokazany jest kod źródłowy do szacowania całki metodą Monte Carlo.

Listing 9.6: Całkowanie metodą Monte Carlo – próbkowanie proste.

```
#include<iostream>
#include<cstdlib>
#include<ctime>
#include<cmath>

const double PI = 3.1415926;

int xy1(double, double);
int xy2(double, double);

int main()
{
  std::srand(std::time(0));
  int n;
  int m1 = 0;
  int m2 = 0;
  double avg1, er1;
  double avg2, er2;
  std::cout << "poadj liczbe losowan N > 10, N = ";
  std::cin >> n;
  for (int i = 0; i < n; i++)
    {
      double x = std::rand() / (RAND_MAX + 1.0);
      double y = std::rand() / (RAND_MAX + 1.0);
      if (xy1(x, y))
        m1++;
      if (xy2(x, y))
        m2++;
    }
  avg1 = m1 / (double)n;
  avg2 = m2 / (double)n;
  er1 = std::sqrt(avg1 * (1 - avg1) / n);
  er2 = std::sqrt(avg2 * (1 - avg2) / n);
  std::cout << "calka 1 = " << avg1 << ", blad 1 = " << er1 << std::
       endl;
  std::cout << "calka 2 = " << avg2 << ", blad 2 = " << er2 << std::
       endl;
  return 0;
}

int xy1(double x, double y)
```

```
40 {
41   double fx = 1.0 / (1.0 + x * x);
42   if (fx >= y)
43     return 1;
44   return 0;
45 }
46
47 int xy2(double x, double y)
48 {
49   double fx = (1.0 / std::sqrt(2.0 * PI)) * std::exp(-(x * x) / 2.0);
50   if (fx >= y)
51     return 1;
52   return 0;
53 }
```

Oczekujemy, że im więcej będzie wygenerowanych punktów, tym bardziej dokładne będzie oszacowanie całki. W Tabeli 9.4 i Tabeli 9.5 mamy podane wartości oszacowania całki J_1 i J_2 dla różnych wartości N.

Tabela 9.4: Oszacowanie całki J_1 w zależności od wygenerowanej liczby punktów.

N = 100	N = 1000	N = 10000
0,84	0,773	0,7835
0,76	0,796	0,7857
0,81	0,802	0,7902
0,73	0,805	0,7882
0,71	0,786	0,7848
Śr. = 0,77	Śr. = 0,7924	Śr. = 0,78648

Tabela 9.5: Oszacowanie całki J_2 w zależności od wygenerowanej liczby punktów.

N = 100	N = 1000	N = 10000
0,32	0,313	0,3431
0,4	0,348	0,3389
0,33	0,378	0,3438
0,35	0,347	0,3428
0,27	0,332	0,3425
Śr. = 0,334	Śr. = 0,3436	Śr. = 0,34222

Oszacowanie błędu metody Monte Carlo jest zagadnieniem złożonym. Ogólnie rzec biorąc, górne oszacowanie błędu metody otrzymamy z pierwiastka kwadratowego z liczby prób. W monografii S. Brandta „Analiza danych" wykazano, że typowa fluktuacja liczby M jest w przybliżeniu równa:

$$\Delta M = \sqrt{M}. \tag{9.46}$$

Tak więc względna dokładność wyznaczania wartości całki (9.37) wynosi:

$$\frac{\Delta J}{J}\frac{\Delta M}{M} = \frac{1}{M}. \tag{9.47}$$

Możemy przyjąć, że maksymalny błąd obliczeń dla $N = 100, 10000, 1000000$ będzie równy odpowiednio 10%, 1% i 0.1%.

9.4.2 Metoda próbkowania średniej

Drugą metodę szacowania całek nazywamy metodą próbkowania średniej. Całkę J możemy rozpatrywać jako wartość oczekiwaną zmiennej losowej:

$$y = f(x), \tag{9.48}$$

gdzie x jest zmienną losową o rozkładzie równomiernym na przedziale $(0, 1)$. Oszacowaniem wartości oczekiwanej $E\{y\}$ zmiennej losowej y jest średnia z N niezależnych obserwacji y_i:

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{N} y_i = \frac{1}{N}\sum_{i=1}^{N} f(x_i). \tag{9.49}$$

Jeżeli wykonujemy całkowanie w granicach (a, b), a nie jak poprzednio w granicach $(0, 1)$, to mamy wyrażenie:

$$\bar{y} = (b-a)\frac{1}{n}\sum_{i=1}^{N} y_i = (b-a)\frac{1}{N}\sum_{i=1}^{N} f(x_i). \tag{9.50}$$

Procedura całkowania metodą próbkowania średniej jest następująca:

1. Uruchamiamy generator liczb pseudolosowych o rozkładzie równomiernym na przedziale $(0, 1)$.
2. Według tego rozkładu losujemy punkty $x_1, x_2, \ldots, x_N$.
3. Dla wylosowanych N punktów obliczamy wartości zadanej funkcji $f(x_1)$, $f(x_2), \ldots, f(x_N)$.
4. Obliczamy średnią z N wartości: $I = 1/N \sum_{i=1}^{N} f(x_i)$.
5. Uważamy, że obliczona średnia I jest oszacowaniem całki (9.37).

Błąd metody możemy szacować korzystając z centralnego twierdzenia granicznego. Dla dużych N mamy:

$$\sigma_I^2 \approx \frac{1}{N}\sigma_f^2 = \frac{1}{N}\left(\frac{1}{N}\sum_{i=1}^{n} f_i^2 - \left(\frac{1}{N}\sum_{i=1}^{n} f_i\right)^2\right), \tag{9.51}$$

gdzie σ_f^2 jest wariancją f.

Kod źródłowy programu do szacowania całek metodą próbkowania średniej pokazany jest na Listingu 9.7.

Listing 9.7: Całkowanie metodą Monte Carlo – próbkowanie średniej.

```
1 #include<iostream>
2 #include<cstdlib>
3 #include<ctime>
```

```
4  #include<cmath>
5
6  const double PI = 3.1415926;
7
8  double f1(double);
9  double f2(double);
10
11 int main()
12 {
13   std::srand(std::time(0));
14   int n;
15   double sum_f1, sum_f1pow;
16   double sum_f2, sum_f2pow;
17   double avg1, er1;
18   double avg2, er2;
19   std::cout << "poadj liczbe losowa N > 10, N = ";
20   std::cin >> n;
21   sum_f1 = sum_f1pow = 0.0;
22   sum_f2 = sum_f2pow = 0.0;
23   for (int i = 0; i < n; i++)
24     {
25       double x = std::rand() / (RAND_MAX + 1.0);
26       double f1x = f1(x);
27       double f2x = f2(x);
28       sum_f1 += f1x;
29       sum_f1pow += f1x * f1x;
30       sum_f2 += f2x;
31       sum_f2pow += f2x * f2x;
32     }
33   avg1 = sum_f1 / n;
34   avg2 = sum_f2 / n;
35   er1 = std::sqrt((sum_f1pow / n - avg1 * avg1) / n);
36   er2 = std::sqrt((sum_f2pow / n - avg2 * avg2) / n);
37   std::cout << "calka = " << avg1 << ", blad = " << er1 << std::endl;
38   std::cout << "calka = " << avg2 << ", blad = " << er2 << std::endl;
39   return 0;
40 }
41
42 double f1(double x)
43 {
44   return 1.0 / (1.0 + x * x);
45 }
46
47 double f2(double x)
48 {
49   return (1.0 / std::sqrt(2.0 * PI)) * std::exp(-(x * x) / 2.0);
50 }
```

9.4.3 Całki wielokrotne

Opisane powyżej techniki szacowania całek jednokrotnych metodą Monte Carlo można rozszerzyć na całki wielokrotne typu:

$$\int_G f(P)dP = \int \cdots \int_G f(x_1, x_2, \ldots, x_n)dx_1\, dx_2\, \ldots\, dx_n, \tag{9.52}$$

gdzie G jest dowolnym obszarem przestrzeni n-wymiarowej, a punkty:

$$P(x_1, x_2, \ldots, x_n)$$

należą do obszaru G. Aby oszacować całkę wielokrotną losujemy N punktów:

$$x_{1k}, x_{2k}, \ldots, x_{nk},$$

gdzie $k = 1, 2, \ldots, N$. Aby zilustrować metodę szacowania całek wielokrotnych oszacujemy następującą całkę (korzystać będziemy z metody prostego próbkowania):

$$2^{-3} \iiint_{x_1^2+x_2^2+x_3^2 \leqslant 1} dx_1\ dx_2\ dx_3 = \frac{\pi}{6} = 0,523599. \tag{9.53}$$

Obszarem całkowania jest kula o promieniu 1, środek kuli leży w początku układu współrzędnych. Dla x_i spełnione są następujące warunki:

$$-1 \leqslant x_1 \leqslant 1,$$
$$-1 \leqslant x_2 \leqslant 1,$$
$$-1 \leqslant x_3 \leqslant 1.$$

Algorytm szacowania całki może być następujący:

1. Uruchamiamy generator liczb pseudolosowych o rozkładzie równomiernym na przedziale $(0, 1)$.
2. Przy pomocy skalowania otrzymujemy liczby pseudolosowe o rozkładzie równomiernym na przedziale $(-1, 1)$.
3. Według tego rozkładu losujemy N punktów o współrzędnych (x_1, x_2, x_3).
4. Dla każdego wylosowanego punktu sprawdzamy, czy spełniony jest warunek: $x_1^2 + x_2^2 + x_3^3 \leqslant 1$.
5. Po wylosowaniu N punktów i ustaleniu, że powyższy warunek został spełniony M razy obliczamy wielkość M/N.
6. Za przybliżoną wartość całki uważamy wielkość:

$$M/N \approx 2^{-3} \iiint_G dx_1, dx_2, dx_3. \tag{9.54}$$

Na Listingu 9.8 pokazany jest kod źródłowy do obliczania całki potrójnej metodą Monte Carlo.

Listing 9.8: Całkowanie metodą Monte Carlo – potrójna metoda prostego próbkowania.

```
1 #include<iostream>
2 #include<cstdlib>
3 #include<ctime>
4 #include<cmath>
5
```

```
6  const double PI = 3.1415926;
7
8  int xy(double, double, double);
9
10 int main()
11 {
12   std::srand(std::time(0));
13   int n;
14   double avg, er1;
15   std::cout << "poadj liczbe losowan N > 10, N = ";
16   std::cin >> n;
17   int m = 0;
18   double norm = (RAND_MAX + 1.0);
19   for (int i = 0; i < n; i++)
20     {
21       double x1 = 2 * (std::rand() / norm) - 1;
22       double x2 = 2 * (std::rand() / norm) - 1;
23       double x3 = 2 * (std::rand() / norm) - 1;
24       if (xy(x1, x2, x3))
25         m++;
26     }
27
28   avg = m / (double)n;
29   er1 = std::sqrt(avg * (1 - avg) / n);
30   std::cout << "calka = " << avg << ", blad = " << er1 << std::endl;
31   return 0;
32 }
33
34 int xy(double x1, double x2, double x3)
35 {
36   double fx = x1*x1+x2*x2+x3*x3;
37   if (fx <= 1)
38     return 1;
39   return 0;
40 }
```

Do szacowania całek wielokrotnych możemy także użyć metody próbkowania średniego. Jeżeli P jest rozkładem równomiernym na G, czyli:

$$dP = \frac{dx_1\ dx_2\ \ldots\ dx_n}{|G|}, \tag{9.55}$$

gdzie $|G|$ jest miarą („objętością") G w przestrzeni n-wymiarowej. W celu oszacowania całki wielokrotnej losujemy N punktów z obszaru G zgodnie z rozkładem równomiernym:

$$x_{1k}, x_{2k}, \ldots, x_{nk}, \tag{9.56}$$

gdzie $k = 1, 2, \ldots, N$. Oszacowanie całki jest dane wzorem:

$$\frac{|G|}{N} \sum_{k=1}^{N} f(x_{1k}, x_{2k}, \ldots, x_{nk}). \tag{9.57}$$

W celu zilustrowania metody obliczymy znaną nam już całkę:

$$2^{-3} \iiint_{x_1^2+x_2^2+x_3^2 \leqslant 1} dx_1\ dx_2\ dx_3 = \frac{\pi}{6} = 0,523599. \tag{9.58}$$

Dla wygody przekształcamy powyższą całkę do postaci:

$$\frac{1}{4} \iint_{x^2+y^2 \leqslant 1} \sqrt{1-(x^2+y^2)} dx\ dy = \frac{\pi}{4} \iint_{x^2+y^2 \leqslant 1} \sqrt{1-(x^2+y^2)} dP. \tag{9.59}$$

Zgodnie z naszymi oczekiwaniami, P jest rozkładem równomiernym na kole

$$x^2 + y^2 \leqslant 1$$

oraz

$$dP = \frac{dx\ dy}{\pi}.$$

Oszacowanie omawianej całki dane jest wzorem:

$$I = \frac{\pi}{4N} \sum_{k=1}^{N} \sqrt{1-(x_k^2+y_k^2)}. \tag{9.60}$$

Punkty (x_k, y_k) są punktami losowanymi według rozkładu równomiernego na kole $x^2 + y^2 \leqslant 1$. Procedura szacowania całki wielokrotnej metodą próbkowania średniego ma postać:

1. Uruchamiamy generator liczb pseudolosowych o rozkładzie równomiernym na przedziale $(0, 1)$.
2. Przy pomocy skalowania otrzymujemy liczby pseudolosowe o rozkładzie równomiernym na przedziale $(-1, 1)$. Oznaczamy te liczby jako RN_s. Stosujemy skalowanie: $RN_s = (max - min)RN + min$, gdzie RN jest liczbą z przedziału $(0, 1)$, min i max reprezentują maksymalną i minimalną wartość, w naszym przypadku mamy: $max = 1, min = -1$.
3. Według tego rozkładu losujemy N punktów o współrzędnych (x_k, y_k).
4. Dla każdego wylosowanego punktu sprawdzamy, czy spełniony jest warunek $x^2 + y^2 \leqslant 1$.
5. Jeżeli warunek ten nie jest spełniony, wylosowany punkt odrzucamy i powtarzamy losowanie aż do momentu spełnienia tego warunku.
6. Obliczamy średnią z N wartości: $1/N \sum_{k=1}^{n} \sqrt{1-(x_k^2+y_k^2)}$.
7. Uważamy, że średnia pomnożona przez czynnik $\pi/4$ jest oszacowaniem rozważanej całki.

Kod źródłowy programu przeznaczonego do szacowania całek wielokrotnych metodą próbkowania średniego pokazany jest na Listingu 9.9.

Listing 9.9: Całkowanie metodą Monte Carlo – potrójna metoda średniego próbkowania.

```
#include<iostream>
#include<cstdlib>
#include<ctime>
#include<cmath>

const double PI = 3.1415926;

double fx(double, double);

int main()
{
  std::srand(std::time(0));
  int n;
  std::cout << "poadj liczbe losowan N > 10, N = ";
  std::cin >> n;
  double norm = RAND_MAX + 1.0;
  double sumf1 = 0.0;
  int m = 0;
  do
    {
      double x1 = 2 * (std::rand() / norm) - 1;
      double x2 = 2 * (std::rand() / norm) - 1;
      if ((x1 * x1 + x2 * x2) <= 1.0)
        {
          sumf1 += fx(x1, x2);
          m++;
        }
    }
  while (m < n);

  double avg = 0.25 * PI * sumf1 / n;
  std::cout << "calka = "<< avg << std::endl;
  return 0;
}

double fx(double x1, double x2)
{
  return std::sqrt((1 - (x1 * x1 + x2 * x2)));
}
```

9.5 Zadania testowe

Testowanie generatorów liczb pseudolosowych może być wykonane za pomocą zadań kontrolnych. Metoda ta polega na rozwiązywaniu metodami Monte Carlo wybranych zadań za pomocą wygenerowanych liczb pseudolosowych badanego generatora i porównanie otrzymanych wyników z wynikami otrzymanymi innymi metodami. Klasyczne zadania kontrolne to szacowanie liczby π czy obliczanie całek. W niniejszym podrozdziale omówimy kilka zadań kontrolnych.

9.5.1 Gra Penney'a

W znakomitym podręczniku Grahama, Knutha i Patashnika omówiona jest gra wymyślona w 1969 roku przez W. Penney'a. Gra polega na tym, że dwóch graczy powiedzmy Ola i Olo rzucają monetą, tak długo, aż wypadnie ustalony na początku gry wzorzec. Przyjmijmy następujące wzorce:

- wzorzec Oli: OOR,
- wzorzec Ola: ORR.

gdzie O oznacza wyrzucenie orła, R oznacza wyrzucenie reszki. Ola wygrywa, gdy wzorzec Oli – OOR wypadnie jako pierwszy, natomiast Olo wygrywa gdy jako pierwszy wypadnie wzorzec ORR. Na pierwszy rzut oka, szanse wygranej powinny być jednakowe dla Oli i Ola. Autorzy cytowanego podręcznika dowodzą (korzystając z funkcji tworzących), że tak nie jest. Okazuje się, że w opisanym przypadku prawdopodobieństwo wygrania Oli jest znacznie większe. Ola będzie wygrywała dwa razy częściej niż Olo! Dokładne wyliczenia pokazują te prawdopodobieństwa:

$$P(Ola) = \frac{2}{3},$$
$$P(Olo) = \frac{1}{3}.$$

Aby sprawdzić ten nieoczekiwany wynik możemy wykonać symulacje komputerowe gry Penney'a i porównać otrzymane wyniki. Program symulacyjny pokazany jest na Listingu 9.10.

Listing 9.10: Symulacja gry Penneya.

```
1  #include<iostream>
2  #include<cstdlib>
3  #include<ctime>
4
5  const int N = 1;
6  const int POW = 900;
7  const int NL = 100;
8
9  int rand(int);
10
11 int main()
12 {
13   int a, b, c, g, i, olo, ola;
14   ola = olo = 0;
15   for(int j = 0; j < POW; j++)
16     {
17       a = rand(N);
18       b = rand(N);
19       c = rand(N);
20       for(int i = 0; i < NL; i++)
21         {
22           if ((a == 0) && (b == 0) && (c == 1))
23             {
24               ola++;
25               break;
26             }
27           if ((a == 0) && (b == 1) && (c == 1))
```

```
28              {
29                olo++;
30                break;
31              }
32            g = rand(N);
33            a = b;
34            b = c;
35            c = g;
36          }
37      }
38    std::cout << "Ola = " << ola << " Olo = " << olo << std::endl;
39    return 0;
40  }
41
42  int rand(int n)
43  {
44    static bool first = true;
45    if (first)
46      {
47        std::srand(std::time(NULL));
48        first = false;
49      }
50    return std::rand() / (RAND_MAX + 1.0) * (n + 1);
51  }
```

Podczas pierwszej symulacji Ola i Olo grają 900 razy. Takich symulacji wykonano 10. W Tabeli 9.6 pokazano przykładowe wyniki. W swoich rozważaniach o grze Pen-

Tabela 9.6: Wynik symulacji Gray Penneya.

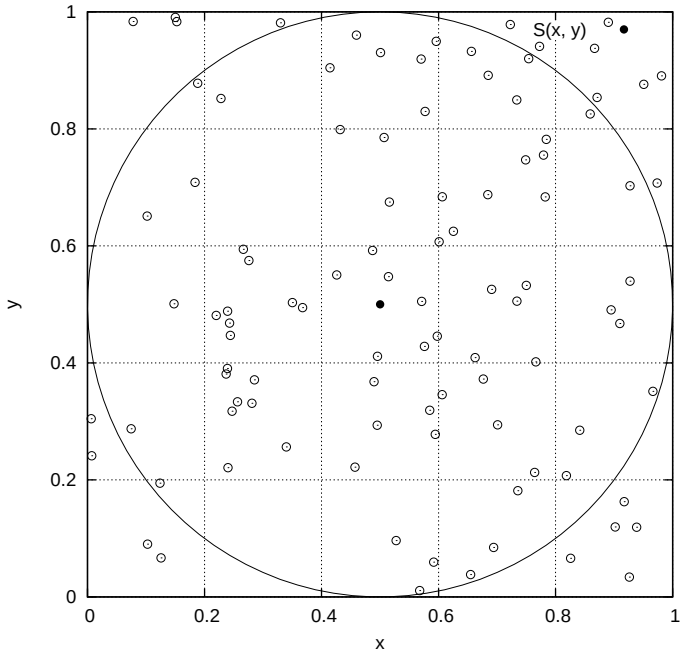
Nr symulacji	Liczba sukcesów Oli	Liczba sukcesów Ola
1	586	314
2	604	296
3	589	317
4	601	299
5	576	324
6	587	313
7	585	315
8	620	280
9	606	294
10	611	289

neya autorzy podręcznika piszą: *„Dziwne rzeczy mogą się zdarzyć w grze Penneya. Na przykład wzorzec OORO wygrywa ze wzorcem OROO w stosunku* 3/2 *i wzorzec OROO wygrywa ze wzorcem ROOO w* 7/5 *przypadkach. Tak, więc OORO powinno być dużo lepsze niż ROOO. Jednakże ROOO wygrywa z OORO w stosunku* 7/5*. Relacje wygrywania pomiędzy wzorcami nie są przechodnie”.*

9.5.2 Szacowanie liczby pi

Zastosujemy omawiane generatory liczb pseudolosowych do obliczenia przybliżonej wartości liczby π. Do oszacowania wykorzystamy metody rachunku prawdo-

podobieństwa i metody symulacji komputerowych. Niech w kwadrat jednostkowy wpisane będzie koło (Rysunek 9.3).



Rysunek 9.3: Metoda szacowania liczby π.

Przy pomocy generatora liczb pseudolosowych losujemy współrzędne punktów. Wylosowany punkt może leżeć wewnątrz koła lub nie. Należy obliczyć stosunek liczby punktów wylosowanych w kole (L) do liczby wszystkich losowań (N). Ten stosunek równy jest stosunkowi powierzchni kwadratu do powierzchni koła. Kod programu realizującego to zadanie zamieszczono na Listingu 9.11.

Listing 9.11: Oszacowanie liczby π.

```
#include<iostream>
#include<cstdlib>
#include<ctime>

double rg_1();

int main()
{
  int n;
  std::cout << "podaj liczbe losowan: ";
  std::cin >> n;
  int l = 0;
  for (int i = 1; i <= n; i++)
    {
      double x = rg_1();
      double y = rg_1();
      double r2 = (x - 0.5) * (x - 0.5) + (y - 0.5) * (y - 0.5);
      if (r2 <= 0.25)
```

```
19          l++;
20      }
21    double pi = 4.0 * l / n;
22    std::cout << "dla n = " << n << ", oszacowanie pi = " << pi << std::
          endl;
23    return 0;
24  }
25
26  double rg_1()
27  {
28    static bool flag = true;
29    if (flag)
30      {
31        std::srand(std::time(0));
32        flag = false;
33      }
34    return std::rand() / (double)(RAND_MAX + 1.0);
35  }
```

Funkcja `rg_1()` generuje liczby pseudolosowe z przedziału $[0, 1]$. Do tego celu wykorzystano generator `rand()`:

```
1   std::rand() / (double)(RAND_MAX + 1.0);
```

Aby zapewnić losowanie zmienne w czasie wykorzystano funkcję `srand()`:

```
1   std::srand(std::time(0));
```

Funkcja `srand()` powinna być wywołana tylko jeden raz, dlatego mamy warunek:

```
1   static bool flag = true;
2   if (flag)
3   {
4     std::srand(std::time(0));
5     flag = false;
6   }
```

Dla każdego punktu należy wylosować jego współrzędne (x, y):

```
1   x = rg_1();
2   y = rg_1();
```

Koło wpisane w kwadrat jednostkowy ma środek w punkcie S o współrzędnych $(0,5; 0,5)$ (Rysunek 9.3). Dla wylosowanego punktu należy obliczyć jego odległość od środka koła (dla przyspieszenia obliczeń obliczamy kwadrat odległości):

```
1   r2 = (x - 0.5) * (x - 0.5) + (y - 0.5) * (y - 0.5);
```

a następnie sprawdzić, czy punkt leży wewnątrz okręgu:

```
1   if (r2 <= 0.25)
2      l++;
```

Jeżeli kwadrat odległości wylosowanego punktu od środka koła jest mniejszy lub równy kwadratowi promienia r (w kole jednostkowym jest to 0,25), to zwiększamy licznik punktów leżących wewnątrz koła. Ostateczne oszacowanie liczby π:

```
1   pi = 4.0 * l / n;
```

jest wyświetlane na ekranie monitora. Ponieważ mamy do czynienia z symulacjami komputerowymi, za każdym razem dostaniemy inny wynik. Należy także spodziewać się, że dokładność oszacowania będzie większa dla większej liczby losowań. Przykładowe rezultaty wykonania programu pokazano w Tabeli 9.7.

Tabela 9.7: Przykładowe oszacowana liczby π.

Liczba losowań (N)	Oszacowanie π
100	2.84
500	3.184
1000	3.2
5000	3.1552
10000	3.1504
50000	3.14008

Rozdział 10

Metody geometrii obliczeniowej

10.1 Wstęp

Geometria obliczeniowa (ang. *computational geometry*) bada algorytmy przeznaczone do rozwiązywania geometrycznych problemów przy pomocy komputerów. Jest to stosunkowo nowy dział informatyki, liczący sobie około 35 lat, jeżeli za początek tych badań przyjmiemy opublikowanie w roku 1978 roku tekstu rozprawy doktorskiej poświęconej algorytmom geometrycznym przez M. I. Shamosa. Tematyka będąca w kręgu zainteresowań geometrii obliczeniowej jest bardzo rozbudowana. Zgodnie z monografią J. O'Rourke najpopularniejsze algorytmy to:

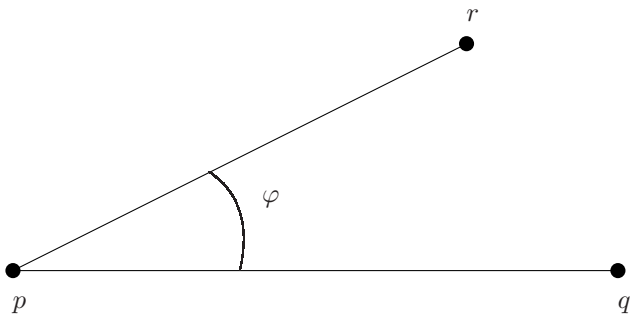
- Wyliczanie powierzchni wielokątów.
- Triangulacja wielokątów.
- Otoczka wypukła w dwóch wymiarach.
- Otoczka wypukła w trzech wymiarach.
- Triangulacja Delaunaya.
- Przecięcie odcinka z odcinkiem.
- Przecięcie odcinka z trójkątem.
- Punkt w wielokącie.
- Punkt w wielościanie.
- Przecięcie wypukłych wielokątów.
- Opis ruchu ramienia robota (kinematyka).

Elementarne wprowadzenie do geometrii obliczeniowej można znaleźć w monografii Banachowskiego, Diksa i Ryttera. Geometria obliczeniowa posługuje się takimi pojęciami jak punkt, prosta, płaszczyzna, dodatkowo tworzone są takie obiekty, jak na przykład odcinek, wektor, wielobok (wielokąt), powierzchnia, wielościan. Najczęściej ograniczamy się do rozważań geometrycznych dla obiektów zdefiniowanych w układzie kartezjańskim dwuwymiarowym i trójwymiarowym.

Punkt p na płaszczyźnie dwuwymiarowej reprezentuje dwójka liczb (współrzędne punktu), co zapisujemy jako $(x(p), y(p))$. Często w literaturze spotkamy oznaczenie $p(x, y)$, gdzie x jest odciętą a y rzędną punktu. Jest to podstawowy element geometryczny, wszystkie inne obiekty tworzone są przy pomocy punktów:

- Odcinek jest reprezentowany przez punkt początkowy i punkt końcowy odcinka.
- Prosta jest definiowana jako para różnych punktów należących do prostej.
- Okrąg i koło jest reprezentowane przez zbiór punktów spełniających określone kryteria, np. okrąg jest zbiorem punktów równo odległych od punktu zwanego środkiem okręgu.
- Wektor jest skierowanym odcinkiem, reprezentowanym przez dwójkę liczb.
- Wielokąt może być zdefiniowany jako wektor punktów będących wierzchołkami wielokąta w kolejności występowania na obwodzie.

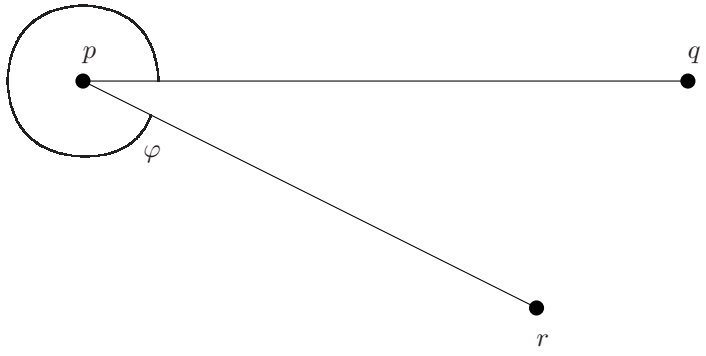
Ważne są relacje pomiędzy punktem i prostą. Jeżeli dwa punkty $p(x, y)$ i $q(z, t)$ wyznaczają odcinek, to możemy zadać pytanie po której stronie tego odcinka leży punkt $r(u, v)$. Na płaszczyźnie 2D (dwuwymiarowej) mamy trzy możliwości, pokazane na kolejnych rysunkach. Znalezienie położenia punktu względem prostej jest



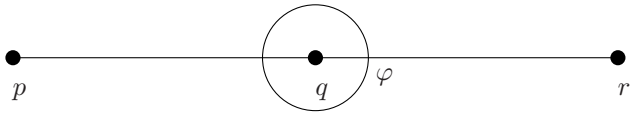
Rysunek 10.1: Punkt r leży nad prostą pq.

stosunkowo nieskomplikowane. Należy obliczyć wyznacznik $\det(p, q, r)$ postaci:

$$\det(p, q, r) = \begin{bmatrix} x & y & 1 \\ z & t & 1 \\ u & v & 1 \end{bmatrix} \tag{10.1}$$



Rysunek 10.2: Punkt r leży pod prostą pq.



Rysunek 10.3: Punkty p, q i r są współliniowe.

Wnioski są następujące:

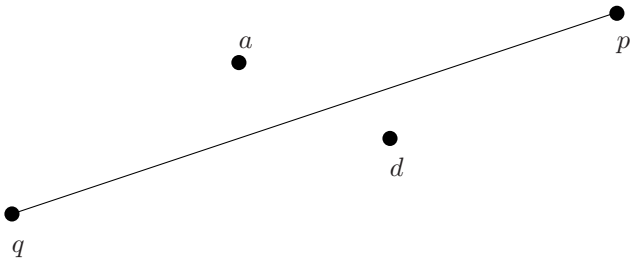
- Jeżeli $\det(p,q,r) > 0$, to punkt r leży po lewej stronie prostej pq ($\sin\varphi > 0$).
- Jeżeli $\det(p,q,r) < 0$, to punkt r leży po prawej stronie prostej pq ($\sin\varphi < 0$).
- Jeżeli $\det(p,q,r) = 0$, to punkty p, q i r są współliniowe ($\sin\varphi = 0$).

W grafice komputerowej często sprawdzamy, czy punkty leżą czy też nie leżą po jednej stronie prostej lub sprawdzamy czy punkt należy do prostej. Zanim omówimy rozwiązanie tych zagadnień przypomnimy pojęcie funkcji znaku (w monografii Banachowskiego, Diksa i Ryttera jest błąd drukarski). Jeżeli x jest liczbą rzeczywistą, to znak liczby x oznaczany symbolem $\mathrm{sgn}(x)$ i definiujemy następująco:

$$\mathrm{sgn}(x) = \begin{cases} 1 & : x > 0 \\ 0 & : x = 0 \\ -1 & : x < 0 \end{cases} \tag{10.2}$$

Na płaszczyźnie gdy jest zdefiniowana prosta, punkty mogą być różnie położone względem prostej – mogą być po obu stronach prostej, mogą leżeć tylko po jednej stronie, mogą też leżeć na prostej qp (Rysunek 10.4). Prostą pq wyznaczają dwa punkty $p(x,y)$ i $q(z,t)$. Chcemy ustalić czy punkty $a(b,c)$ i $d(e,f)$ leżą po jednej stronie czy po obu stronach prostej. Aby rozwiązać to zagadnienie należy sprawdzić warunek:

$$\mathrm{sgn}(\det(p,q,a)) = \mathrm{sgn}(\det(p,q,d)).$$



Rysunek 10.4: Położenie punktów względem prostej.

Punkty leżą po tej samej stronie prostej, gdy warunek jest spełniony. Punkt może leżeć na prostej (Rysunek 10.5). Punkt $r(u, v)$ leży na prostej pq, gdy spełnione są warunki:

$$x(p) \leqslant x(r) \leqslant x(q) \wedge \operatorname{sgn}(\det(p, q, r)) = 0 : x(p) \leqslant x(q),$$
$$y(p) \leqslant y(r) \leqslant y(q) \wedge \operatorname{sgn}(\det(p, q, r)) = 0 : y(p) \leqslant y(q).$$

Mając dane równanie prostej postaci:

$$ax + by + c = 0, \tag{10.3}$$

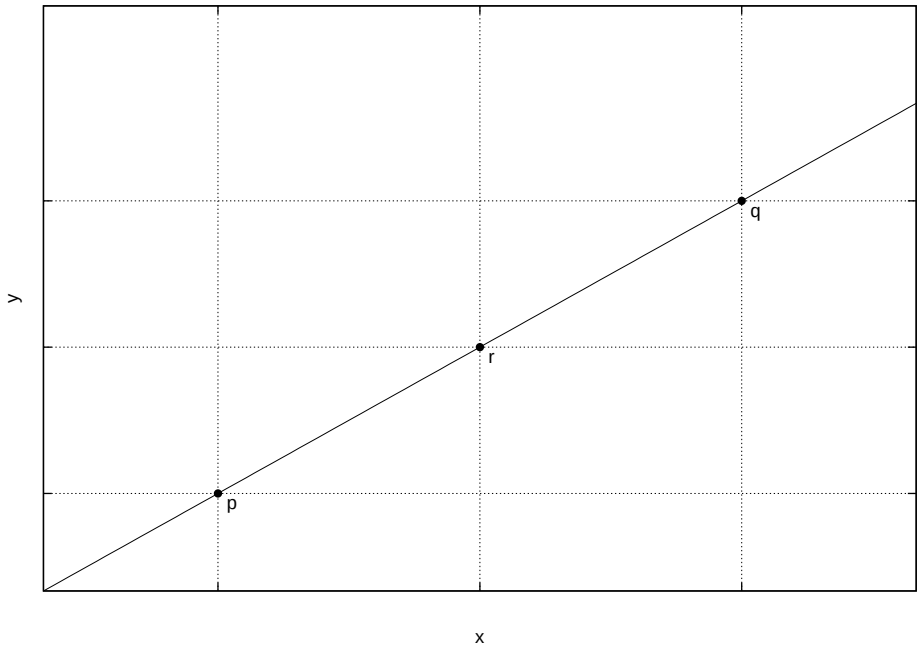
i współrzędne punktu $r(x_1, y_1)$ można obliczyć odległość d punktu od prostej zgodnie ze wzorem:

$$d = \frac{|ax_1 + by_1 + c|}{\sqrt{a^2 + b^2}}. \tag{10.4}$$

Kod źródłowy programu użytego do wyliczania odległości punktu od prostej pokazany jest na Listingu 10.1.

Listing 10.1: Obliczanie odległości punktu od prostej.

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

int main()
{
  double A, B, d;
  double x1, x2, y1, y2;
  double xp,yp;
  std::cout << "podaj 1 punkt prostej x1 = ";
  std::cin  >> x1;
  std::cout << "podaj 1 punkt prostej y1 = ";
  std::cin >> y1;
  std::cout << "podaj 2 punkt prostej x2 = ";
  std::cin >> x2;
  std::cout << "podaj 2 punkt prostej y2 = ";
  std::cin >> y2;
  A = y2 - y1;
```



Rysunek 10.5: Punkt r leży na prostej pq.

```
18   B = x2 - x1;
19
20   std::cout << "podaj badany punkt xp = ";
21   std::cin >> xp;
22   std::cout << "podaj badany punkt yp = ";
23   std::cin >> yp;
24
25   d = std::fabs(A * (x1 - xp) + B * (yp - y1)) / std::sqrt(A * A + B *
         B);
26   std::cout << "odleglosc = "<< d << std::endl;
27   return 0;
28 }
```

Dla prostej o równaniu:

$$y = x - 1, \tag{10.5}$$

odległość punktu $p(13, 13)$ od tej prostej jest równa $0,707107$. Prostą o równaniu $y = x - 1$ wyznaczają dwa punkty, na przykład $p1(-3, -4)$ oraz $p2(7, 6)$.

Wyliczanie pól figur geometrycznych sprawia kłopoty, ze względu na konieczność pamiętania dość zawiłych wzorów. Jeżeli wielokąt wypukły opisany jest na płaszczyźnie zbiorem wierzchołków, to obliczenie jego powierzchni jest zadaniem trywialnym.

Jeżeli wielokąt jest wyznaczony przez zbiór n punktów postaci:

$$\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\},$$

to wzór na pole powierzchni wielokąta ma postać:

$$\frac{1}{2}(x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + \dots\ x_{n-1}y_n - x_ny_{n-1}). \tag{10.6}$$

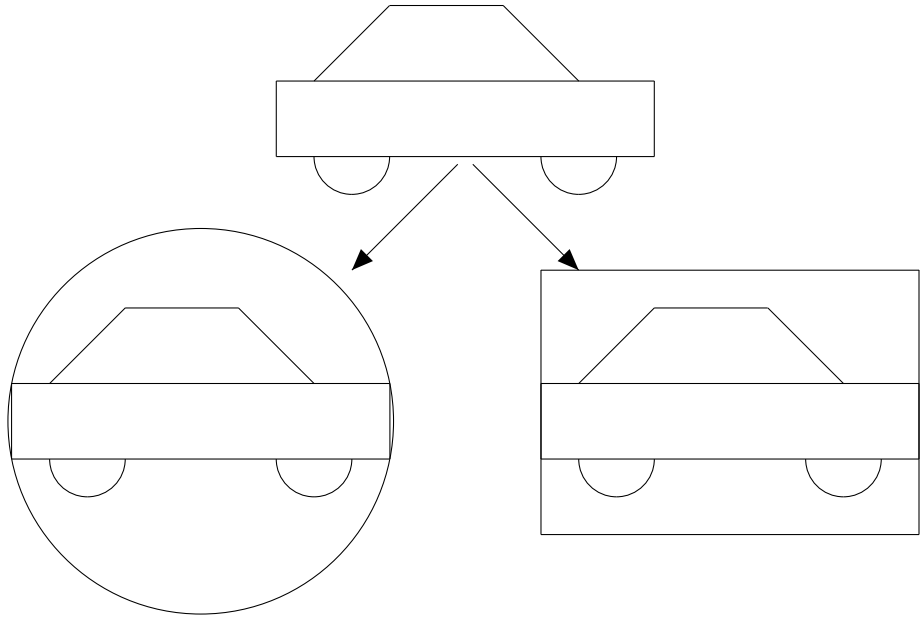
Kod źródłowy programu użytego do wyliczania pola powierzchni wielokąta pokazany jest na Listingu 10.2. Znak wartości pola powierzchni zależy od porządku wprowadzania wierzchołków. W naszym algorytmie położenia wierzchołków wprowadzamy kolejno zgodnie z ruchem wskazówek zegara.

Listing 10.2: Obliczanie pola wielokąta.

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

const int W = 20;

int main()
{
  double x[W],y[W];
  int n,i;
  double area = 0.0;
  std::cout << "podaj liczbe wierzcholkow n = ";
  std::cin >> n;

  std::cout << "podaj x1 = ";
  std::cin >> x[0];
  std::cout << "podaj y1 = ";
  std::cin >> y[0];

  for (i=1;i<n;i++)
    {
      std::cout << "podaj x" << i + 1 << " = ";
      std::cin >> x[i];
      std::cout << "podaj y" << i + 1 << " = ";
      std::cin >> y[i];
      area += x[i - 1] * y[i] - x[i] * y[i - 1];
    }
  std::cout << "pole = "<< 0.5 * std::fabs(area) << std::endl;
  return 0;
}
```

Dla wielokąta wyznaczonego przez zbiór pięciu punktów $\{(0,0),(0,1),(1,2),(1,1),(2,0)\}$ zgodnie z oczekiwaniem otrzymamy wartość pola równą $2,0$.

W grach komputerowych ważnym zagadnieniem jest wykrywanie kolizji. Wykrywanie kolizji w grach komputerowych nie jest trywialnym zagadnieniem. Dysponujemy wieloma technikami wykrywania kolizji. Jednym z najprostszych sposobów wykrywania kolizji jest zastosowanie techniki brył otaczających. W tym celu w przestrzeni 3D najczęściej wykorzystujemy sześcian lub sferę, a w przestrzeni 2D wykorzystujemy kwadrat lub okrąg (Rysunek 10.6). Promień okręgu dobieramy w ten sposób, aby wszystkie elementy obiektu były w nim zamknięte. Technicznie oznacza to, że przeglądamy wszystkie wierzchołki obiektu i sprawdzamy czy są wewnątrz okręgu. Podobnie postępujemy znajdując kwadrat otaczający obiekt. Gdy dla obiektów zostaną znalezione okręgi otaczające (położenie środka okręgu i jego promień), wykrycie kolizji jest już zadaniem prostym. Należy ustalić,

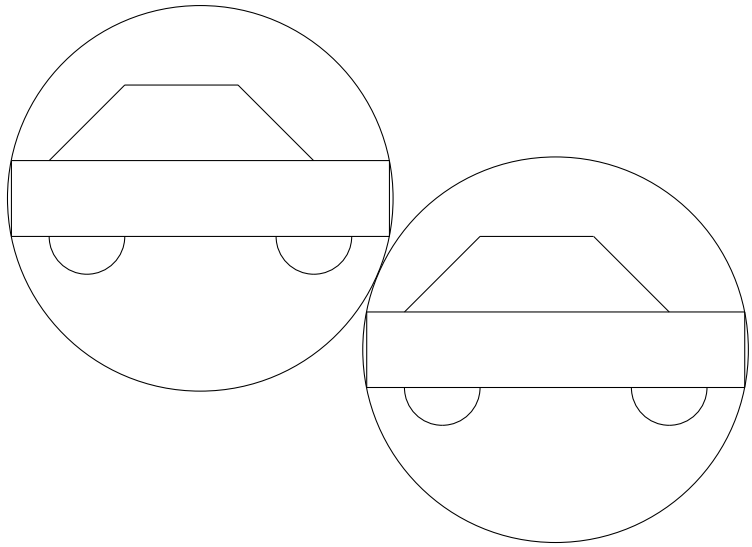


Rysunek 10.6: Bryły otaczające – okrąg i kwadrat.

czy spełniony jest warunek, że odległość pomiędzy środkami dwóch okręgów jest mniejsza niż suma ich promieni. Na Rysunku 10.7 pokazana jest sytuacja, gdy znaleziona jest kolizja. W wielu przypadkach zastosowanie prostokątów otaczających lub prostopadłościanów może dać dokładniejsze wyniki przy wykrywaniu kolizji, gdyż bryła otaczająca lepiej przybliży kształty obiektu. Taka sytuacja pokazana jest na Rysunku 10.8. Zastosowanie okręgu ograniczającego obiekt byłoby zbyt dużym przybliżeniem. W celu utworzenia prostokąta ograniczającego należy przejrzeć listę wierzchołków obiektu i wyszukać wierzchołki o największej i najmniejszej współrzędnej x i y. Funkcja wyznaczająca prostokąt ograniczający może mieć postaci pokazanej na Listingu 10.3. Prostokąt otaczający zdefiniowany jest dwoma wierzchołkami $LD(x, y)$ oraz $PG(x, y)$, gdzie LD jest to dolny prawy wierzchołek prostokąta, a PG jest to prawy górny wierzchołek prostokąta.

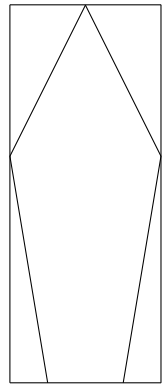
Listing 10.3: Prostokąt otaczający – testowanie kolizji.

```
1  #include <iostream>
2
3  struct Point {int x, y; };
4
5  const int NP = 5;
6
7  int main()
8  {
9    Point ob[NP] = {{5,2}, {4,12}, {7,18}, {10,12}, {9,2}};
10   Point p = {12,5};    // testowany   punkt
11   int minx=ob[0].x;
12   int miny=ob[0].y;
13   int maxx=ob[0].x;
14   int maxy=ob[0].y;
```



Rysunek 10.7: Wykrywanie zderzenia przy pomocy okręgów otaczających.

```
15
16  for (int i = 1; i < NP; i++)
17    {
18      if (ob[i].x < minx) minx = ob[i].x;
19      if (ob[i].y < miny) miny = ob[i].y;
20      if (ob[i].x > maxx) maxx = ob[i].x;
21      if (ob[i].y > maxy) maxy = ob[i].y;
22    }
23    //   wykrywanie   kolizji :
24  if ((p.x >= minx && p.y >= miny) && (p.x <= maxx && p.y <= maxy))
25    std::cout << "Kolizja\n";
26  else
27    std::cout << "Brak kolizji\n";
28  return 0;
```



Rysunek 10.8: Prostokąt ograniczający obiektu.

```
29 }
```

W programie przy pomocy pętli `for` przeglądamy wierzchołki obiektu `ob` i wyznaczamy współrzędne wierzchołków prostokąta otaczającego ((`minx`, `miny`) oraz (`maxx`, `maxy`)). W naszym przykładzie otrzymamy wartości $LD(4,2)$ i $PG(10,18)$. Sprawdzamy czy punkt `p` o współrzędnych $(12,5)$ znajduje się wewnątrz prostokąta otaczającego obiekt `ob`. W tym przypadku nie jest to prawda i otrzymujemy komunikat „Brak kolizji". Gdyby punkt `p` miał współrzędne $(5,5)$ otrzymalibyśmy komunikat „Kolizja".

10.2 Przynależność punktu do figury

W geometrii obliczeniowej rozważa się często zagadnienie przynależności jednych obiektów do innych, na przykład pytamy się czy dany punkt na płaszczyźnie znajduje się wewnątrz zdefiniowanego okręgu czy też leży poza nim.

Najprostsze do zbadania jest zagadnienie przynależności punktu do prostokąta. Niech prostokąt będzie zdefiniowany dwoma punktami $A(x_1, y_1)$ i $B(x_2, y_2)$. Punkt A oznacza dolny lewy wierzchołek prostokąta, punkt B oznacza górny prawy wierzchołek prostokąta. Sprawdzimy, czy punkt $P(x, y)$ leży wewnątrz prostokąta. W tym celu należy sprawdzić, czy odcięta punktu leży pomiędzy odciętymi wierzchołków prostokąta i czy rzędna punktu leży pomiędzy rzędnymi wierzchołków prostokąta. Muszą być spełnione nierówności:

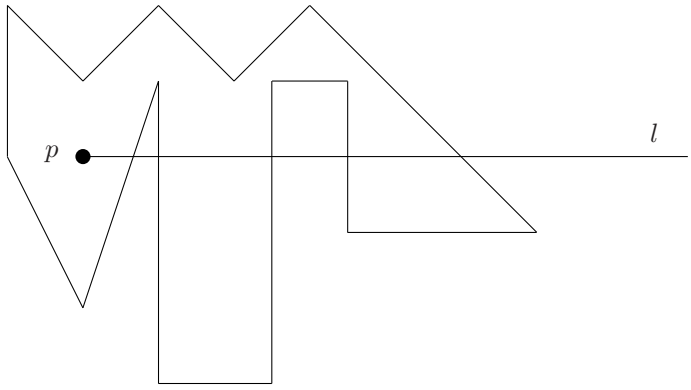
$$\min(x_1, x_2) \leqslant x \leqslant \max(x_1, x_2),$$
$$\min(y_1, y_2) \leqslant y \leqslant \max(y_1, y_2).$$

Problemy lokalizacji punktu w prostokącie czy okręgu są dość proste. Bardziej ogólny problem polega na rozwiązaniu następującego zadania. *Jeśli jest dany wielokąt prosty W i punkt P, sprawdzić czy P należy do wnętrza W.* Istnieje zaskakujące proste rozwiązanie tego pytania. Niech l będzie półprostą o początku w P. Punkt P leży wewnątrz wielokąta W wtedy i tylko wtedy, gdy l przecina brzeg W nieparzystą ilość razy (Rysunek 10.9).

10.3 Test przecinania się odcinków

Do zagadnienia znalezienia punktu przecięcia się dwóch odcinków leżących na płaszczyźnie można podejść wprost, rozwiązując układ równań liniowych. Formalnie dysponujemy dwoma równaniami (równania prostych) oraz mamy dwie niewiadome (współrzędne (x, y) punktu przecięcia). Nasze równania to:

$$a_1 x + b_1 y = d_1$$
$$a_2 x + b_2 y = d_2$$



Rysunek 10.9: Badanie zawierania się punktu w wielokącie prostym. Półprosta l przecina się z wielokątem na lewo od P jeden raz, na prawo od P – 5 razy, wobec czego P należy do wnętrza wielokąta.

Rozwiązanie tego układu równań jest proste:

$$x = \frac{b_2 d_1 - b_1 d_2}{a_1 b_2 - a_2 b_1}$$
$$y = \frac{a_1 d_2 - a_2 d_1}{a_1 b_2 - a_2 b_1}$$

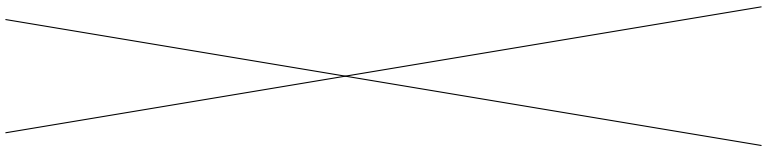
Należy pamiętać, że mamy trzy możliwości (Rysunek 10.10):

- Istnieje jedno rozwiązanie, mianownik we wzorach jest niezerowy.
- Nie ma rozwiązania, proste są równoległe, nie przecinają się, mianownik jest równy zero.
- Istnieje nieskończona liczba rozwiązań, proste pokrywają się, mianownik jest równy zero.

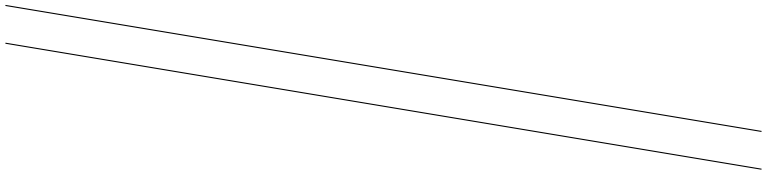
K. Loudon w swojej monografii omawia wydajny algorytm sprawdzania czy dwa odcinki się przecinają. Na Listingu 10.4 pokazany jest program sprawdzający czy odcinki się przecinają, metodę podał Loudon.

Listing 10.4: Test przecinania się odcinków – algorytm Loudona.

```
1  #include <iostream>
2
3  typedef struct
4  {
5    double x, y;
6  } Point;
7
8  inline double min(double x, double y) {return x < y ? x : y;}
9  inline double max(double x, double y) {return x > y ? x : y;}
10
11 bool loudon(Point p1, Point p2, Point p3, Point p4);
12
13 int main()
```



Linie przeciniające się: jedno rozwiązanie.



Linie równoległe: brak rozwiązania.

Linie pokrywają się: nieskończenie wiele rozwiązań.

Rysunek 10.10: Przecinanie się prostych na płaszczyźnie, trzy typy rozwiązań.

```
14 {
15   Point ob1[2] = {{1, 1}, {7, 6}};
16   Point ob2[2] = {{2, 8}, {6, 1}};
17   std::cout << loudon(ob1[0], ob1[1], ob2[0], ob2[1]) << std::endl;
18   return 0;
19 }
20
21 bool loudon(Point p1, Point p2, Point p3, Point p4)
22 {
23   double z1, z2, z3, z4;
24   int t1, t2, t3, t4 ;
25
26   if (! (max(p1.x, p2.x) >= min(p3.x, p4.x) &&
27          max(p3.x, p4.x) >= min(p1.x, p2.x) &&
28          max(p1.y, p2.y) >= min(p3.y, p4.y) &&
29          max(p3.y, p4.y) >= min(p1.y, p2.y)))
30     return false;
31
32   if ((z1 = ((p3.x - p1.x) * (p2.y - p1.y)) - ((p3.y - p1.y) * (p2.x -
         p1.x))) < 0)
33     t1 = -1;
34   else if (z1 > 0)
35     t1 = 1;
36   else
37     t1 = 0;
38
39   if ((z2 = ((p4.x - p1.x) * (p2.y - p1.y)) - ((p4.y - p1.y) * (p2.x -
         p1.x))) < 0)
40     t2 = -1;
41   else if (z2 > 0)
42     t2 = 1;
```

```
43    else
44      t2 = 0;
45
46    if ((z3 = ((p1.x - p3.x) * (p4.y - p3.y)) - ((p1.y - p3.y) * (p4.x -
          p3.x))) < 0)
47      t3 = -1;
48    else if (z3 > 0)
49      t3 = 1;
50    else
51      t3 = 0;
52
53    if ((z4 = ((p2.x - p3.x) * (p4.y - p3.y)) - ((p2.y - p3.y) * (p4.x -
          p3.x))) < 0)
54      t4 = -1;
55    else if (z4 > 0)
56      t4 = 1;
57    else
58      t4 = 0;
59
60    if ((t1 * t2 <= 0) && (t3 * t4 <= 0))
61      return true;
62    return false;
63  }
```

Algorytm jest zoptymalizowany pod kątem szybkości wykonania i dokładności (są oczywiście inne, dużo prostsze algorytmy, nie są one jednak tak wydajne).

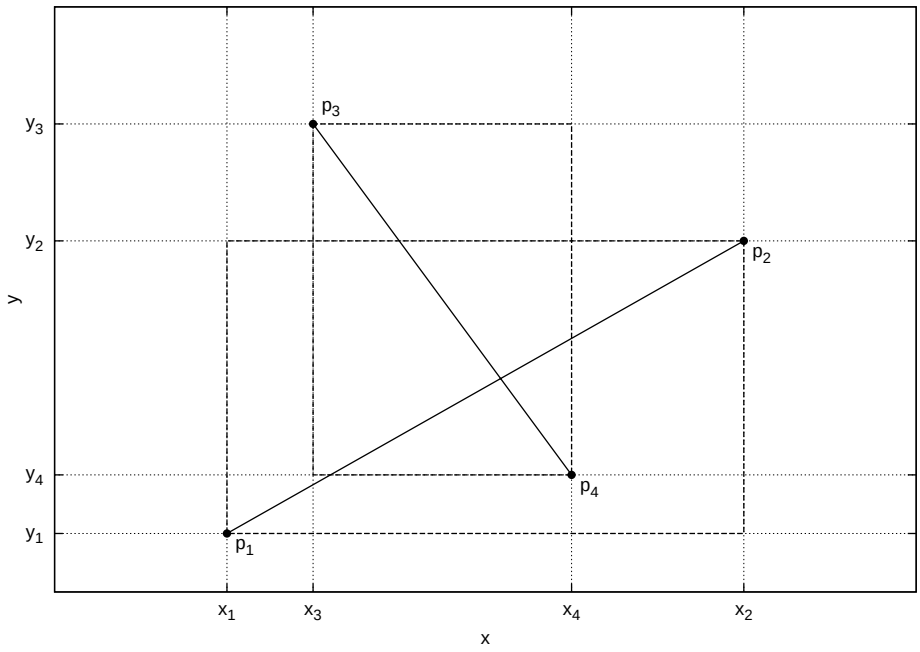
Odcinki są definiowane przy pomocy punktów początkowych i końcowych. Odcinek pierwszy definiowany jest parą punktów $p_1(x_1, y_1)$ oraz $p_2(x_2, y_2)$ odcinek drugi definiowany jest parą punktów $p_3(x_3, y_3)$ i $p_4(x_4, y_4)$. Rozpatrujemy tutaj zadanie ustalenia przecięcia dwóch odcinków na płaszczyźnie. W teście rozważamy dwa prostokąty otaczające (Rysunek 10.11). Algorytm jest dwuczęściowy – najpierw wykonywany jest szybki test odrzucenia, a potem wykonywany jest właściwy test. Krótki test odrzucenia ma postać:

```
1   if (! (max(p1.x, p2.x) >= min(p3.x, p4.x) &&
2          max(p3.x, p4.x) >= min(p1.x, p2.x) &&
3          max(p1.y, p2.y) >= min(p3.y, p4.y) &&
4          max(p3.y, p4.y) >= min(p1.y, p2.y)))
5     return false;
```

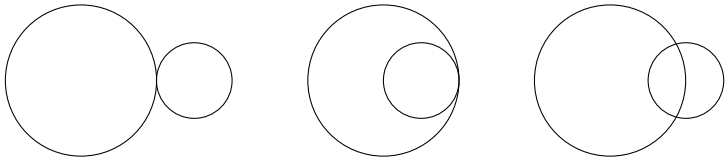
Według K. Loudona złożoność rozważanego algorytmu wynosi $O(1)$, gdyż wszystkie kroki wykonywane są w stałym czasie. Funkcja `loudon` zwraca wartość **`true`**, gdy odcinki się przecinają, w przeciwnym przypadku zwraca wartość **`false`**. W pokazanym przykładzie odcinki przecinają się.

10.4 Test przecinania się okręgów

Zagadnienie wyznaczania punktów przecięcia się dwóch okręgów ma duże znacznie praktyczne w grafice komputerowej. W grach komputerowych wykrycie kolizji ma istotne znaczenie. Jeżeli jako prymityw otaczający rozważany obiekt wybierzemy okrąg, to kolizje między dwoma obiektami wyznaczymy rozwiązując zagadnienie



Rysunek 10.11: Algorytm Loudona do sprawdzenia czy odcinki się przecinają.



Rysunek 10.12: Przecinanie się dwóch okręgów.

przecinania się okręgów (patrz Rysunek 10.12). Okrąg pierwszy jest określony przez podanie położenia środka okręgu $p_1(p_{1x}, p_{1y})$ oraz jego promienia r_1, okrąg drugi jest określony przez podanie położenia środka okręgu $p_2(p_{2x}, p_{2y})$ oraz jego promienia r_2. Najprostsza metoda wyznaczenia punktów przecięcia się dwóch okręgów polega na rozwiązaniu układu równań:

$$\begin{cases} (x - p_{1x})^2 + (y - p_{1y})^2 & = r_1^2, \\ (x - p_{2x})^2 + (y - p_{2y})^2 & = r_2^2. \end{cases} \tag{10.7}$$

Kod źródłowy (opracowany korzystając z monografii P. Stańczyka „Algorytmy praktyczne") pokazany jest na Listingu 10.5.

Listing 10.5: Test przecinania się okregów.

```
1 #include<iostream>
2 #include<cmath>
3
```

```
4  const double EPS = 10e-9;
5
6  inline bool isZero(double x) { return x >= -EPS && x <= EPS; }
7
8  int main()
9  {
10   double ppx1,ppy1,ppx2, ppy2;
11     // okrag   1
12   double p1x = 4.0;
13   double p1y = 5.0;
14   double r1 = 3.0;
15     // okrag   2
16   double p2x = 2.0;
17   double p2y = 0.0;
18   double r2 = 4.0;
19
20   p2x -= p1x;
21   p2y -= p1y;
22     // wspolsrodkowe
23   if (isZero(p2x) && isZero(p2y))
24     {
25       std::cout << "okregi wspolsrodkowe" << std::endl;
26       return 0;
27     }
28
29   double A = (-r2 * r2 + r1 * r1 + p2x * p2x + p2y * p2y) * 0.5;
30     // ta   sama   wspolrzedna   y
31   if (isZero(p2y))
32     {
33       double x = A / p2x;
34       double y2 = r1 * r1 - x * x;
35       if (y2 < -EPS)
36         {
37           std::cout << "taka sama y" << std::endl;
38           return 0;
39         }
40         // okregi   styczne
41       if (isZero(y2))
42         {
43           ppx1 = p1x + x;
44           ppy1 = p1y;
45           std::cout << "x = " << ppx1 << " y = " << ppy1 << std::endl;
46           return 0;
47         }
48         // przecinaja   sie
49       else
50         {
51           ppx1 = p1x + x;
52           ppy1 = p1y + std::sqrt(y2);
53           ppx2 = p1x + x;
54           ppy2 = p1y - std::sqrt(y2);
55           std::cout << " x1 = " << ppx1 << " y1 = " << ppy1 << std::
                endl;
56           std::cout << " x2 = " << ppx2 << " y2 = " << ppy2 << std::
                endl;
57           return 0;
58         }
59     }
60   double a = p2x * p2x + p2y * p2y;
```

```
61    double b = -2.0 * A * p2x;
62    double c = A * A - r1 * r1 * p2y * p2y;
63    double d = b * b - 4.0 * a * c;
64    if (d < -EPS)
65      {
66        std::cout << "nie ma przeciecia" << std::endl;
67        return 0;
68      }
69    double x = -b / (2.0 * a);
70      // jesli   sa   styczne
71    if (isZero(d))
72      {
73        ppx1 = p1x + x;
74        ppy1 = p1y + (A - p2x * x) / p2y;
75        std::cout << "x1 = " << ppx1 << " y1 = " << ppy1 <<std::endl;
76        std::cout << "x2 = " << ppx2 << " y2 = " << ppy2 <<std::endl;
77      }
78      // okregi   przecijnaja   sie
79    else
80      {
81        double e = std::sqrt(d) / (2.0 * a);
82        ppx1 = p1x + x + e;
83        ppy1 = p1y + (A - p2x * (x + e)) / p2y;
84        ppx2 = p1x + x - e;
85        ppy2 = p1y + (A - p2x * (x - e)) / p2y;
86        std::cout << "x1 = " << ppx1 << " y1 = " << ppy1 << std::endl;
87        std::cout << "x2 = " << ppx2 << " y2 = " << ppy2 << std::endl;
88      }
89    return 0;
90  }
```

Po uruchomieniu tego programu mamy następujący wynik:

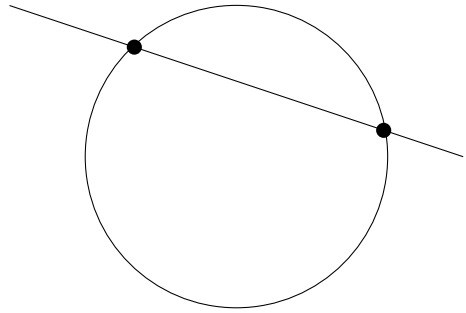
```
1   x1 = 5.28141 y1 = 2.28744
2   x2 = 1.20135 y2 = 3.91946
```

10.5 Test przecinania się odcinka i okręgu

Poszukiwanie punktów przecięcia prostej z okręgiem ma duże znaczenie praktyczne. Bardzo często w grach komputerowych poruszający się obiekt może zderzyć się z przeszkodą – należy obliczyć taki punkt. Proste trajektorie poruszających się obiektów reprezentują odcinki, przeszkoda modelowana jest okręgiem otaczającym. W celu obliczenia punktu przecięcia okręgu z prostą (patrz Rysunek 10.13) należy rozwiązać odpowiedni układ równań. Możemy mieć trzy możliwości:

- Prosta przecina okrąg, są dwa punkty przecięcia.
- Prosta jest styczna do okręgu, jest jeden punkt.
- Prosta nie przecina okręgu.

Okrąg jest określony przez podanie położenia środka okręgu $p_r(r_x, r_y)$ oraz jego promienia r. Prosta jest określona przez podanie punktu początkowego $p_1(p_{1x}, p_{1y})$



Rysunek 10.13: Przecięcie okręgu z prostą.

oraz punktu końcowego $p_2(p_{2x}, p_{2y})$. Układ równań ma postać:

$$\begin{cases} x & = b\, p_{1x} + (1-b)p_{2x}, \\ y & = b\, p_{1y} + (1-b)p_{2y}, \\ r^2 & = (x-r_x)^2 - (y-r_y)^2. \end{cases} \tag{10.8}$$

Kod źródłowy programu (opracowany korzystając z monografii P. Stańczyka „Algorytmy praktyczne") do wyznaczania punktów przecięcia prostej z okręgiem pokazany jest na Listingu 10.6.

Listing 10.6: Test przecinania się prostej i okręgu.

```
1  #include<iostream>
2  #include<cmath>
3
4  const double EPS = 10e-9;
5
6  inline bool isZero(double x) { return x >= -EPS && x <= EPS; }
7
8  int main()
9  {
10   double ppx, ppy, ppx1, ppy1, ppx2, ppy2;
11     // okrag
12   double rx = 0.0;
13   double ry = 0.0;
14   double r = 2.0;
15     // p1
16   double p1x = -3.0;
17   double p1y = 1.0;
18     // p2
19   double p2x = 4.0;
20   double p2y = 1.0;
21
22   double a = p1x * p1x + p1y * p1y + p2x * p2x + p2y * p2y - 2.0 * (
         p1x *p2x + p1y * p2y);
23   double b = 2.0 * (rx * (p2x - p1x) + ry * (p2y - p1y) + p1x * p2x +
         p1y * p2y - p2x * p2x - p2y * p2y);
24   double c = -(r * r) + p2x * p2x + p2y * p2y + rx * rx + ry * ry -
         2.0 * (rx * p2x + ry * p2y);
25   double d = b * b - 4.0 * a * c;
26   if (d < -EPS)
```

```
27      std::cout << "brak rozwiazania" << std::endl;
28    double t = -b / (2.0 * a);
29    double e = std::sqrt(std::fabs(d)) / (2.0 * a);
30    if (isZero(d))  // pojedyncze  rozwiazania
31      {
32        ppx = t * p1x + (1.0 - t) * p2x;
33        ppy = t * p1y + (1.0 - t) * p2y;
34        std::cout << "x = " << ppx << " y = " << ppy << std::endl;
35      }
36    else  // podwojne  rozwiazanie
37      {
38        ppx1 = (t + e) * p1x + (1.0 - t - e) * p2x;
39        ppy1 = (t + e) * p1y + (1.0 - t - e) * p2y;
40        std::cout << "x1 = " << ppx1 << " y1 = " << ppy1 << std::endl;
41        ppx2 = (t - e) * p1x + (1.0 - t + e) * p2x;
42        ppy2 = (t - e) * p1y + (1.0 - t + e) * p2y;
43        std::cout << "x2 = " << ppx2 << " y2 = " << ppy2 << std::endl;
44      }
45    return 0;
46  }
```

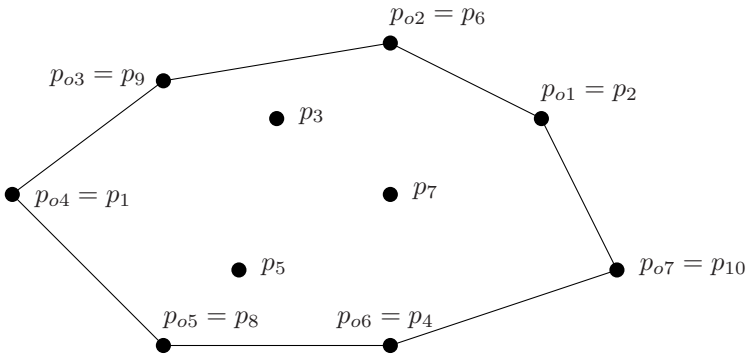
Po uruchomieniu tego programu mamy następujący wynik:

```
1   x1 = -1.73205 y1 = 1
2   x2 = 1.73205 y2 = 1
```

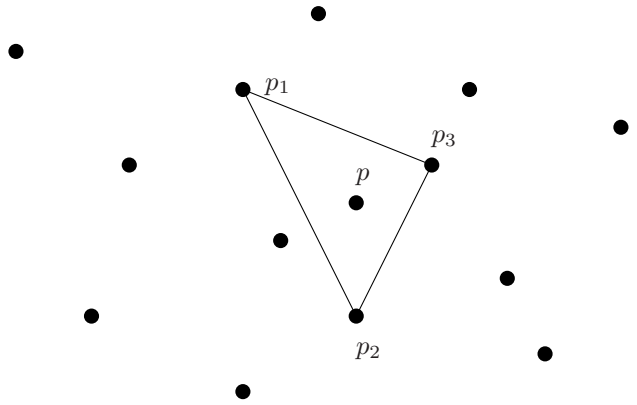
10.6 Obrys wypukły

Najbardziej rozważaną strukturą w geometrii obliczeniowej jest obrys wypukły (często też nazywany otoczką wypukłą, ang. *convex hull*). Definicja wypukłości i obrysu wypukłego w monografii Josepha O'Rourke zapisana jest na dwóch stronach i zawiera jedenaście punktów, co już wskazuje na to, że jest to struktura niebanalna.

Nie wdając się w szczegóły, podamy definicję zgodną z monografią Banachowskiego: *Obrysem wypukłym dowolnego zbioru punktów S nazywamy najmniejszy zbiór wypukły zawierający S.* Problem jest ogólny, w niniejszym skrypcie omówimy zagadnienie znajdowania obrysu wypukłego dla punktów umieszczonych na płaszczyźnie (skończony zbiór punktów w dwóch wymiarach). Zadanie formułowane jest następująco: dla zbioru n punktów leżących na płaszczyźnie należy znaleźć najmniejszy wielokąt wypukły zawierający wszystkie punkty (Rysunek 10.14). Praktycznie musimy wyznaczyć wierzchołki tego wielokąta. Znalezione wierzchołki sortuje się ze względu na ich kolejność występowania na obwodzie wielokąta. Znalezione wierzchołki wielokąta są tzw. punktami ekstremalnymi (ang. *extreme points*). Istnieje prosty sposób ustalenia czy punkt zbioru S jest nie-ekstremalny (ang. *nonextreme point*): *Punkt jest nie-ekstremalny jeżeli leży wewnątrz trójkąta, którego wierzchołki są punktami ze zbioru S, i sam nie jest jednym z wierzchołków tego trójkąta.* Na Rysunku 10.15 pokazano sposób ustalania czy punkt zbioru S jest ekstremalny. Znając metodę wyznaczania punktów ekstremalnych można podać postać naiwnego algorytmu obliczania obrysu wypukłego. Składa się on z



Rysunek 10.14: Obrys wypukły punktów na płaszczyźnie (zbiór S składa się z 10 punktów, 3 punkty nie są ekstremalne), wierzchołki wielokąta są uporządkowane w kierunku przeciwnym do kierunku ruchu wskazówki zegara.



Rysunek 10.15: Wyznaczanie punktów ekstremalnych. Punkt p nie jest punktem ekstremalnym, ponieważ leży wewnątrz trójkąta (p_1, p_2, p_3).

dwóch kroków:

1. Znaleźć wszystkie wierzchołki obrysu wypukłego dla zbioru S.
2. Uporządkować wierzchołki w kolejności ich występowania na obwodzie wielokąta.

Złożoność obliczeniowa dla n punktów zgodnie z monografią O'Rourke jest następująca:

1. Dla n punktów w celu ustalenia punktów ekstremalnych trzeba sprawdzić, co najwyżej $\binom{n}{3}$ trójkątów. Koszt obliczeń jest rzędu $O(n^4)$.
2. Optymalny czas sortowania jest rzędu $O(n \log n)$

Z tych oszacowań wynika, że całkowity koszt obliczeń jest rzędu $O(n^4)$. Naiwny algorytm znajdowania obrysu wypukłego nie jest oszczędny.

W literaturze przedmiotu dużą popularnością cieszą się dwa algorytmy znajdowania obrysu wypukłego:

- Algorytm Grahama, zwany czasem skanem Grahama.
- Algorytm Jarvisa, zwany czasem pochodem Jarvisa.

Algorytm Grahama ma złożoność obliczeniową rzędu $O(n \log n)$ i nie zależy od liczby punktów obrysu wypukłego. Algorytm Jarvisa ma złożoność obliczeniową rzędu $O(kn)$, gdzie n jest liczbą punktów w zbiorze S, a k jest liczbą wierzchołków w wielokącie wypukłym.

10.7 Długość łuku na kuli

Ważnym zagadnieniem jest wyznaczanie odległości pomiędzy dwoma punktami w przestrzeni. Lokalizacja punktów w przestrzeni może być realizowana różnymi sposobami. Zwykle punkty umieszczone w przestrzeni lokalizowane są przy pomocy współrzędnych, określonych dla różnych typów układów współrzędnych. Istnieje wiele typów takich układów współrzędnych, najpopularniejsze układy współrzędnych to:

- Prostokątny układ współrzędnych kartezjańskich.
- Biegunowy układ współrzędnych.
- Cylindryczny układ współrzędnych.
- Sferyczny układ współrzędnych.

Wyznaczanie odległości pomiędzy dwoma punktami w prostokątnym układzie kartezjańskim nie sprawia żadnego kłopotu. Jeżeli punkty p_1 i p_2 mają współrzędne $p_1(x_1, y_1, z_1)$ i $p_2(x_2, y_2, z_2)$, to odległość d między tymi punktami wyraża się wzorem:

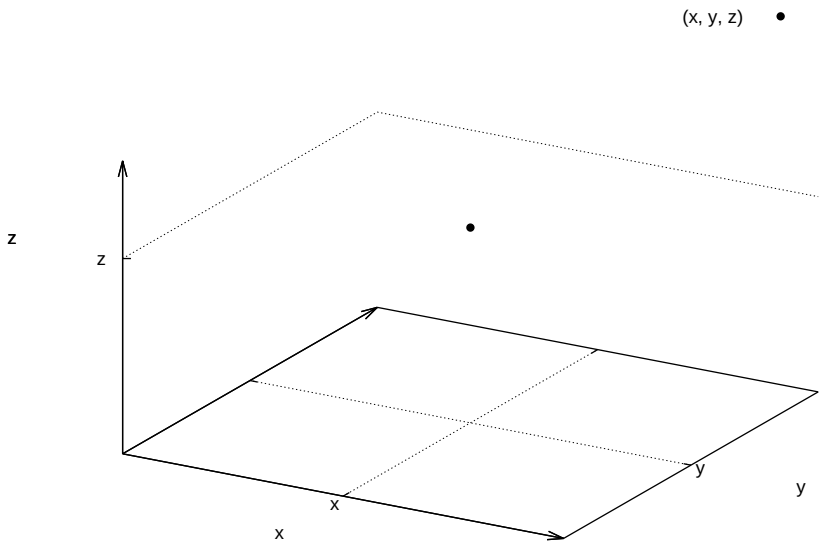
$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}. \tag{10.9}$$

Przy rozpatrywaniu pewnych zadań geometrycznych na płaszczyźnie euklidesowej wygodnie jest czasem wykorzystać inny niż prostokątny układ współrzędnych. Na Rysunku 10.17 pokazany jest *biegunowy układ współrzędnych*. Odległość d dwóch punktów p_1 i p_2 na płaszczyźnie euklidesowej, które mają współrzędne biegunowe $p_1(r_1, \varphi_1)$ i $p_2(r_2, \varphi_2)$ ma postać:

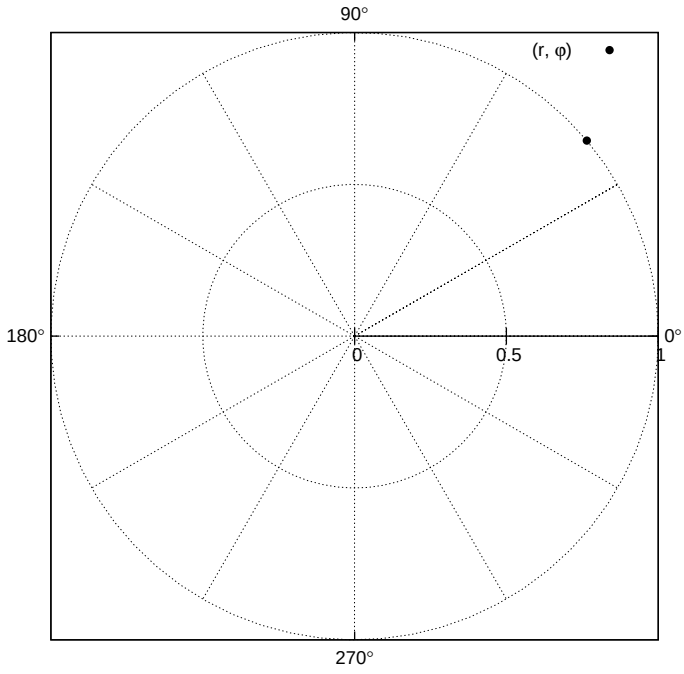
$$d(p_1, p_2) = \sqrt{r_1^2 + r_2^2 - 2r_1 r_2 \cos(\varphi_1 - \varphi_2)}. \tag{10.10}$$

W przestrzeni euklidesowej wygodnie jest czasem wprowadzić inny układ współrzędnych – *walcowy układ współrzędnych*. Nazwa tego układu (czasem ten układ jest nazywany *cylindrycznym układem współrzędnych*) pochodzi stąd, że pierwsza współrzędna walcowa r jest stała i dodatnia, leży na powierzchni pewnego walca (Rysunek 10.18). Odległość d dwóch punktów $p_1(r_1, \varphi_1, h_1)$ i $p_2(r_2, \varphi_2, h_2)$ w cylindrycznym układzie współrzędnych wyraża się wzorem:

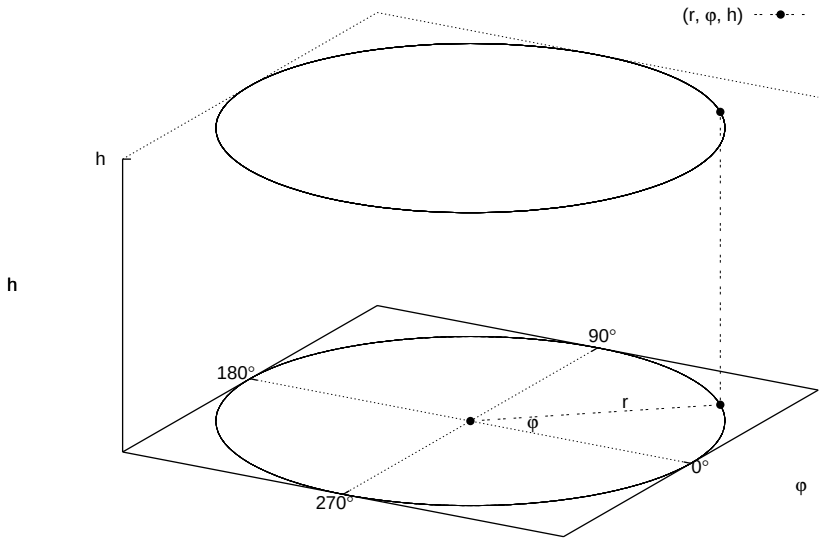
$$d(p_1, p_2) = \sqrt{r_1^2 + r_2^2 - 2r_1 r_2 \cos(\varphi_1 - \varphi_2) + (h_1 - h_2)^2}. \tag{10.11}$$



Rysunek 10.16: Prostokątny układ współrzędnych.



Rysunek 10.17: Biegunowy układ współrzędnych.



Rysunek 10.18: Walcowy układ współrzędnych.

Możemy także wykorzystać w obliczeniach *sferyczny układ współrzędnych*. Nazwa ta pochodzi stąd, że punkty, których pierwsza współrzędna sferyczna r jest stała i dodatnia, leżą na sferze o promieniu r (Rysunek 10.19). Odległość d dwóch punktów $p_1(r_1, \varphi_1, \phi_1)$ i $p2(r_2, \varphi_2, \phi_2)$ w sferycznym układzie współrzędnych wyraża się wzorem:

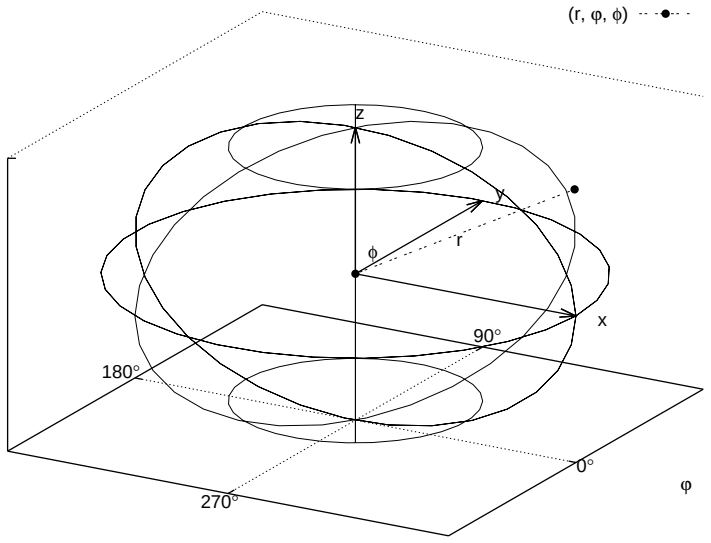
$$d(p_1, p_2) = \sqrt{r_1^2 + r_2^2 - 2r_1 r_2(\sin\varphi_1 \sin\varphi_2 \cos(\phi_1 - \phi_2) + \cos\varphi_1 \cos\varphi_2)}. \quad (10.12)$$

Okazuje się, że skomplikowany problem geometryczny może być mniej skomplikowanym, jeżeli się wybierze odpowiedni układ współrzędnych. Ten fakt wykorzystują fizycy, dobierając układ współrzędnych do rozwiązywania zadań o konkretnej symetrii.

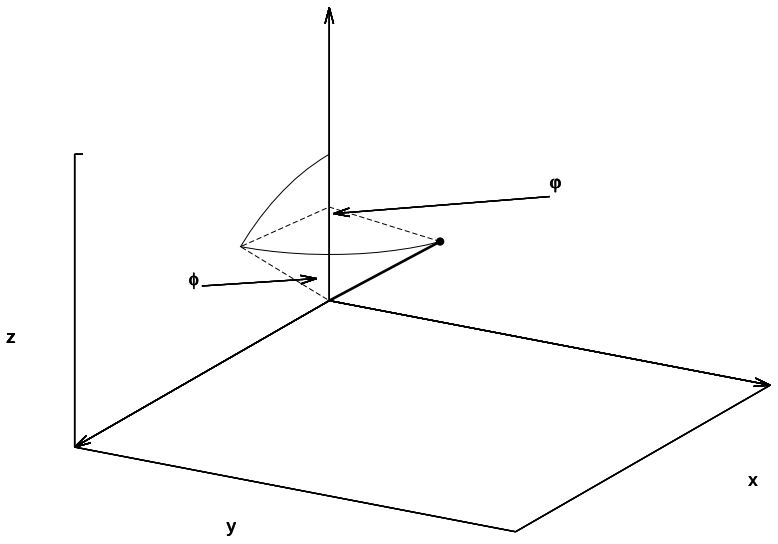
W sferycznym układzie współrzędnych położenie punktu $p(r, \varphi, \phi)$ określone jest przez podanie:

- Odległości r punktu $p(r, \varphi, \phi)$ od początku układu współrzędnych (jest to promień wodzący).
- Kąt φ jest to kąt między punktem $p(r, \varphi, \phi)$, a osią x w kierunku osi y
- Kąt ϕ jest to kąt między punktem $p(r, \varphi, \phi)$, a osią z kierunku osi x

Położenie punktu $p(r, \varphi, \phi)$ w sferycznym układzie współrzędnych pokazane jest na Rysunku 10.20. Rozwiązując zadania opisane na sferze, wygodnie jest używać



Rysunek 10.19: Sferyczny układ współrzędnych.



Rysunek 10.20: Punkt $p(r, \varphi, \phi)$ w sferycznym układzie współrzędnych.

współrzędnych sferycznych. Relacje między położeniem punktu $p(x, y, z)$ we współrzędnych prostokątnych i położeniem punktu $p(r, \varphi, \phi)$ we współrzędnych sferycznych są następujące:

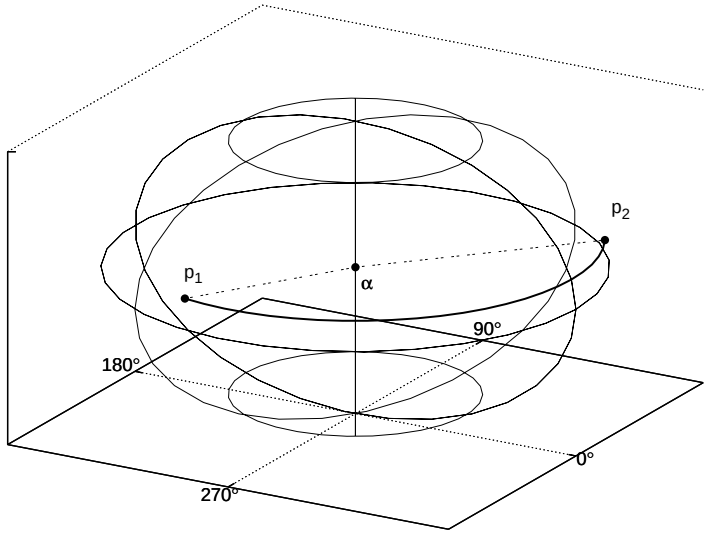
$$x = r \sin\varphi \cos\phi, \tag{10.13}$$

$$y = r \sin\varphi \sin\phi, \tag{10.14}$$

$$z = r \cos\varphi. \tag{10.15}$$

Zagadnienie wyliczania odległości na sferze ma aspekt praktyczny. Obliczamy na przykład trasy okrętów, samolotów przemieszczających się po całej kuli ziemskiej.

Zadaniem naszym jest rozwiązanie zadania wyliczenia długości łuku pomiędzy dwoma punktami p_1 i p_2 leżącymi na sferze. W rozważaniach pomocny będzie Rysunek 10.21. Jeżeli poprowadzimy promienie wodzące pomiędzy środkiem sfery



Rysunek 10.21: Położenie punktów na sferze i odległość między nimi.

i punktami p_1 i p_2 to utworzony między tymi promieniami kąt wyraża się wzorem:

$$\alpha = \cos^{-1}\left(\frac{x_1x_2 + y_1y_2 + z_1z_2}{r^2}\right). \tag{10.16}$$

Długość łuku s wyraża się wzorem:

$$s = \alpha r. \tag{10.17}$$

Algorytm wyliczania długości łuku na powierzchniach sferycznych. Położenie punktów podajemy we współrzędnych sferycznych. Należy podać współrzędne punktu $p_1(r_1, \varphi_1, \phi_1)$ i $p_2(r_2, \varphi_2, \phi_2)$, które znajdują się na powierzchni sfery (Rysunek 10.21). Proste łączące te punkty z początkiem układu współrzędnych utworzą kąt α. Dzięki znajomości kąta α możemy określić jaka część obwodu koła zajmuje łuk utworzony przez nasze punkty. Część ta wynosi:

$$\frac{\alpha}{2\pi}.$$

Pamiętamy, że 2π jest to długość okręgu w radianach. Wiemy, że wzór na obwód koła wynosi $2\pi r$, skąd obliczamy długość łuku:

$$\frac{\alpha}{2\pi} 2\pi r = \alpha r. \tag{10.18}$$

W algorytmie w celu wyliczenia kąta α dokonujemy konwersji położenia punktów ze współrzędnych sferycznych do współrzędnych prostokątnych (10.13), (10.14), (10.15). Kod źródłowy do wyznaczenia długości łuku pokazany jest na Listingu 10.7

Listing 10.7: Obliczanie długości łuku.

```
1  #include<iostream>
2  #include<cmath>
3
4  const double RAD = 3.1415926 / 180.0;
5
6  int main()
7  {
8    double p1_x, p1_y, p1_z, p2_x, p2_y, p2_z;
9      // p ( r ,  varphi ,  phi )
10   double p1_r = 1.0;
11   double p1_phi = 0.0;
12   double p1_varphi = 90.0;
13
14   double p2_r = 1.0;
15   double p2_phi = 90.0;
16   double p2_varphi = 90.0;
17
18   p1_x = p1_r * std::sin(p1_varphi * RAD) * std::cos(p1_phi * RAD);
19   p1_y = p1_r * std::sin(p1_varphi * RAD) * std::sin(p1_phi * RAD);
20   p1_z = p1_r * std::cos(p1_varphi * RAD);
21   std::cout << "koordynaty 1 punktu:" << std::endl;
22   std::cout << p1_x << std::endl;
23   std::cout << p1_y << std::endl;
24   std::cout << p1_z << std::endl;
25
26   p2_x = p2_r * std::sin(p2_varphi * RAD) * std::cos(p2_phi * RAD);
27   p2_y = p2_r * std::sin(p2_varphi * RAD) * std::sin(p2_phi * RAD);
28   p2_z = p2_r * std::cos(p2_varphi * RAD);
29
30   std::cout << "\nkoordynaty 2 punktu:" << std::endl;
31   std::cout << p2_x << std::endl;
32   std::cout << p2_y << std::endl;
33   std::cout << p2_z << std::endl;
34
35   double alpha = std::acos(((p1_x * p2_x) + (p1_y * p2_y) + (p1_z *
         p2_z)) / std::pow(p1_r, 2.0));
```

```
36    std::cout << "\ndlugosc luku = " << alpha * p1_r << std::endl;
37    return 0;
38 }
```

Dla podanych punktów wynik obliczeń jest zgodny z oczekiwaniem:

```
1    koordynaty 1 punktu:
2    1
3    0
4    2.67949e-08
5
6    koordynaty 2 punktu:
7    2.67949e-08
8    1
9    2.67949e-08
10
11   dlugosc luku = 1.5708
```

Opisana metoda wyznaczania długości łuku może być wykorzystana do wyznaczania odległości między punktami położonymi na powierzchni Ziemi. Nie jest to zadanie zbyt trudne, ale musimy brać pod uwagę pewne uwarunkowania geograficzne. Położenie na Ziemi w systemach geograficznych definiuje się podając szerokość i długość geograficzną. Szerokość geograficzna na równiku ma wartość 0°, na biegunie Ziemi – 90°. Mamy dwa bieguny, wobec tego dla półkuli północnej (biegun północny) wartość szerokości oznacza się symbolem N (ang. *north*), dla półkuli południowej mamy oznaczenie S (ang. *south*). Długości geograficzne zmieniają się od wartości 0° (dla tzw. pierwszego południka) do 180°, w obu kierunkach. Położenia od pierwszego południka w kierunku na zachód oznaczane są symbolem W (ang. *west*), a dla położeń w kierunku wschodnim – oznaczane są symbolem E (ang. *east*). Dla wygody (np. w obliczeniach komputerowych) wprowadza się specjalna konwencję: długości zachodnie mają wartość dodatnią, długości wschodnie są ujemne.

W monografii K. Loudona szczegółowo jest omówione zagadnienie wyznaczania odległości pomiędzy punktami na Ziemi pomiędzy Paryżem i Perth. Jak podaje Loudon, położenie Paryża to $(49,010N$ i $2,548E)$ a położenie Perth w Australii to $(32,940S$ i $115,967E)$. Schemat obliczeń jest następujący: należy najpierw przeliczyć położenie geograficzne na współrzędne sferyczne i kąty na radiany. Współrzędne sferyczne to trójka liczb: r, φ i ϕ. Wiemy, że r jest promieniem Ziemi, wartość ta wynosi $3440,065$ mil morskich. Współrzędna φ to jest długością geograficzną, a współrzędna ϕ jest szerokością geograficzną. Położenia miejscowości we współrzędnych sferycznych są następujące:

- Paryż: $(3440,056; 2,548; 40,990)$,
- Perth: $(3440,065; 115,967; 121,940)$.

Wyznaczona odległość między tymi miastami to 7706 mil morskich.

Listing 10.8: Obliczanie odległości geograficznej.

```
1 #include<iostream>
2 #include<cmath>
```

```
3
4  inline double rad_to_deg(double rad) {return (rad * 360.0) / (2.0 *
       3.1415926);}
5  inline double deg_to_rad(double deg) {return deg * 2.0 * 3.1415926 /
       360.0;}
6  const double EARTH_RADIUS = 3440.065;
7
8  int main()
9  {
10   double lat1, lon1, lat2, lon2;
11   double droga, alpha, ile;
12   double p1_x, p1_y, p1_z, p2_x, p2_y, p2_z;
13     // p ( r ,   varphi ,   phi )
14     // Paryz
15   lat1 = 49.010;
16   lon1 = -2.548;
17     // Perth
18   lat2 = -32.940;
19   lon2 = -115.967;
20
21   if (lat1 < -90.0 || lat1 > 90.0 || lat2 < -90.0 || lat2 > 90.0)
22     std::cout << "blad danych" << std::endl;
23   if (lon1 < -180.0 || lon1 > 180.0 || lon2 < -180.0 || lon2 > 180.0)
24     std::cout << "blad danych" << std::endl;
25
26   double p1_r = EARTH_RADIUS;
27   double p1_phi = -1.0 * deg_to_rad(lon1);
28   double p1_varphi = (deg_to_rad(-1.0 * lat1)) + deg_to_rad(90.0);
29
30   std::cout << "koordynaty 1 punktu, sferyczne:" << std::endl;
31   std::cout << p1_r << std::endl;
32   std::cout << rad_to_deg(p1_phi) << std::endl;
33   std::cout << rad_to_deg(p1_varphi) << std::endl;
34
35   double p2_r = EARTH_RADIUS;
36   double p2_phi = -1.0 * deg_to_rad(lon2);;
37   double p2_varphi = (deg_to_rad(-1.0 * lat2)) + deg_to_rad(90.0);
38
39   std::cout << "koordynaty 2 punktu, sferyczne:" << std::endl;
40   std::cout << p2_r << std::endl;
41   std::cout << rad_to_deg(p2_phi) << std::endl;
42   std::cout << rad_to_deg(p2_varphi) << std::endl;
43
44   p1_x = p1_r * std::sin(p1_varphi) * std::cos(p1_phi);
45   p1_y = p1_r * std::sin(p1_varphi) * std::sin(p1_phi);
46   p1_z = p1_r * std::cos(p1_varphi);
47
48   p2_x = p2_r * std::sin(p2_varphi) * std::cos(p2_phi);
49   p2_y = p2_r * std::sin(p2_varphi) * std::sin(p2_phi);
50   p2_z = p2_r * std::cos(p2_varphi);
51
52   alpha = std::acos(((p1_x * p2_x) + (p1_y * p2_y) + (p1_z * p2_z)) /
         std::pow(p1_r, 2.0));
53   std::cout << "\nodleglosc Paryz - Pert = " << alpha * p1_r <<" mil
         morskich"<< std::endl;
54   return 0;
55 }
```

Po uruchomieniu tego program mamy następujący wynik:

```
koordynaty 1 punktu, sferyczne:
3440.07
2.548
40.99
koordynaty 2 punktu, sferyczne:
3440.07
115.967
122.94

odleglosc Paryz - Pert = 7744.83 mil morskich
```

Bibliografia

[1] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych*. Wydawnictwo Naukowo-Techniczne, Warszawa, 1996.

[2] B. Baron, Ł. Piątek. *Metody numeryczne w C++ Builder*. Helion, Gliwice, 2004.

[3] A. Bjorck, G. Dahlquist. *Metody numeryczne*. Państwowe Wydawnictwo Naukowe, Warszawa, 1987.

[4] K. Borsuk. *Geometria analityczna w n wymiarach*. Wydawnictwo Czytelnik, 1950.

[5] I. Parberry F. Dunn. *3D Math primer for graphics and game development*. Jones & Bartlett Publishers, 2002.

[6] G. Forsythe, M. Malcom, C. Moler. *Computer methods for mathematical computations*. Prentice-Hall, Inc., New York, 1977.

[7] R. Graham, D. Knuth, O. Patashnik. *Matematyka konkretna*. Wydawnictwo Naukowe PWN, Warszawa, 1996.

[8] R. Hornbeck. *Numerical Methods*. Quantum Publishers, Inc., New York, 1975.

[9] S. Jermakow. *Metoda Monte Carlo i zagadnienia pokrewne*. Państwowe Wydawnictwo Naukowe, Warszawa, 1976.

[10] R. Johnston. *Numerical methods*. John Wiley & Sons, New York, 1982.

[11] E. Kalinowska, K. Kalinowski. *Metody numeryczne*. Wyższa Szkoła Informatyki i Zarządzania, Bielsko Biała, 2003.

[12] B. W. Kernighan, D. M. Ritchie. *Język ANSI C*. Wydawnictwo Naukowo-Techniczne, Warszawa, 2003.

[13] K. Loudon. *Algorytmy w C*. Helion, Gliwice, 2003.

[14] J. O'Rourke. *Computational geometry in C*. Cambridge University Press, New York, 1998.

[15] J. Pachner. *Handbook of numerical analysis applications*. McGraw-Hill, Inc., 1984.

[16] S. Prata. *Język C++. Szkoła programowania*. Helion, Gliwice, 2006.

[17] F. Preparata, M. Shamos. *Geometria obliczeniowa*. Helion, Gliwice, 2003.

[18] W. Press, S. Teukolsky, W. Vetterling, B. Flannery. *Numerical Recipies in C*. Cambridge University Press, 1995.

[19] A. Ralston. *Wstęp do analizy numerycznej*. Państwowe Wydawnictwo Naukowe, Warszawa, 1983.

[20] L. F. Shampine, R. C. Allen. *Numerical computing: an introduction*. W. B. Saunders Company, Philadelphia, 1973.

[21] K. Sieklucki. *Geometria i topologia*. Państwowe Wydawnictwo Naukowe, Warszawa, 1978.

[22] M. Stark. *Geometria analityczna*. Państwowe Wydawnictwo Naukowe, Warszawa, 1958.

[23] P. Stańczyk. *Algorytmy praktyczne*. Wydawnictwo Naukowe PWN, Warszawa, 2009.

[24] D. Stephens, C. Diggins, J. Turkanis, J. Cogswell. *C++ receptury*. Helion, Gliwice, 2006.

[25] B. Stroustrup. *Programowanie, teoria i praktyka z wykorzystaniem C++*. Helion, Gliwice, 2010.

[26] J. Tyszer. *Symulacja cyfrowa*. Wydawnictwo Naukowo-Techniczne, Warszawa, 1990.

[27] R. Wieczorkowski, R. Zielinski. *Komputerowe generatory liczb losowych*. Wydawnictwo Naukowo-Techniczne, Warszawa, 1997.