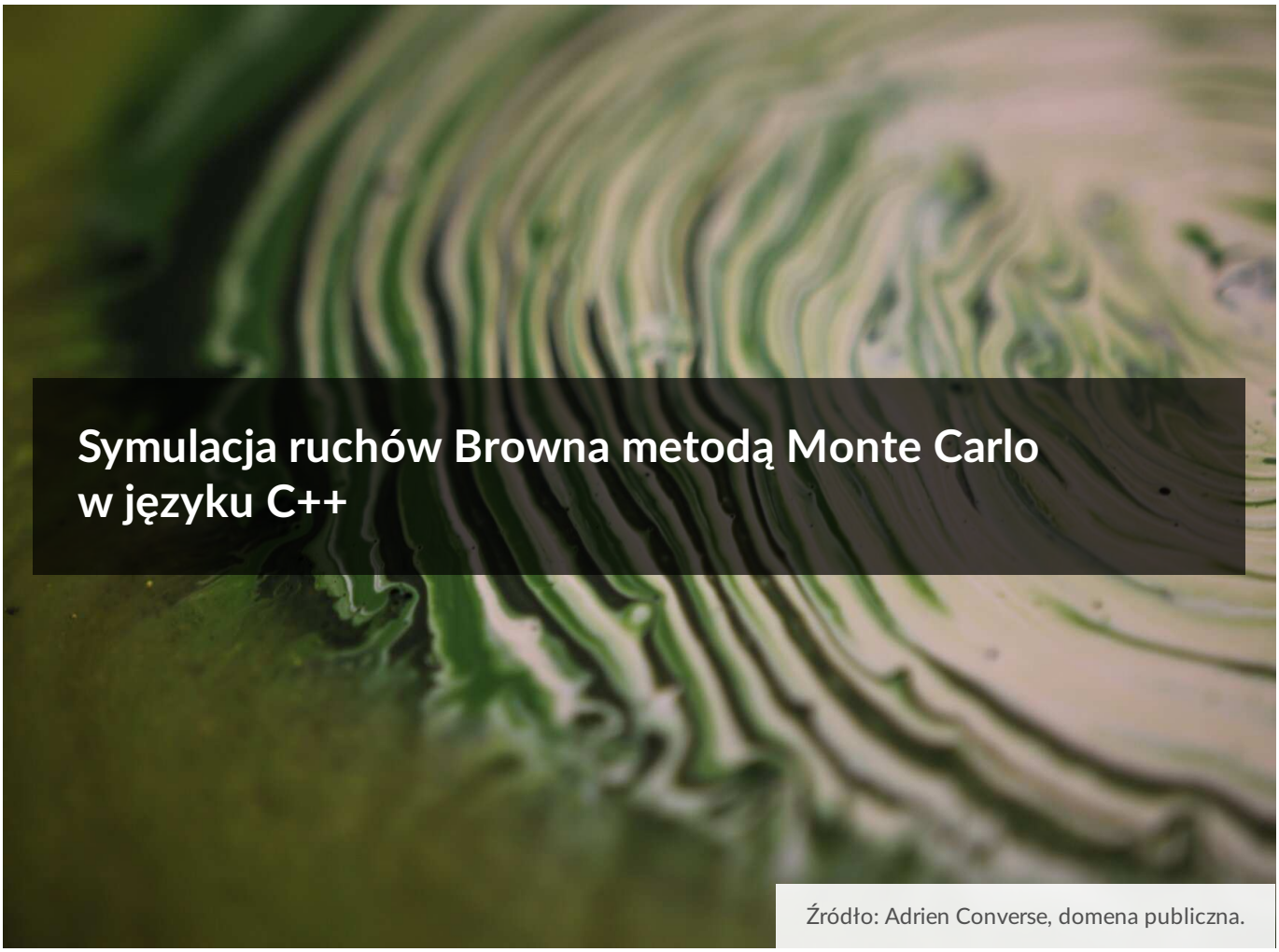
Symulacja ruchów Browna metodą Monte Carlo w języku C++



Symulacja ruchów Browna metodą Monte Carlo w języku C++

Źródło: Adrien Converse, domena publiczna.

Znasz już teorię dotyczącą ruchów Browna, nadszedł więc moment na to, aby prześledzić poruszanie się cząstki na ekranie.

Więcej teorii oraz ćwiczeń znajdziesz w:

- Symulacja ruchów Browna metodą Monte Carlo,
- Symulacja ruchów Browna metodą Monte Carlo – zadania maturalne.

Ciekawi cię, jak wyglądają implementacje w innych językach programowania? Możesz się z nimi zapoznać w dwóch pozostałych materiałach z tej serii:

- Symulacja ruchów Browna metodą Monte Carlo w języku Python,
- Symulacja ruchów Browna metodą Monte Carlo w języku Java.

Twoje cele

- Poznasz sposób wykorzystania rozkładu normalnego Gaussa w języku >C++.
- Zbadasz poruszanie się cząstki zgodnie z procesem Wienera.
- Zinterpretujesz analogię pomiędzy procesem Wienera a ruchami Browna.

Film samouczek

Problem 1

Specyfikacja problemu:

Napisz program, który będzie symulować ruch chodziarza na płaszczyźnie jednowymiarowej.

Dane:

- `liczbaEksperymentow` – liczba naturalna
- `maksymalnaLiczbaKrokow` – liczba naturalna

Wynik:

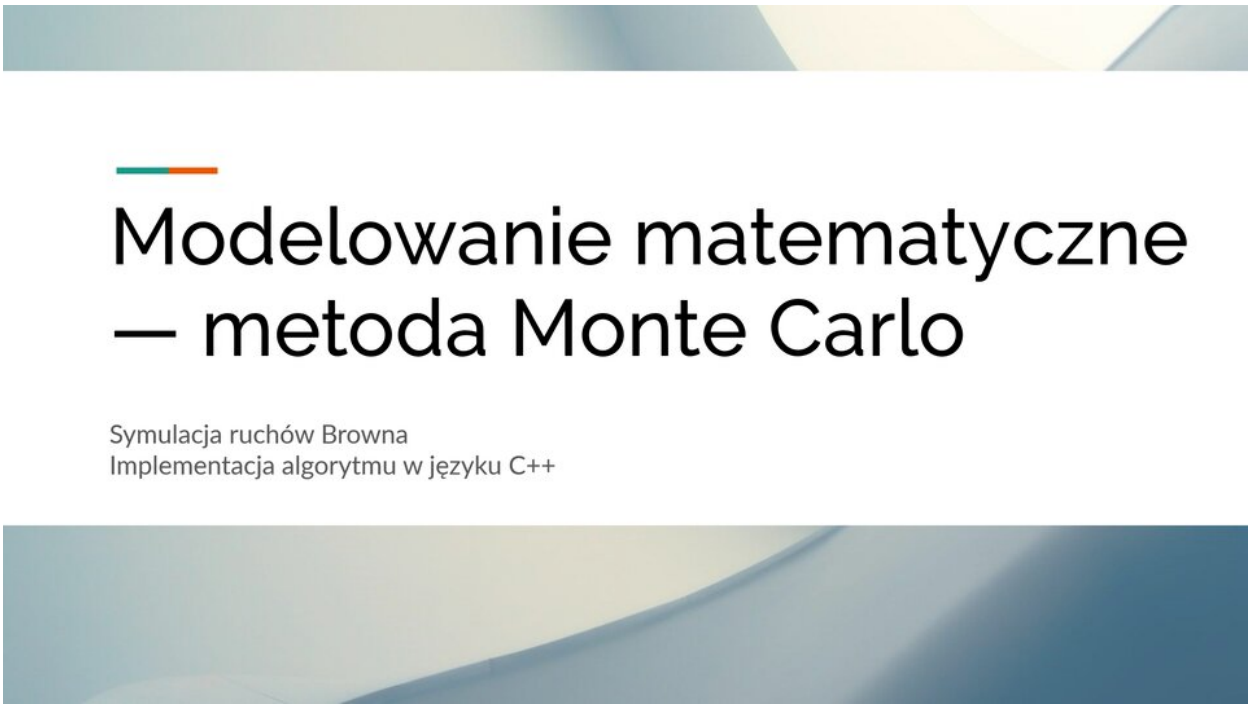
- Program wyświetla wartość średniego przemieszczenia i średniej odległości w zależności od liczby kroków, które może wykonać chodziarz.

```
1
```

```
1
```

Polecenie 1

Porównaj swoje rozwiązanie z filmem.



Film dostępny pod adresem /preview/resource/R5hFBQuIEgyNA

Film nawiązujący do treści materiału - dotyczy modelowania matematycznego metodą Monte Carlo.

Kod programu zaprezentowanego w filmie:

Plik o rozmiarze 1.21 KB w języku polskim

Przeczytaj

Konfiguracja środowiska

W e-materiale Modelowanie ruchów Browna przeprowadziliśmy proste modelowanie, w którym zakładaliśmy, że w każdej jednostce czasu cząsteczka przemieszcza się o określoną z góry odległość. W ramach przypomnienia zrealizujmy identyczny algorytm w środowisku Qt.

W tej sekcji wykorzystamy projekt omówiony w e-materiale Konstrukcje fraktalne w języku C++.

Potrzebne nam będzie prawidłowo skonfigurowane środowisko Qt oraz przykładowy projekt, który utworzyliśmy. Wykorzystamy również stworzony specjalnie na potrzeby e-materiału szablon, którego zadaniem będzie rysowanie trasy cząstki na podstawie dostarczonych mu punktów.

Zacznijmy zatem od wprowadzenia odpowiednich zmian do projektu.

W pliku *mainwindow.h* należy umieścić następujący kod:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <list>

using namespace std;

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

    virtual void paintEvent(QPaintEvent *event) override;
```

```
21
22      void addPoint(double x, double y);
23      void wait(double time);
24
25      static const int PARTICLE_SIZE = 10;
26
27  private:
28      Ui::MainWindow* ui;
29      list<QPoint> points;
30  };
31  #endif // MAINWINDOW_H
```

Analogicznie w pliku *mainwindow.cpp* wprowadzamy zmiany w kodzie:

```
1   #include "mainwindow.h"
2   #include "ui_mainwindow.h"
3   #include "qpainter.h"
4   #include <QTime>
5
6   using namespace std;
7
8   MainWindow::MainWindow(QWidget *parent)
9       : QMainWindow(parent)
10      , ui(new Ui::MainWindow)
11  {
12      ui->setupUi(this);
13      this->setFixedSize(600, 600);
14      this->setWindowTitle("Symulacja ruchów Browna");
15  }
16
17  MainWindow::~MainWindow() {
18      delete ui;
19  }
20
21  void MainWindow::addPoint(double x, double y) {
22      int new_x = x * 50.0 + size().width()/2;
23      int new_y = size().height()/2 - y * 50.0;
24
25      points.push_back(QPoint(new_x, new_y));
26      repaint();
27  }
```

```
28
29 void MainWindow::wait(double time) {
30     QTime dieTime= QTime::currentTime().addMSecs(time * 1000.0);
31     while (QTime::currentTime() < dieTime)
32         QCoreApplication::processEvents(QEventLoop::AllEvents, 10
33 }
34
35 void MainWindow::paintEvent(QPaintEvent *event) {
36     QPainter painter(this);
37
38     auto last = points.begin();
39     for (auto it = points.begin(); it != points.end(); ++it) {
40         if (it != points.begin()) {
41             painter.drawLine(*it, *last);
42             last = it;
43         }
44     }
45
46     if (!points.empty()) {
47         painter.setBrush(Qt::red);
48         painter.drawEllipse(last->x() - PARTICLE_SIZE / 2, last->
49     }
50 }
51
52 void MainWindow::closeEvent(QCloseEvent *event) {
53     exit(0);
54 }
```

Z kolei plik *main.cpp* zmieniamy na:

```
1  #include "mainwindow.h"
2
3  #include <QApplication>
4  #include <cstdlib>
5  #include <cmath>
6  #include <random>
7
8  using namespace std;
9
10 int main(int argc, char *argv[]) {
11     // Początek kodu - utworzenie okna do rysowania
```

```
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    /* początek właściwej części kodu */

    /* koniec właściwej części kodu */

    // Koniec programu - wyświetlenie wyniku
    return a.exec();
}
```

W szablonie dla klasy `MainWindow` dostępne są dwie interesujące nas metody: `addPoint(double x, double y)` oraz `wait(double time)`. Pierwsza pozwala na dodanie punktu do trasy, którą renderuje program. Druga natomiast powoduje odczekanie przez program określonego w parametrze `time` czasu w sekundach. Ponieważ na początku programu w szablonie utworzony jest obiekt klasy `MainWindow` o nazwie `w`, przykładowe wywołanie metod może wyglądać następująco:

```
w.addPoint(x, y);
w.wait(dt);
```

Takie działanie spowoduje dodanie punktu o współrzędnych `x`, `y` do tworzonej przez program trasy oraz sprawi, że będzie potrzebny czas oczekiwania na reakcję, zawarty w zmiennej `dt`.

Symulacja procesu Wienera w środowisku graficznym

Jak wiemy, proces Wienera jest procesem losowym z czasem ciągłym. Możemy określać jego stan po pewnym interwale czasowym. Uproszczona definicja wygląda następująco:

$$W(0) = 0$$

$$W(t + dt) = W(t) + k \cdot N(0,\ dt)$$

- gdzie $N(\mu, \sigma^2)$ jest zmienną losową, zgodną z rozkładem normalnym Gaussa z parametrami wartość oczekiwana μ oraz odchylenie standardowe σ. Przy czym k jest współczynnikiem długości wykonywanych kroków.

Pierwszym i największym problemem związanym z implementacją ruchów Browna z wykorzystaniem procesu Wienera, z jakim się spotykamy, jest generowanie losowej liczby, zgodnie z rozkładem normalnym Gaussa. Ponieważ ten e-materiał nie jest poświęcony tematyce rozkładów prawdopodobieństwa, skorzystamy z dostarczanej wraz z językiem C++ biblioteką `<random>`, która jest częścią standardu języka od wersji C++11. Zawiera ona różne generatory liczb pseudolosowych oraz implementacje najbardziej popularnych rozkładów prawdopodobieństwa, m.in. rozkładu normalnego Gaussa.

Dostarczone generatory oraz rozkłady dostępne są jako klasy, zatem potrzebne będzie utworzenie zarówno generatora, jak i rozkładu. Można to osiągnąć za pomocą następującego fragmentu kodu. W przedstawionym kodzie wartości, którymi inicjujemy rozkład, są przykładowe. W późniejszym etapie zostaną zastąpione w taki sposób, aby spełniały definicję procesu Wienera.

```
double wartosc_oczekiwana = 0.0,
    odchylenie_standardowe = 0.3;

default_random_engine generator;
normal_distribution<double> rozklad(wartosc_oczekiwana, odchyleni
```

Wygenerowanie losowej liczby zgodnej z rozkładem normalnym może się odbyć za pomocą następującej instrukcji:

```
double losowa_liczba = rozklad(generator);
```

Przejdźmy zatem do realizacji procesu Wienera. Ponieważ chcemy stworzyć dwuwymiarową symulację ruchów Browna, musimy przeprowadzić równolegle dwa procesy Wienera. Jeden z nich będzie opisywał ruch cząsteczki według współrzędnej x, drugi - według współrzędnej y. Zgodnie z definicją, proces zaczyna się w punkcie początkowym 0. Zainicjujmy zatem na początek obie zmienne wartością 0.0.

```
// początkowe położenie
double x = 0.0,
    y = 0.0;
```

Warto już teraz zastanowić się, w jakich odstępach czasowych chcemy sprawdzać „stan" naszej cząstki. Pomyślmy także, jak długo powinna trwać symulacja. Aby płynnie obserwować ruchy cząstki, proponujemy przyjąć parametr `dt = 0.1`. Na początku spróbujmy przeprowadzić 10-sekundową symulację. Optymalny współczynnik długości

wykonywanych kroków dla utworzonego szablonu to `k = 1`. Aby cząstka nie uciekła poza granice okna, ustawmy wartość `k` właśnie na tę wartość.

```
// początkowe położenie
double x = 0.0,
    y = 0.0;

double t = 0.0,          // aktualny czas symulacji
    t_max = 10.0,        // długość symulacji
    dt = 0.1,            // interwał czasowy
    k = 1.0;             // współczynnik długości kroków
```

Utwórzmy pętlę, która będzie się wykonywać do momentu, gdy czas przekroczy 10 sekund.

```
// początkowe położenie
double x = 0.0,
    y = 0.0;

double t = 0.0,          // aktualny czas symulacji
    t_max = 10.0,        // długość symulacji
    dt = 0.1,            // interwał czasowy
    k = 1.0;             // współczynnik długości kroków

while (t < t_max) {
    // zwiększ aktualny czas
    t += dt;
}
```

Przejdźmy zatem do przemieszczania cząstki. Zgodnie z definicją procesu Wienera, wartość oczekiwana, którą powinniśmy zainicjować rozkład, wynosi 0. Wówczas generator będzie zwracał równą liczbę wartości dodatnich i ujemnych. Odchylenie standardowe jest pierwiastkiem wariancji, która w definicji procesu Wienera wynosi *dt*. Skorzystajmy z funkcji `sqrt()` z biblioteki `<cmath>`, aby obliczyć jej pierwiastek. Ponieważ wartość oczekiwana i odchylenie standardowe będą stałe, możemy utworzyć rozkład przed wejściem do pętli `while`. Przemieszczenie będzie odbywało się zatem o losową liczbę, wygenerowaną przez rozkład oraz generator (ewentualnie pomnożoną przez parametr `k`).

```
// początkowe położenie
double x = 0.0,
```

```
    y = 0.0;

double t = 0.0,          // aktualny czas symulacji
    t_max = 10.0,        // długość symulacji
    dt = 0.1,            // interwał czasowy
    k = 1.0;             // współczynnik długości kroków

// utworzenie generatora i rozkładu
default_random_engine generator;
normal_distribution<double> rozklad(0.0, sqrt(dt));

while (t < t_max) {
    // przemieszczenie
    x += k * rozklad(generator);
    y += k * rozklad(generator);

    // zwiększ aktualny czas
    t += dt;
}
```

Ostatnim krokiem będzie zapisanie instrukcji dodającej współrzędne kolejnego punktu do rysowanej trasy co jednostkę czasu oraz odczekanie odpowiedniego czasu, aby obserwować animację w sposób odzwierciedlający symulację.

```
// początkowe położenie
double x = 0.0,
    y = 0.0;

double t = 0.0,          // aktualny czas symulacji
    t_max = 10.0,        // długość symulacji
    dt = 0.1,            // interwał czasowy
    k = 1.0;             // współczynnik długości kroków

// utworzenie generatora i rozkładu
default_random_engine generator;
normal_distribution<double> rozklad(0.0, sqrt(dt));

// rysuj pierwszą pozycję
w.addPoint(x, y);

while (t < t_max) {
```

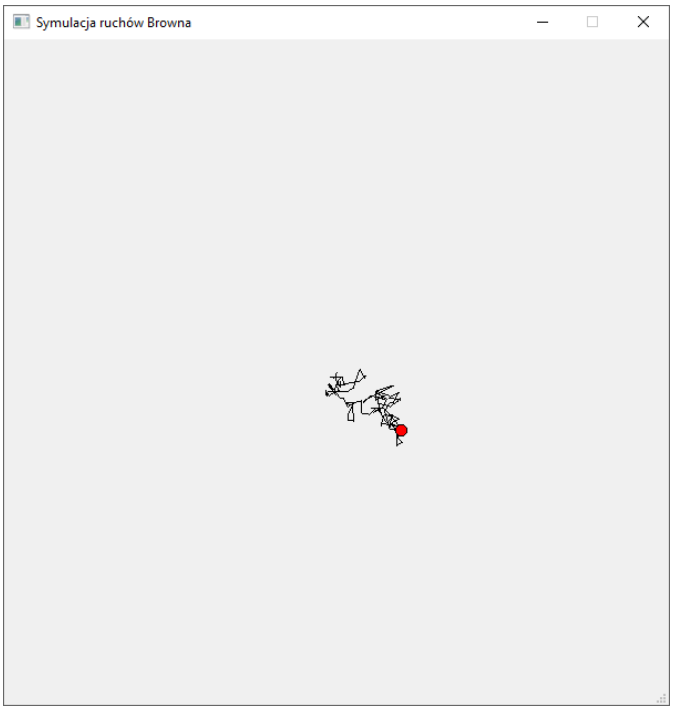
```
    // przemieszczenie
    x += k * rozklad(generator);
    y += k * rozklad(generator);

    // zwiększ aktualny czas
    t += dt;

    // odczekaj dt
    w.wait(dt);
    // rysuj aktualną pozycję
    w.addPoint(x, y);
}
```

Umieśćmy teraz napisany kod w szablonie i sprawdźmy jego działanie:

```
#include "mainwindow.h"

#include <QApplication>
#include <cstdlib>
#include <cmath>
#include <random>

using namespace std;

int main(int argc, char *argv[]) {
    // Początek kodu - utworzenie okna do rysowania
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    /* początek właściwej części kodu */
    // początkowe położenie
    double x = 0.0,
        y = 0.0;

    double t = 0.0,          // aktualny czas symulacji
        t_max = 10.0,        // długość symulacji
        dt = 0.1,            // interwał czasowy
        k = 1.0;             // współczynnik długości kroków

    // utworzenie generatora i rozkładu
```

```
    default_random_engine generator;
    normal_distribution<double> rozklad(0.0, sqrt(dt));

    // rysuj pierwszą pozycję
    w.addPoint(x, y);

    while (t < t_max) {
        // przemieszczenie
        x += k * rozklad(generator);
        y += k * rozklad(generator);

        // zwiększ aktualny czas
        t += dt;

        // odczekaj dt
        w.wait(dt);
        // rysuj aktualną pozycję
        w.addPoint(x, y);
    }


    /* koniec właściwej części kodu */

    // Koniec programu
    return a.exec();
}
```

Po uruchomieniu programu wyświetli się okno, w którym rysowana będzie aktualna pozycja cząstki oraz przebyta przez nią trasa. Oto przykładowy stan końcowy okna:



Źródło: Contentplus.pl Sp. z o.o., licencja: CC BY-SA 3.0.

Jak widzimy, można zwiększyć czas symulacji. Oto jak przedstawia się stan symulacji po czasie 100 sekund:

Źródło: Contentplus.pl Sp. z o.o., licencja: CC BY-SA 3.0.

Słownik

interwał

odstęp w czasie lub odległość w przestrzeni

renderowanie

(ang. *rendering*) przedstawienie wyników cyfrowych za pomocą formy odpowiedniej dla wybranych danych (np. graficznie lub dźwiękowo)

Sprawdź się

Pokaż ćwiczenia: 

Ćwiczenie 1

W programie utworzono funkcję, która generuje losowe liczby zgodnie z rozkładem normalnym Gaussa. Przeprowadź eksperyment, w którym sprawdzisz, jaka jest wartość oczekiwana (średnia) zwracanych przez funkcję liczb. W tym celu wygeneruj za jej pomocą n liczb i oblicz ich średnią wartość. Wynik zaokrąglij do jednego miejsca po przecinku.

Aby móc poprawnie sprawdzić swoje rozwiązanie, nie zmieniaj funkcji `losowa_liczba_gauss()`. Wartość zmiennej `mean` w funkcji powinna wynosić 4612811918334230528.

Działanie programu przetestuj dla `n = 100000`.

Specyfikacja problemu:

Dane:

- n – liczba naturalna

Wynik:

- `wartosc_oczekiwana` – liczba rzeczywista

Twoje zadania

1. Program sprawdza wartość oczekiwaną zwracaną przez funkcję `losowa_liczba_gauss`.

Ćwiczenie 2

W programie utworzono dwuwymiarową tablicę, która zawiera losowe wektory przesunięcia. Przeprowadź prostą symulację, polegającą na przesuwaniu cząstki o kolejne wektory z tablicy. Wypisz końcową pozycję jako dwie liczby oddzielone znakiem spacji, zaokrąglone do dwóch miejsc po przecinku. Symulację rozpocznij od punktu $(a,\ b)$. Działanie programu przetestuj dla punktu $(1.42,\ -2.42)$ i danej tablicy z wektorami.

Specyfikacja problemu:

Dane:

- `wektory` – dwuwymiarowa tablica liczb rzeczywistych
- `x` – liczba rzeczywista; pierwsza współrzędna punktu początkowego
- `y` – liczba rzeczywista; druga współrzędna punktu początkowego

Wynik:

- `xk` – liczba rzeczywista; pierwsza współrzędna punktu końcowego
- `yk` – liczba rzeczywista; pierwsza współrzędna punktu końcowego

Twoje zadania

1. Program sprawdza końcową pozycję cząstki poruszającej się według wektorów zawartych w tablicy.

Dla nauczyciela

Autor: Maurycy Gast
Przedmiot: Informatyka

Temat: Symulacja ruchów Browna metodą Monte Carlo w języku C++

Grupa docelowa:

Szkoła ponadpodstawowa, liceum ogólnokształcące, technikum, zakres rozszerzony

Podstawa programowa:

Cele kształcenia – wymagania ogólne

I. Rozumienie, analizowanie i rozwiązywanie problemów na bazie logicznego i abstrakcyjnego myślenia, myślenia algorytmicznego i sposobów reprezentowania informacji.

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera oraz innych urządzeń cyfrowych: układanie i programowanie algorytmów, organizowanie, wyszukiwanie i udostępnianie informacji, posługiwanie się aplikacjami komputerowymi.

Treści nauczania – wymagania szczegółowe

I + II. Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

3) objaśnia, a także porównuje podstawowe metody i techniki algorytmiczne oraz struktury danych, wykorzystując przy tym przykłady problemów i algorytmów, w szczególności:

h) metodę Monte Carlo (obliczanie przybliżonej wartości liczby π, symulacja ruchów Browna),

Kształtowane kompetencje kluczowe:

- kompetencje cyfrowe;
- kompetencje osobiste, społeczne i w zakresie umiejętności uczenia się;
- kompetencje matematyczne oraz kompetencje w zakresie nauk przyrodniczych, technologii i inżynierii.

Cele operacyjne (językiem ucznia):

- Poznasz sposób wykorzystania rozkładu normalnego Gaussa w języku >C++.

- Zbadasz poruszanie się cząstki zgodnie z procesem Wienera.
- Zinterpretujesz analogię pomiędzy procesem Wienera a ruchami Browna.

Strategie nauczania:

- konstruktywizm;
- konektywizm.

Metody i techniki nauczania:

- dyskusja;
- rozmowa nauczająca z wykorzystaniem multimedium i ćwiczeń interaktywnych;
- metody aktywizujące.

Formy pracy:

- praca indywidualna;
- praca w parach;
- praca w grupach;
- praca całego zespołu klasowego.

Środki dydaktyczne:

- komputery z głośnikami, słuchawkami i dostępem do internetu;
- zasoby multimedialne zawarte w e-materiale;
- tablica interaktywna/tablica, pisak/kreda;
- telefony z dostępem do internetu;
- oprogramowanie dla języka C++, w tym kompilator GCC/G++ 4.5 (lub nowszej wersji) i Code::Blocks 16.01 (lub nowszej wersji), Orwell Dev-C++ 5.11 (lub nowszej wersji) lub Microsoft Visual Studio.

Przebieg lekcji

Przed lekcją:

1. **Przygotowanie do zajęć.** Nauczyciel loguje się na platformie i udostępnia e-materiał: „Symulacja ruchów Browna metodą Monte Carlo w języku C++". Uczniowie zapoznają się z treściami w sekcji „Film samouczek" w kontekście programowania.

Faza wstępna:

1. Przedstawienie tematu zajęć oraz wspólne z uczniami ustalenie kryteriów sukcesu.
2. **Rozpoznanie wiedzy uczniów.** Uczniowie tworzą pytania dotyczące tematu zajęć, na które odpowiedzą w trakcie lekcji.

Faza realizacyjna:

1. **Praca z tekstem.** Jeżeli przygotowanie uczniów do lekcji jest niewystarczające, nauczyciel prosi o indywidualne zapoznanie się z treścią zawartą w sekcji „Film samouczek". Każdy uczestnik zajęć podczas cichego czytania wynotowuje najważniejsze kwestie poruszane w tekście.
2. **Praca z multimedium.** Nauczyciel wyświetla zawartość sekcji „Film samouczek". Uczniowie wspólnie zapoznają się z treścią zawartego w niej multimedium. Zapisują ewentualne problemy i pytania. Po czym następuje dyskusja, w trakcie której nauczyciel wyjaśnia niezrozumiałe treści. Następnie wykonują przedstawiony w filmie program.
3. **Ćwiczenia umiejętności.** Uczniowie wykonują ćwiczenie nr 1 z sekcji „Sprawdź się". W przedstawionym programie utworzono funkcję, która generuje losowe liczby zgodnie z rozkładem normalnym Gaussa. Zadaniem uczniów jest przeprowadzenie eksperymentu, w którym sprawdzą, jaka jest wartość oczekiwana (średnia) zwracanych przez funkcję liczb. W tym celu generują za jej pomocą 100 000 liczb i oblicz ich średnią wartość. Wynik zaokrąglają do jednego miejsca po przecinku.

Faza podsumowująca:

1. Nauczyciel wyświetla na tablicy temat lekcji i cele zawarte w sekcji „Wprowadzenie". W kontekście ich realizacji podsumowuje przebieg zajęć, a także wskazuje mocne i słabe strony pracy uczniów.
2. Wybrany uczeń podsumowuje zajęcia z programowania w C++, zwracając uwagę na nabyte umiejętności.

Praca domowa:

1. Uczniowie wykonują ćwiczenie nr 2 z sekcji „Sprawdź się". W przedstawionym programie utworzono dwuwymiarową tablicę, która zawiera losowe wektory przesunięcia. Zadaniem uczniów jest przeprowadzenie prostej symulacji, w której wyznaczą końcową pozycję cząstki przesuwanej o kolejne wektory z tablicy. Końcową pozycję wypisują jako dwie liczby oddzielone znakiem spacji, zaokrąglone do dwóch miejsc po przecinku. Symulację rozpoczynają od punktu (1.42, 2.42).

Materiały pomocnicze:

- Oficjalna dokumentacja techniczna dla języka C++.
- Oficjalna dokumentacja techniczna dla kompilatora GCC/G++ 4.5 (lub nowszej wersji).
- Oficjalna dokumentacja techniczna dla oprogramowania Code::Blocks 16.01 (lub nowszej wersji), Orwell Dev-C++ 5.11 (lub nowszej wersji) lub Microsoft Visual Studio.

Wskazówki metodyczne:

- Treści w sekcji „Przeczytaj" można wykorzystać na lekcji jako podsumowanie i utrwalenie wiedzy uczniów.