Politechnika Krakowska

Wydział Fizyki, Matematyki i Informatyki

ZADANIE 2

Zaawansowane techniki programowania

Prowadzący: dr inż. Jerzy Jaworowski

II stopień - rok 1, semestr 1

wykonanie:

Piotr Adam Tomaszewski

Nr albumu: 104896

Wstęp

Należało zaimplementować program z wykorzystaniem technologii servletów o nazwie Cube. Servlet ma otrzymywać jako dane wejściowe parametr o nazwie a, który przekazywany jest za pomocą metody POST. Parametr ten jest liczbą rzeczywistą określającą długość boku sześcianu położonego w początku układu współrzędnych. Jako kolejny parametr c przekazywana jest gęstość materiału z którego zbudowana jest kostka. W materiale występują defekty, które mają charakter przestrzeni o kształcie kulistym i gęstości materiału g. W wyniku obliczeń program ma zwrócić masę rzeczywistą analizowanego bloku materiału. Do wyznaczenia rzeczywistej masy należało wykorzystać metodę Monte Carlo.

Organizacja plików

Cube.java

Jest to jedyna klasa programu, która implementuje w sobie zadania servletu. Wykonywane są w niej metody doGet oraz doPost. Pozwalają one odebrać wartości parametrów przekazane przez użytkownika do servletu.

Metoda ***doGet*** pozwala na jawne przekazanie parametrów przez 'klienta' przy użyciu link URL. Celem tej metody jest pobranie czterech parametrów: x, y, z, r odpowiadającym współrzędnym kuli wraz z długością jego promienia.

`@param request` - zapytanie wysyłane przez przeglądarkę

`@param response` - odpowiedź wysyłana do przeglądarki

`@throws ServletException` - określenie ogólnego wyjątku, gdy servlet napotka trudności.

`@throws IOException` - ogólna klasa wywoływana przy nie udanej lub przerwanej operacji wejścia/wyjścia

```
protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
    Ball sphere = new Ball(
                    Double.parseDouble(request.getParameter("x")),
                    Double.parseDouble(request.getParameter("y")),
                    Double.parseDouble(request.getParameter("z")),
                    Double.parseDouble(request.getParameter("r"))
                );
    ballList.add(sphere);
}
```

Rysunek 1 Implementacja metody doGet

Metoda ***doPost*** która wywoływana jest w momencie przekazywania żądania POST odbiera parametr a odpowiadający długości sześcianu, c określający wartość gęstości materiału z którego sześcian został wykonany oraz g określający wartość gęstości materiału w obszarach defektów. Następnie wywoływana jest funkcja w której liczone są defekty a po jej wykonaniu przekazywana jest informacja zwrotna o masie rzeczywistej analizowanego bloku materiału.

`@param request` - zapytanie wysyłane przez przeglądarkę

`@param response` - odpowiedź wysyłana do przeglądarki

`@throws ServletException` - określenie ogólnego wyjątku, gdy servlet napotka trudności.

`@throws IOException` - ogólna klasa wywoływana przy nie udanej lub przerwanej operacji wejścia/wyjścia

```
@Override
protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

    request.setCharacterEncoding("UTF-8");
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    double a = Double.parseDouble(request.getParameter("a"));
    double c = Double.parseDouble(request.getParameter("c"));
    double g = Double.parseDouble(request.getParameter("g"));

    double defects = countDefect(0, 0, 1000000, a);
    out.format(Locale.ENGLISH, "%.5f",
            ((defects * g) + (Math.pow(a, 3) - defects) * c)
    );
    ballList.clear();
}
```

Rysunek 2 Implementacja metody doPost

Do pomocy w przechowywaniu współrzędnych punktów kół zaimplementowano listę o typie Ball. Ball jest to klasa wewnętrzna, którą celowo zaimplementowano, aby móc jak najbardziej realnie przedstawić świat rzeczywisty w języku programowania. Posiada ona pola prywatne typu rzeczywistego oraz konstruktor, dzięki któremu przypisujemy wartości do parametrów i potem możemy się względem nich odwoływać. Dodatkowo zaimplementowano gettery i setery, czyli metody pozwalające na pobieranie/ustawianie wartości atrybutu obiektu oraz metodę specjalną toString().

```
private class Ball{

    private double x;
    private double y;
    private double z;
    private double r;

    public Ball(){}

    public Ball(double x, double y, double z, double r) {
        this.x = x;
        this.y = y;
        this.z = z;
        this.r = r;
    }

    public double getX() {
        return x;
    }

    public void setX(double x) {
        this.x = x;
    }
```

Rysunek 3 Implementacja klasy wewnętrznej Ball

Metoda ***countDefect*** przechodząca po każdym elemencie listy zawierającej punkty i promień koła mieszczącego się w środku, pomiędzy lub na zewnątrz sześcianu. Sprawdzane jest to w tej funkcji. Jeśli koło znajduje się w środku sześcianu wywoływana jest metoda licząca objętość koła znanym matematycznym wzorem. Jeśli nie, to do obliczenia objętości koła w sześcianie wykorzystywana jest metoda Monte Carlo.

`@param defects` - przekazywana wartość początkowa defektów równa 0

`@param sum` - przekazywana początkowa wartość sumy równa 0

`@param n` - ilość prób

`@param a` - długość boku sześcianu

`@return` wartość defektów które wystąpiły w sześcianie

```
private double countDefect(double defects, int sum, int n, double a) {
    for (Ball ball : ballList){
        if ((ball.x + ball.r)<a && (ball.x - ball.r)>0
            && (ball.y + ball.r)<a && (ball.y - ball.r)>0
            && (ball.z + ball.r)<a && (ball.z - ball.r)>0){
            defects += volumeSphere(ball.r);
        } else {
            sum = randPointInBall(a,n,ball.r,ball.x ,ball.y,ball.z,sum,0);
            defects+=defectCount(sum,n,ball.x,ball.y,ball.z,ball.r);
        }
        sum=0;
    }
    return defects;
}
```

Rysunek 4 Implementacja metody countDefect

Metoda ***volumeSphere*** wyznaczająca objętość koła znajdującego się wewnątrz sześcianu.

`@param r` - promień koła

`@return` objętość koła

```
private double volumeSphere(double r){
    double vol = (double)4/3*Math.PI*Math.pow(r, 3);
    return vol;
}
```

Rysunek 5 Implementacja metody volumeSphere

Metoda ***randPointBall*** polega na wylosowaniu n punktów znajdujących się w obrębie kuli – czyli tak jakbyśmy narysowali mniejszy sześcian na kole i w jego obrębie losowali punkty. Po wygenerowaniu punktu wywoływana jest funkcja sprawdzająca czy wylosowany wcześniej punkt znajduje się w kuli czy poza nią.

`@param a` - długość boku sześcianu

`@param n` - ilość punktów do wylosowania

`@param r` - długość promienia koła

`@param x` - współrzędna x koła

`@param y` - współrzędna y koła

`@param z` - współrzędna z koła

`@param sum` - początkowa wartość ilości punktów w kole

`@param i` - iterator pętli

`@return` zliczona ilość losowych punktów znajdujących się w kole

```
private int randPointInBall(
    double a, int n, double r, double x,
    double y, double z, int sum, int i
) {
    double xx;
    double yy;
    double zz;
    for (i = 0; i <= n; i++){
            xx = rand(x - r, x + r);
            yy = rand(y - r, y + r);
            zz = rand(z - r, z + r);
            sum=checkPointInCube(a,xx,yy,zz,r,x,y,z,sum);
        }
    return sum;
}
```

Rysunek 6 Implementacja metody randPointBall

Metoda ***rand*** losująca wartości z przedziału od najbardziej skrajnego punktu promienia do drugiego skrajnego punktu promienia koła

`@param left` - współrzędna punktu po lewej stronie

`@param right` - współrzędna punktu po prawej stronie

`@return` wartość wylosowana

```
private double rand(double left, double right) {
    double randd = Math.random();
    return randd*(right-left)+left;
}
```

Rysunek 7 Implementacja metody rand

Metoda ***checkPointInCube*** sprawdzająca na początku czy koło w jakimś stopniu nachodzi na sześcian. Jeśli nie to dalsze kroki są pomijane bo nie ma sensu liczyć objętości poza sześcianem. Jeśli jednak koło nachodzi na sześcian to jest sprawdzana własność, czy wylosowany punkt znajduje się w kole. Jeśli tak to inkrementowana jest wartość zmiennej sum, która potem jest zwracana.

`@param a` - długość boku sześcianu

`@param xx` - wylosowana współrzędna x

`@param yy` - wylosowana współrzędna y

`@param zz` - wylosowana współrzędna z

`@param r` - długość promienia koła

`@param x` - współrzędna x koła

`@param y` - współrzędna y koła

`@param z` - współrzędna z koła

`@param sum` - dotychczasowa ilość punktów w kole

`@return` nowa ilość punktów w kole

```
private int checkPointInCube (
        double a, double xx, double yy, double zz, double r,
        double x, double y, double z, int sum
){
    if (((xx>=0)&&(a>=xx))&&((yy>=0)&&(a>=yy))&&((zz>=0)&&(a>=zz))){
        x=x-xx;
        y=y-yy;
        z=z-zz;
        if ((Math.sqrt((x*x) + (y*y) + (z*z)))<=r){
            sum++;
        }
    }
     return sum;
}
```

Rysunek 8 Implementacja metody checkPointInCube

Metoda ***defectCount*** wyznacza nam przybliżoną wartość całki w metodzie Monte Carlo, mnożąc uzyskaną wartość średnią - sumy punktów podzieloną przez ilość prób - przez długość całkowanego przedziału. Długość przedziału liczona jest ze standardowego wzoru matematycznego długości odcinka między dwoma punktami.

`@param sum` - ilość punktów znajdujących się w kole

`@param n` - ilość prób w których sprawdzano położenie wylosowanego punktu

`@param x` - współrzędna x kuli

`@param y` - współrzędna y kuli

`@param z` - współrzędna z kuli

`@param r` - promień kuli

`@return` objętość części kuli znajdującej się w sześcianie

```
private double defectCount(
        int sum, int n, double x, double y, double z, double r
){
    return ((double)sum/n)*Math.abs((x+r)-(x-r))
            *Math.abs((y+r)-(y-r))
            *Math.abs((z+r)-(z-r));
}
```

Rysunek 9 Implementacja metody defectCount