Falscher Dateiname -0,5

ADF 2*x* & PRO 2*x* — Übungen zu Fortgeschrittenen Algorithmen & Datenstrukturen und OOP — **SS 18, Übung 1**

Abgabetermin: Mi in der KW 12

☒ **Gr. 1,** Dr. G. Kronberger — **Name** PAPESH Konstantin — **Aufwand in h** 8

☐ **Gr. 2,** Dr. H. Gruber

☐ **Gr. 3,** Dr. D. Auer — **Punkte** 19,5 — **Kurzzeichen Tutor / Übungsleiter** L.S.l

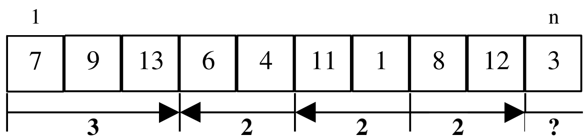
1. Läufe-Test (8 Punkte)

Gegeben sei ein Zufallszahlengenerator, der ganzzahlige Zufallszahlen erzeugt. Gesucht ist eine

```
PROCEDURE ComputeRuns(      n: INTEGER;
                      VAR maxAsc, maxDesc: INTEGER;
                      VAR    asc,    desc: IntArray);
```

die untersucht, wieviele aufsteigende und absteigende Teilfolgen, sogenannte Läufe (engl. *runs*), in einer von dem gegebenen Zufallszahlengenerator erzeugten Zufallszahlenfolge der Länge *n* enthalten sind. Die Ergebnisse *maxAsc* und *maxDesc* müssen die maximalen Längen der auf- bzw. absteigenden Läufe liefern. Das Ergebnisfeld *asc* muss an der Stelle *i* (für 1 <= *i* <= *maxAsc*) die Anzahl der aufsteigenden Läufe der Länge *i*, das Ergebnisfeld *desc* muss an der Stelle *i* (für 1 <= *i* <= *maxDesc*) die Anzahl der absteigenden Läufe der Länge *i* enthalten. Ein abschließender Lauf der Länge 1 soll nicht gezählt werden, da hier nicht entschieden werden kann, ob er auf- oder absteigend wäre.

Beispiel: Für die hier dargestellte Zufallszahlenfolge der Länge *n* = 10 (mit darunter dargestellten Läufen und ihren Längen)



muss *ComputeRuns maxAsc* = 3 mit *asc*[1] = 0, *asc*[2] = 1 und *asc*[3] = 1 sowie *maxDesc* = 2 mit *desc*[1] = 0 und *desc*[2] = 2 liefern.

2. Monte-Carlo-Methode (12 + 2 + 2 Punkte)

Das Volumen eines durch die (*x*, *y*, *z*)-Koordinaten der beiden Brennpunkte *f1* und *f2* gegebenen Rotationsellipsoids kann wie folgt näherungsweise berechnet werden: Man generiert zufällige, möglichst gleichverteilte Punkte in einem quaderförmig den Rotationsellipsoid umgebenden Teil des Raums. Für jeden Punkt prüft man, ob dieser auch innerhalb des Rotationsellipsoids liegt („Treffer“) oder nicht. (Dabei gilt: Summe der Entfernungen jedes Punkts innerhalb des Rotationsellipsoids von den beiden Brennpunkten ist kleiner oder gleich der ebenfalls gegebenen Größe *distSum*). Aus dem Verhältnis der Treffer zur Anzahl der generierten Punkte kann man dann einen Näherungswert für das Volumen des Rotationsellipsoids berechnen.

a) Implementieren Sie eine

```
FUNCTION Volume(f1, f2: Point; distSum: REAL): REAL;
```

die das Volumen des Rotationsellipsoids näherungsweise berechnet. Geben Sie die Näherungswerte für 10, 100, 1000, … generierte Punkte an.

b) Stellen Sie experimentell fest, wieviele Punkte man braucht, sodass sich der Näherungswert zum ersten Mal um weniger als 0,001 vom exakten Wert unterscheidet.

c) Lassen Sie dieselben Programme nun mit einem selbst geschriebenen Zufallszahlen-Generator laufen und kommentieren Sie eventuelle Unterschiede in den Ergebnissen.

ADF2x & PRO2X Algorithmen & Datenstrukturen und OOP – SS 2018
Übungsabgabe 1

Konstantin Papesh

21. März 2018

Zusammenfassung

In dieser Übung wird die Programmierung und Analyse eines Zufallszahlengenerators behandelt und deren beispielhafte Einsatzweise beschrieben.

1.1 Läufe-Test

Ein Zufallszahlengenerator ist gegeben, mit diesem ist eine Zahlenfolge zu generieren und diese soll anschließend analysiert werden. Die generierten Zahlen werde miteinander verglichen, um somit Läufe in der Zahlenfolge feststellen zu können. Dabei wird jeweils protokolliert, ob der Lauf steigend oder fallend war und wie lange er angedauert hat.

Das ist die Angabe, keine Lösungsidee ~7

1.1.1 Implementierung

Listing 1.1: statistics.pas

```
program statistics;
uses ModLCGRandom;

(*interface
procedure ComputeRuns( n : integer;
                       var maxAsc, maxDesc : integer;
                       var asc, desc : intArray);*)

type
    intArray = array[integer] of 1..20;

var h : integer;   schlechter Name - 0,5
    n : integer;
    hOld : integer;   seltsamer Name
    curRun : integer;
    i : integer;
    asc, desc : intArray;
    maxAsc, maxDesc : integer;

```

```
procedure ComputeRuns(n : integer;
                      var maxAsc, maxDesc : integer;
                      var asc, desc : intArray);
begin
    h := 1337; (*Seed value*)
    i := 0;
    hOld := h;
    curRun := 0;
    repeat
        initLCG(h);
        h := RandInt;
        if i = 0 then
            hOld := h; (*So first run isn't counted*)
        if h > hOld then(*Means run is now ascending*)
        begin
            if (curRun >= 0) then(*current run is ascending or just started*)
            begin
                inc(curRun);
            end
            else if (curRun < 0) then(*current run is descending*)
            begin
                dec(curRun); (*Because else the run will always display one short*)
                if (-curRun > maxDesc) then
                begin
                    maxDesc := -curRun;
                end;
                inc(desc[-curRun]);
                curRun := 0;
            end;
        end
        else if h < hOld then(*means run is now descending*)
        begin
            if curRun <= 0 then(*current run is descending or just started*)
            begin
                dec(curRun);
            end
            else if curRun > 0 then(*current run is ascending*)
            begin
                inc(curRun);  (*because else the run will always display one short*)
                if curRun > maxAsc then (*new record asc!*)
                begin
                    maxAsc := curRun;
                end;
                inc(asc[curRun]);
                curRun := 0;
            end;
        end
        else (*current number is same as the one before*)
        begin
            if curRun > 0 then(*current run is ascending*)
            begin
                inc(curRun);  (*Because else the run will always display one short*)
                if curRun > maxAsc then (*new record asc!*)
                begin
                    maxAsc := curRun;
                end;
                inc(asc[curRun]);
```

```
                curRun := 0;
            end
            else if curRun < 0 then(*current run is descending*)
            begin
                dec(curRun);  (*Because else the run will always display one short*)
                if -curRun > maxDesc then
                begin
                    maxDesc := -curRun;
                end;
                inc(desc[-curRun]);
                curRun := 0;
            end;
        end;
        inc(i);
        hOld := h;
    until i = n;
    writeLn('### Statistics ###');
    writeLn('Maximum Ascend:', maxAsc);
    writeLn('Maximum Descend:', maxDesc);
    writeLn('Maximum Ascend happened:', asc[maxAsc]);
    writeLn('Maximum Descend happened:', desc[maxDesc]);
end;
begin
    n := 20;
    ComputeRuns(n, maxAsc, maxDesc, asc, desc);
end.
```

Listing 1.2: ModLCGRandom.pas - Für das Erstellen der Zufallszahlen zuständig.

```
UNIT ModLCGReal;

INTERFACE
FUNCTION randInt() : integer;
FUNCTION randReal() : real;
PROCEDURE initLCG(randSeed : integer);

IMPLEMENTATION
USES math;
CONST
    a = 48721;
    c = 1;
    m = 32768; (*2 ^ 16*)
```

schlechte Namen

```
VAR
    x: integer;

FUNCTION randInt : INTEGER;
BEGIN
    x := (a*x+c)MOD m;
    RandInt := x;
END;

FUNCTION randReal : real;
BEGIN
    randReal := randInt/m;
END;

PROCEDURE initLCG(randSeed : integer);
```

```
BEGIN
    x:= randSeed;
END;

BEGIN
END.
```

1.1.2 Ausgabe

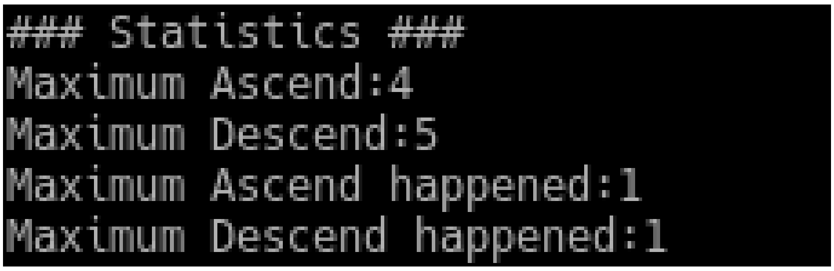


Abbildung 1.1: Ausgabe in der Konsole

Felder asc und desc -1

1.2 Monte-Carlo-Methode

Es ist ein Rotationsellipsoid definiert mithilfe zweier Punkte, *f1* und *f2*, und *distSum*. Dieser Wert ist die Summe der Entfernungen von der Hülle zu den zwei Punkten. Bei der Monte-Carlo-Methode wird nun ein zufälliger Punkt generiert und von diesem die Distanz zu den Punkten *f1* und *f2* berechnet und zusammengezählt. Ist diese Summe kleiner als *distSum*, befindet sich der Punkt innerhalb des Ellipsiods, sonst ausserhalb.

1.2.1 Implementierung

Listing 1.3: montecarlo.pas

```
program montecarlo;

type
    point = array[0..2] of real;
var
  f1, f2 : point;
    distSum: real;
    n : longint;

procedure centerEllipse(var f1, f2 : point); (* Moves the f1 and f2 point so that
     they are connected via the x-axis *)
var
    distance : real;
begin
    distance := sqrt(sqr(f1[0]-f2[0])+sqr(f1[1]-f2[1])+sqr(f1[2]-f2[2])); (*Distance
      between f1 and f2 *)
    f1[0] := -(distance/2);
    f1[1] := 0;
    f1[2] := 0;
    f2[0] := (distance/2);
```

Pythagoras-Funktion (fehlende funktionale Zerlegung) -0,5

```
    f2[1] := 0;
    f2[2] := 0;
end;

procedure boundingBoxCalc(f1, f2 : point;
                          distSum : real;
                          var boundingBoxXLength, boundingBoxYLength, boundingBoxZLength : real;
                          var boundingBoxVol : real); (*Calculates the bounding box *)
begin
    boundingBoxXLength := distSum;
    boundingBoxYLength := boundingBoxXLength;
    boundingBoxZLength := boundingBoxYLength;
    boundingBoxVol := boundingBoxXLength * boundingBoxYLength * boundingBoxZLength;
end;

function Volume(f1, f2 : point;
                distSum: real) : real;
var
    countInEllipse : longint;
  i : longint;
    x, y, z : real;
    boundingBoxXLength : real;
    boundingBoxYLength : real;
    boundingBoxZLength : real;
    boundingBoxVol : real;
begin
  countInEllipse := 0;
    centerEllipse(f1, f2);
    boundingBoxCalc(f1, f2, distSum, boundingBoxXLength, boundingBoxYLength, boundingBoxZLength, boundingBoxVol);
  (* Using Monte Carlo Method *)
  for i := 1 to n do begin
    x := random()*boundingBoxXLength-boundingBoxXLength/2;
    y := random()*boundingBoxYLength-boundingBoxYLength/2;
    z := random()*boundingBoxZLength-boundingBoxZLength/2;
    if((sqrt(sqr(f1[0]-x)+sqr(f1[1]-y)+sqr(f1[2]-z))+sqrt(sqr(f2[0]-x)+sqr(f2[1]-y)+sqr(f2[2]-z)))<=distSum) then
            inc(countInEllipse);
  end;
  Volume := boundingBoxVol * (countInEllipse/n);
end;
begin
  distSum := 2.0;
  f1[0] := 1.2;
  f1[1] := 0.8;
  f1[2] := 0.3;
  f2[0] := -0.5;
  f2[1] := -1.0;
    f2[2] := 1.0;
    n := 10;
    writeln('Volume for n = ', n, ' is', Volume(f1, f2, distSum));
    n := 100;
    writeln('Volume for n = ', n, ' is', Volume(f1, f2, distSum));
    n := 1000;
    writeln('Volume for n = ', n, ' is', Volume(f1, f2, distSum));
```

Code-Verdoppelung, zu langsam

?! − 0,5

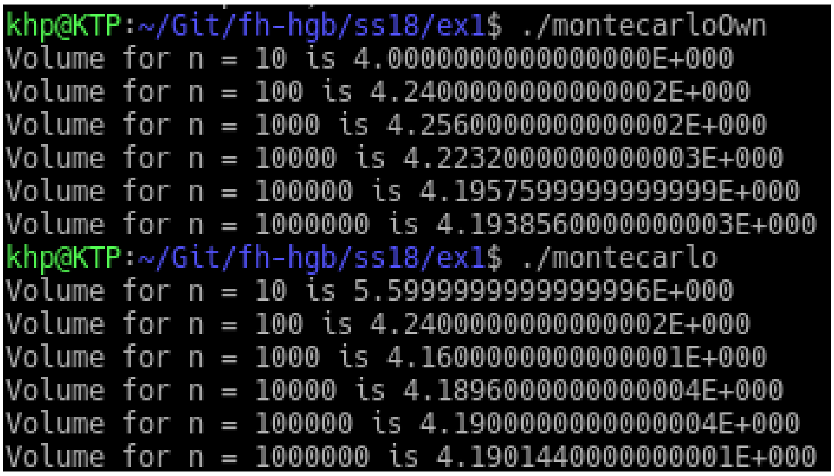
```
    n := 10000;
    writeln('Volume for n = ', n, ' is', Volume(f1, f2, distSum));
    n := 100000;
    writeln('Volume for n = ', n, ' is', Volume(f1, f2, distSum));
    n := 1000000;
    writeln('Volume for n = ', n, ' is', Volume(f1, f2, distSum));
end.
```

Um das Programm mithilfe eines selbstprogrammierten Zufallszahlengenerator betreiben zu können muss sowohl die Funktion *Volume* etwas abgeändert werden, als auch im Header die Benützung von ModLCGRandom1.1.1 definiert werden.

Listing 1.4: montecarloOwn.pas

```
function Volume(f1, f2 : point;
                distSum: real) : real;
var
    countInEllipse : longint;
  i : longint;
    x, y, z : real;
    boundingBoxXLength : real;
    boundingBoxYLength : real;
    boundingBoxZLength : real;
    boundingBoxVol : real;
    initialSeed : integer;
begin
    countInEllipse := 0;
    initialSeed := 5433;
    centerEllipse(f1, f2);
    boundingBoxCalc(f1, f2, distSum, boundingBoxXLength, boundingBoxYLength,
    boundingBoxZLength, boundingBoxVol);
    (* Using Monte Carlo Method *)
    initLCG(initialSeed);
  for i := 1 to n do begin
    x := randReal*boundingBoxXLength-boundingBoxXLength/2;
    y := randReal*boundingBoxYLength-boundingBoxYLength/2;
    z := randReal*boundingBoxZLength-boundingBoxZLength/2;
    if((sqrt(sqr(f1[0]-x)+sqr(f1[1]-y)+sqr(f1[2]-z))+sqrt(sqr(f2[0]-x)+sqr(f2[1]-y)+
    sqr(f2[2]-z)))<=distSum) then
            inc(countInEllipse);
  end;
  Volume := boundingBoxVol * (countInEllipse/n);
end;
```

1.2.2 Ausgabe



```
khp@KTP:~/Git/fh-hgb/ss18/ex1$ ./montecarloOwn
Volume for n = 10 is 4.0000000000000000E+000
Volume for n = 100 is 4.2400000000000002E+000
Volume for n = 1000 is 4.2560000000000002E+000
Volume for n = 10000 is 4.2232000000000003E+000
Volume for n = 100000 is 4.1957599999999999E+000
Volume for n = 1000000 is 4.1938560000000003E+000
khp@KTP:~/Git/fh-hgb/ss18/ex1$ ./montecarlo
Volume for n = 10 is 5.5999999999999996E+000
Volume for n = 100 is 4.2400000000000002E+000
Volume for n = 1000 is 4.1600000000000001E+000
Volume for n = 10000 is 4.1896000000000004E+000
Volume for n = 100000 is 4.1900000000000004E+000
Volume for n = 1000000 is 4.1901440000000001E+000
```

Abbildung 1.2: Ausgabe in der Konsole

1.2.3 Auswertung

Das ist die Lösungs-idee!!!

a) Um die Berechnung zu vereinfachen wird zuerst das gegebene Rotationsellipsoid auf die x-Achse zentriert, dies macht die Berechnung der Bounding Box einfacher, in welcher der zufällige Wert liegen wird. Nachdem das Ellipsiod zentriert ist, generieren wir eine Box, welche den Ellipsiod beinhaltet und in welcher alle zufällig generierten Werte liegen müssen. Auch berechnen wir das Volumen ebendieser Box. Anschließend werden Zufallszahlen generiert, welche dann Zufallspunkte bilden und die Distanzen dieser zu den Punkten *f1* und *f2* werden zusammengezählt und mit *distSum* verglichen. Falls diese Summe kleiner ist, wird ein Zähler erhöht. Nachdem alle Zufallspunkte generiert und verglichen wurden wird die Summe aller Treffer mit der Summe aller generierten Punkte dividiert und mit dem Volumen der Box multipliziert. Somit erhält man das ungefähre Volumen des Ellipsiods.

b) Durch empirisches Testen wurde festgestellt, dass ab ca. 1,000,000 Zufallspunkte eine Genauigkeit von 0.001 vorliegt.

c) Beim Vergleich zwischen eines selbst geschriebenen- und eines Standart-Zufallszahlen-Generators konnten keine wesentlichen Unterschiede festgestellt werden. Nur schwankt der Volumenwert bei hoher Punktedichte beim selbst geschriebenen mehr als wie beim Pascal-Zahlengenerator. Um wie viel? −0,5