Self Avoiding Walks

Arthur van Dam
Gerben Wolterink

26 februari 1999

Samenvatting

Dit verslag geeft een beschrijving van het computerpracticum bij *Modelvorming en Simulatie*. Het volgende probleem wordt hierin behandeld:

We beschouwen een *self-avoiding walk* (SAW); een 'walk' waarvan de lijnsegmenten elkaar niet doorsnijden. In de praktijk kunnen dit bijvoorbeeld polymeerketens zijn. De vraag is, hoe (het kwadraat van) de afstand tussen begin- en eindpunt afhangt van het aantal segmenten (stappen).

1 Inleiding

Het is bekend dat de verwachtingswaarde van het kwadraat van de afstand tussen beginpunt en eindpunt van een polymeer voldoet aan de vergelijking van de vorm $\langle r^2 \rangle \sim N^{2\nu}$. Hierbij is N het aantal segmenten waaruit het polymeer bestaat. De formule is alleen geldig voor grote N. Onze vraag is nu wat de waarde van ν is in dimensie 2 en 3. Met behulp van een computermodel wordt geprobeerd om deze waarde zo goed mogelijk te bepalen.

2 Computersimulatie

Er zijn diverse manieren om dit probleem te modelleren. Zo zijn er de methoden *exacte enumeratie*, *Monte Carlo* en *incomplete enumeratie*. Er is voor deze laatste aanpak gekozen. Deze methode op een naïeve manier implementeren kan bijvoorbeeld als volgt:

- groei een polymeer van lengte N en bereken r^2
- herhaal dit m keer
- bepaal $\langle r^2 \rangle = \frac{1}{m} \sum_s r_s^2$

Deze implementatie is echter fout. De reden is dat sommige configuraties een grotere kans hebben om gegenereerd te worden dan andere. Vooral de compacte configuraties hebben een grotere kans. Deze methode geeft geen goede verwachtingswaarde, aangezien ons uitgangspunt hier is dat elk polymeer gelijke kans van bestaan heeft. Een goede implementatie van *incomplete enumeratie* heeft hier geen last van. Dit kan als volgt:

- groei een polymeer met lengte N
  - kijk bij elke toevoeging n in hoeveel richtingen het polymeer kan groeien: m_n het polymeer mag zichzelf hierbij niet doorsnijden

  - bij polymeer met lengte n wordt het rosenbluthgewicht $w_{n+1} = w_n \cdot m_n$

- herhaal dit m keer
- bepaal $\langle r^2 \rangle = \frac{\sum_m w_m r_m^2}{\sum_m w_m}$

De kans op een bepaalde configuratie C is gelijk aan $P_c = \prod_{i=0}^{N-1} \frac{1}{m_n}$. Het gewicht van deze configuratie is $\prod_{i=0}^{N-1} m_n$. Dit geeft aan het gewogen ensemble een bijdrage van 1. Zo tellen alle configuraties even zwaar. Op deze manier ontstaan echter nog wel grote statistische fluctuaties. Dit vanwege het feit dat een polynoom met een groot rosenbluth gewicht kan domineren bij bepaling van $\langle r^2 \rangle$. Wanneer in één simulatie wel een groot rosenbluth gewicht bij zit en bij een andere niet, kunnen de resultaten nogal verschillen.

Dit probleem kan verholpen worden door pruning en enrichment toe te passen. Pruning houdt in dat bij een te laag gewicht (t.o.v. gemiddelde) het gewicht van het polymeer met 50% kans wordt verdubbeld of op nul wordt gezet. Het op nul zetten van het gewicht staat gelijk aan het stoppen met groeien van de polymeer.

Enrichment houdt in dat bij een te groot gewicht (t.o.v. gemiddelde) er 2 copiën van het polymeer worden gemaakt met elk het halve gewicht. Dit alles zorgt ervoor dat de statistische fluctuaties een stuk kleiner worden.

Om te kunnen schatten wat de fout (ongeveer) is in het verkregen antwoordt voor ν, wordt de standaarddeviatie gebruikt. Wanneer er m polymeren gegroeid moeten worden, wordt dit bijgehouden in S sets van M polymeren. Dan wordt het volgende gedefinieerd:

$$\sigma^2(\nu) = \langle \nu^2 \rangle - \langle \nu \rangle^2, \qquad \langle \nu \rangle = \frac{1}{S} \sum_{i=1}^{S} \nu_i, \qquad \langle \nu^2 \rangle = \frac{1}{S} \sum_{i=1}^{S} \nu_i^2. \tag{1}$$

De fout in ν na S sets van M polymeren definiëren we als: fout $= \sigma(\nu)/S$. Bij elk blok wordt na M polymeren gegroeid te hebben de *gewogen* waarde van ν bepaald volgens formule (2).

$$\langle r^2 \rangle = \frac{\sum_s w_s r_s^2}{\sum_s w_s} \sim N^{2\nu} \Longrightarrow \nu = \frac{\log\left(\frac{\sum_s w_s r_s^2}{\sum_s w_s}\right)}{2 \log N} \tag{2}$$

Deze S waarden van ν (elk bepaald door M polymeren) kunnen dus gebruikt worden bij het schatten van de statistische fout op de waarde van ν bepaald door alle $(M \cdot S)$ gegroeide polymeren. De interpretatie van $\sigma(\nu)$ is dat de ν bepaald door $M \cdot S$ polymeren met een kans van 68% binnen $\sigma(\nu)$ van de echte waarde van ν ligt.

3 Implementatie

De computersimulatie hebben wij geïmplementeerd in een *C++*-programma. De listings van het programma voor drie dimensies staan in Bijlage A. De programma's voor 2-D en 3-D lijken natuurlijk erg veel op elkaar. Aangezien in het programma voor drie dimensies nog wat extra trucs zijn gebruikt, wordt alleen dit programma besproken.

Het programma is voor de duidelijkheid opgesplitst in twee delen: Het starten en verwerken van de simulatie (`SAW3.cpp`) en het groeien van een nieuw 'takje' aan een polymeer (`grow3.cpp`).

3.1 SAW3.cpp

Het commentaar in de listing zal veel duidelijkheid geven. Enkele trucs worden hier nog even toegelicht: De `lattice_size` wordt gezet op `2*gridsize + 1`. Eén roosterpunt voor het midden en 2 maal `gridsize` voor links èn rechts van het midden (dit alles in drie richtingen). De `gridsize` (= afstand van midden van `lattice` tot rand) wordt slechts op $0.2 \cdot$ polymeerlengte gezet. In 3 dimensies kan een polymeer namelijk zoveel kanten op dat er met een zeer kleine kans hoofdzakelijk 'rechtuit' gegroeid wordt. Een `lattice`, volledig passend voor volledige polymeerlengte zou geheugenverspilling zijn. In 3-D bleek de factor 0.2 nog steed vrij weinig 'botsingen' met de wand te veroorzaken.

Na alle initialisaties wordt een lus gestart voor het groeien van alle (`M*S`) polymeren. De eerste stap krijgt een *Rosenbluthgewicht* van 0.25, omdat we $1/4 \cdot w$ bijhouden. Dit scheelt over het hele polymeer een factor $(0.25)^N$. Op deze manier wordt het *Rosenbluthgewicht* niet groter dan de toegestane grootte van een Floating-point-getal. Is een polymeer eenmaal 'klaar' dan wordt het belopen pad in `lattice` weer gewist zodat er geen invloed is op de volgende polymeren. `x`, `y` en `z` worden niet gewist; de volgende polymeren overschrijven de inhoud gewoon. Na afloop wordt de waarde van ν behaald uit (2) en de gegevens van alle polymeren en uit de `deelnu`'s wordt een foutschatting volgens (1) voor deze waarde bepaald.

3.2 grow3.cpp

Aangezien het `lattice` in principe te klein is voor de polymeren, wordt eerst gecontroleerd of de grenzen niet worden overschreden. Zo ja, return dan de lengte. In `main` wordt het pad gewist en een nieuw polymeer wordt gestart. Het voortijdig gestopte polymeer wordt niet geregistreerd.

Hierna vindt de *Pruning*-check plaats: Als het huidige *Rosenbluthgewicht* `w` kleiner is dan de (*ondergrens*) $\cdot$ (*het gemiddelde gewicht bij deze lengte tot nu toe*), dan wordt met kans 0.5 de polymeer voortgezet met dubbel gewicht of afgebroken. Een dergelijke check vindt ook plaats voor *Enrichment*: Het gewicht wordt gehalveerd en het polymeer verder gegroeid. Het extra stuk wat vanaf dan is gegroeid, wordt na afloop gewist in `lattice` en er wordt nog een 'staart' aan het beginstuk gegroeid (ook met half gewicht). Na deze checks is zeker dat het gewicht binnen de toegestane grenzen ligt, en wordt het dus meegenomen in de waarde van het gemiddelde gewicht. Vervolgens wordt gecontroleerd of de polymeer al lengte `MPL` heeft. Zo ja, dan worden de gewenste grootheden bijgewerkt. Als bovendien ook nog blijkt dat dit de laatste polymeer in een set was (`m%M==0`) dan wordt de `deelnu` over die set bepaald. Hierbij gebruiken we (2); de bovenste sommatie wordt in de bewuste set gegeven door `sumr2w-oldsumr2w` en de onderste door `sumw-oldsumw`. De lengte N is hier gelijk aan `MPL`.

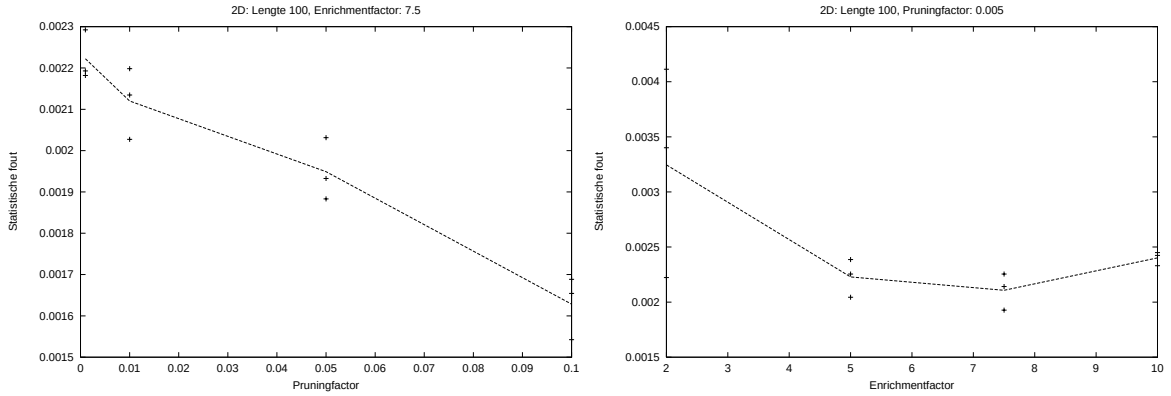
Indien de polymeer nog niet af was, wordt gekeken in welke richtingen een nieuwe stap gezet kan worden. De coordinaten worden opgeslagen in `nbx`, `nby` en `nbz` en het aantal in `numnb`. Als er geen buren meer zijn (`numnb==0`), wordt de lengte geretourneerd, anders wordt een random getal tussen 0 en `numnb` gekozen en de bijbehorende stap wordt gezet. Het gewicht van het nieuwe polymeer wordt met $1/4 \cdot$`numnb` vermenigvuldigd en de stap wordt 'gemarkeerd' in `x`, `y`, `z` en `lattice`. Een recursieve aanroep zorgt voor het doorgroeien van de polymeer.

4 Experimenten

4.1 Parameters

Om een bepaald tendens te ontdekken in de waarde van ν hebben we voor allerlei lengtes polymeren gegroeid. Het aantal gegroeide polymeren hebben we zoveel mogelijk constant gehouden: 500 sets van 500 polymeren. In 3-D bij grote lengtes wordt het aantal sets soms wat groter, omdat er meer enrichment plaats vindt. Wanneer een polymeer al vele malen is gesplitst in twee 'koppen' dan worden eerst alle koppen afgewerkt, voordat de *loop* in `main` de controle terug krijgt. Ondertussen kan het aantal gegroeide polymeren `m` al groter zijn dan `M*S`. Dit is echter niet erg bezwaarlijk, aangezien de foutschatting alleen maar lager wordt.

Tot slot moeten we ook nog criteria stellen voor het uitvoeren van pruning en enrichment. We willen dus een kleine fout krijgen. We zullen echter ook rekening moeten houden met de looptijd van de simulatie. Kiezen we bijvoorbeeld een vrij hoog pruning-criterium, dan wordt er vaak *geprun'ed* en in 50% van die gevallen wordt een polymeer afgebroken. Hierdoor kan de simulatie erg lang gaan duren. We hebben verschillende pruning- en enrichment-factoren uitgetest in korte simulaties en de fout bestudeerd. In figuur 1 staat de fout tegen de pruning- of enrichment-factoren uitgezet bij polymeerlengte 100 in 2-D. Voor de overige lengtes hebben we vergelijkbare plaatjes gemaakt, maar het lijkt ons overbodig deze hier te tonen. Bij constante enrichment-factor is te zien dat een verhoging van de pruning-factor



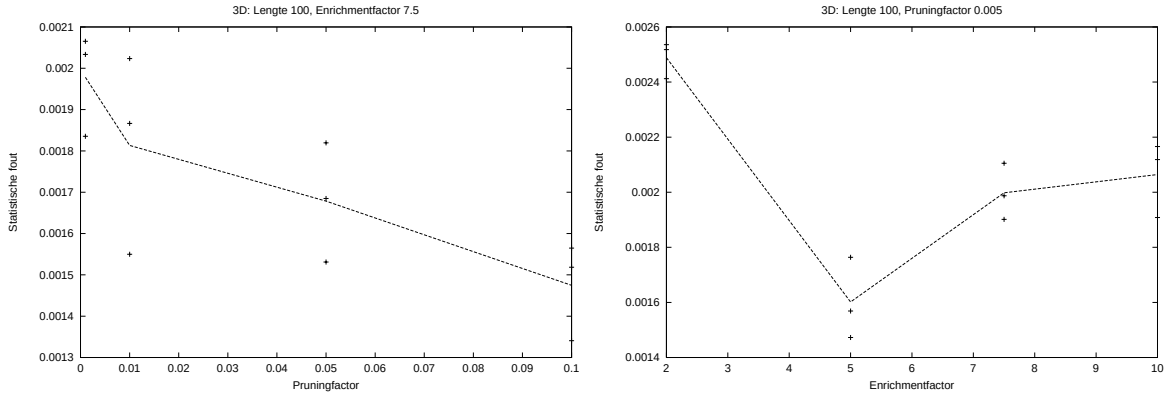
Figuur 1: Invloed van pruning- en enrichment-factoren op de fout in 2D

een kleinere fout oplevert. Dit is logisch, want indien een gewicht teveel van het gemiddelde afwijkt, wordt hier meteen wat aan gedaan. Om toch de tijdsduur wat binnen de perken te houden, zullen we 0.05 nemen.
Bij constante pruningfactor is een minimum in de foutschatting te zien bij enrichment-factor 7.5 Wij hebben deze waarde dan ook tijdens verdere experimenten gebruikt. In tabel 1 zijn de gebruikte factoren vermeld.

lengte N	2	5	10	25	50	100	175	250	500
pruningfactor	0.05	0.05	0.05	0.05	0.04	0.04	0.04	0.025	0.005
enrichmentfactor	5	5	5	7.5	7.5	7.5	7.5	6	5

Tabel 1: Gebruikte pruning- en enrichmentfactoren in 2-D

Deze zelfde experimenten hebben we gedaan voor 3-D. In figuur 2 staan voor lengte 100 de resultaten. Ook hier is weer een daling te zien bij toenemende pruningfactor, maar om



Figuur 2: Invloed van pruning- en enrichment-factoren op de fout in 3D

de tijd binnen de perken te houden, zullen we hiervoor 0.05 nemen (bij $N = 100$). De enrichmentfactor houden we in het minimum, op 5. In 3-D is het gewicht van polymeren groter (ze kunnen immers meer kanten op) en dus moet daar wat meer op gecorrigeerd worden (kleinere factor). Voor grotere lengten nemen we toch een grotere enrichmentfactor, omdat er anders zo vaak *ge-enrich'ed* wordt, dat het aantal sets van polymeren heel erg groot wordt. In tabel 2 zijn de gebruikte factoren vermeld.

lengte N	2	4	5	10	25	50	100	250	450	500	1000
pruningfactor	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.025	0.005	0.005	0.0005
enrichmentfactor	5	5	5	5	5	5	5	6	6	10	10

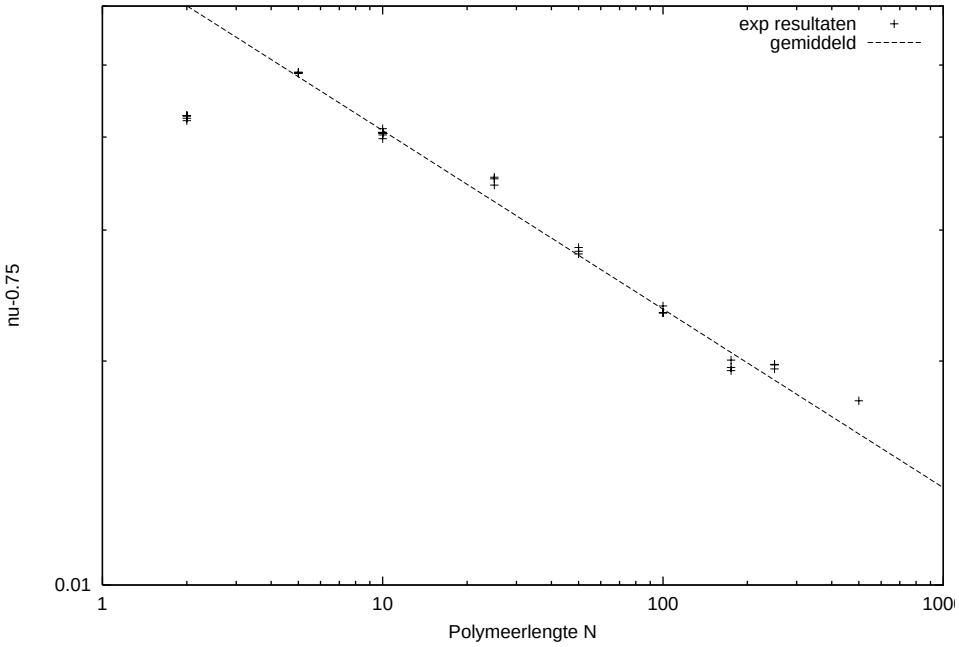
Tabel 2: Gebruikte pruning- en enrichmentfactoren in 3-D

4.2 Resultaten

Met de parameterkeuzes uit sectie 4.1 hebben we voor allerlei lengtes in 2-D en 3-D experimenten uitgevoerd.

De uitkomsten van ν in 2-D tegen de verschillende waarden van N bleek een exponentiële functie op te leveren: voor toenemende n convergeerde ν naar een vaste waarde. Bekend is dat de goede waarde voor ν 0.75 is. Daarom zijn de (*gevonden waarden* - 0.75) tegen N op een dubbele logschaal geplot. Het bleek dat er toen een rechte lijn door de punten getrokken kon worden (Zie fig.3). Bekend is dat een functie die een rechte lijn als grafiek heeft op dubbel logaritmische schaal van de vorm $y = ax^b$ is. Door twee punten op de lijn in te vullen en het verkregen stelsel op te lossen werd de volgende benadering voor ν in 2-D gevonden: $\nu = 0.068 \cdot N^{-0.23} + 0.75$. Voor $N \to \infty$ blijkt ν naar 0.75 te convergeren, hetgeen ook de exacte oplossing is.

Op eenzelfde manier is te werk gegaan bij het vinden van de waarde van ν in 3 dimensies. Hier was echter de goede oplossing niet exact bekend. Er werd geexperimenteerd met verschillende waarden die van de experimenteel gevonden waarden afgetrokken werden. Er zijn



Figuur 3: Oplossingen van nu tegen polymeerlengte N in 2-D

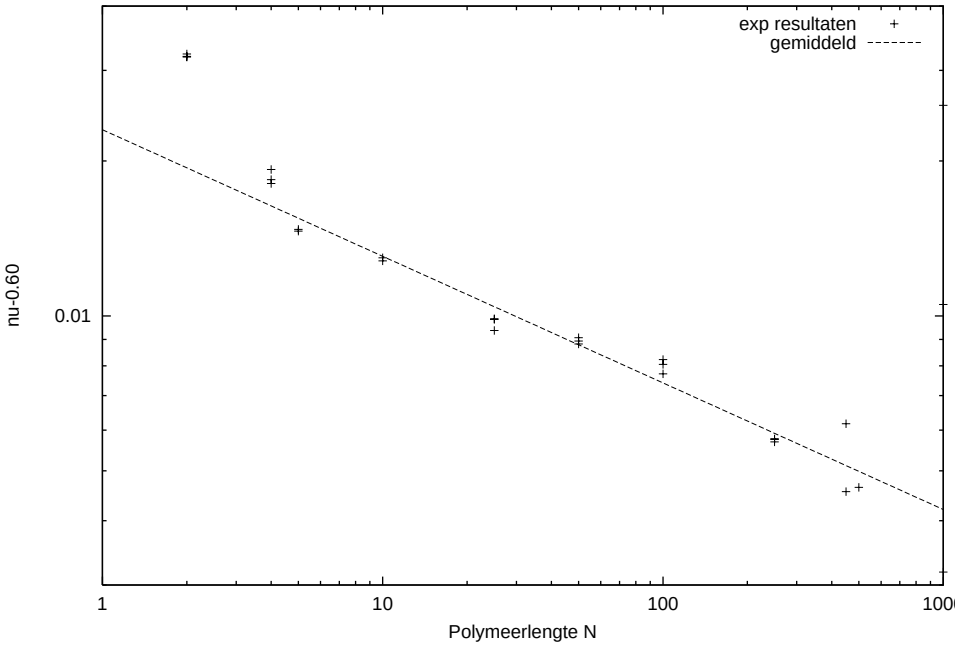
verschillende grafieken gemaakt met de waarden van x tussen 0.58 en 0.62. Bij $0.58 \leq x > 0.60$ was er ook wel een redelijke rechte lijn door de punten te trekken, maar waren de waarden (=afwijking t.o.v. x) hoger dan bij $x = 0.60$. Voor $x = 0.61$ en $x = 0.62$ bleek er steeds minder convergentie op te treden. Het correcte antwoord voor ν in 3 dimensies ligt dus in de buurt van 0.60. Mogelijk iets lager tot ± 0.59. We trekken nu 0.6 van alle waarden voor ν af en bepalen net als in 2-D een benaderende lijn met als resultaat $\nu = 0.019N^{-0.22} + 0.60$. Voor $N \to \infty$ blijkt ν dus naar 0.60 te convergeren. In de literatuur werd een waarde van 0.592 vermeld. De benaderingen zitten dus in de goede richting.

In figuur 4 zijn de resultaten, samen met de benaderende lijn weergegeven.Bij lagere waarden voor x kon er geen mooie rechte lijn meer door de punten in de grafiek getrokken worden. De gebruikte punten in de grafiek hadden een statistische fout van $O(10^{-4})$ of lager, dus daardoor kunnen er in deze gevonden antwoorden niet veel afwijkingen ontstaan zijn.
Ter illustratie is voor 3-D de output van alle experimenten in Bijlage B geplaatst. De fout neemt – over het geheel genomen – af bij toenemende lengte. Ook is bij lengte 500 te zien dat het aantal gegroeide sets sterk toeneemt, terwijl we 50 sets hadden opgegeven aan het programma. Dit komt dus doordat er vaak *ge-enrich'ed* wordt. Bij lengte 1000 voorkomen we dit door een enrichment-factor van 100.

5 Conclusie

Het verband tussen het kwadraat van de afstand van begin- tot eindpunt van een polymeer blijkt inderdaad te voldoen aan:

$$\langle r^2 \rangle \sim N^{2\nu} \tag{3}$$



Figuur 4: Oplossingen van nu tegen polymeerlengte N in 3-D

In 2-D gaven de experimenten het resultaat $\nu = 0.75$, als verwacht. In 3-D leverden de experimenten de waarde $\nu = 0.6$ op. Deze waarden zijn *asymptotische* waarden; in de limiet $N \to \infty$.

Verder is gebleken dat het *Rosenbluthgewicht* een effectieve methode is om alle polymeren 'gelijke kans' te geven. En met pruning en enrichment werden de fluctuaties beperkt.

Dit alles geeft een redelijke schatting van ν, alhoewel een preciezere analyse van de pruning- en enrichment-factoren èn de foutschatting nodig is om – vooral in 3-D – echt goede schattingen te verkrijgen.

A Listings

A.1 SAW3.cpp

```
#include <stdlib.h>
#include <time.h>
#include <iostream.h>
#include <math.h>

int *x, *y, *z,      // x-, y-, resp z-coordinaten van polymeer
    *teltak;         // Hoe vaak er al een i-de tak is aangegroeid (i.v.m. gemw bijhouden)
char ***lattice;     // geeft voor iedere site aan of 'ie bezet(1) of onbezet (0) is
int nbx[6], nby[6], nbz[6];  // voor opslag van x-, y-, resp. z-coordinaten van (max 6) buren
int MPL,             // Maximale Polymeer Lengte
    M,               // Aantal polymeren per set
    m = 0,           // teller voor hoeveel poly's al zijn gegroeid (start met 0)
    setnr = 0,       // Aantal sets (van M poly's) dat reeds is gegroeid
    xmid, ymid, zmid,// x- y- en z-coord. van midden van lattice
    pruntel=0,       // telt aantal pruningen
    enrichtel=0,     // telt aantal enrichments
    gridsize;        // Halve breedte/hoogte van het grid (vanaf middelpunt tot rand)

double *gemw;        // gemiddeld Rosenbluth gewicht bij bepaalde lengte van poly
double crit_prun,    // critische factor voor pruning (ong. 0.01)
       crit_enr,     // critische factor voor enrichment (ong. 10)
       sumw=0.0,     // som (over alle poly's) van Rosenbluthgewicht
       oldsumw=0.0,  // oude waarde van sumw (na vorige set)
       sumr2w=0.0,   // som (over alle poly's) van Rosenbluthgewicht*r^2
       oldsumr2w=0.0,// oude waarde van sumr2w (na vorige set)
       rcorrection=2.0*2.3283064365387e-10,  // voor random getal tussen 0 en 1
       somnu2 = 0.0, // som nu^2 over alle sets
       somnu = 0.0;  // som nu over alle sets


/**
 * groei een nieuw 'takje' aan polymeer
 **/
extern int grow(int,double);

void main(){
    int S,                              // Aantal sets van polymeren
        i,j,k;                          // Tellers voor loops
    double poly_len;

    srandom(time(NULL));                // Init random-number-generator
  /**
   * Input user-variables
   **/
    cin  >> MPL;       // Maximale lengte van het polymeer
    cin  >> S;         // Aantal sets van polymeren
    cin  >> M;         // Aantal polymeren per set
    cin  >> crit_prun; // Kritische waarde pruning (ondergrens factor maal gemiddelde w)
    cin  >> crit_enr;  // Kritische waarde enrichment (bovengrens factor maal gemiddelde w)

  /**
   * Overige var's initialiseren
   **/
    gridsize = (int)(MPL*0.2);          // Kleiner dan polymeerlengte, want
                                        // poly groeit toch niet 1 recht stuk
    x      = new int[MPL+1];
    y      = new int[MPL+1];
    z      = new int[MPL+1];
    teltak = new int[MPL+1];
    gemw   = new double[MPL+1];
    int lattice_size = 2*gridsize+1;    // middenpunt + 2 * lengte in 1 richting
```

```
    lattice = new char** [lattice_size];    // array van pointers naar 'rijen' van lattice
    for(i = 0; i < lattice_size; i++){      // rijen van het lattice
        lattice[i] = new char*[lattice_size];
        for(j = 0; j < lattice_size; j++){  // kolommen van het lattice
            lattice[i][j] = new char[lattice_size];
        }
    }

    for(i = 0; i < lattice_size; i++){
        for(j = 0; j < lattice_size; j++){
            for(k = 0; k < lattice_size; k++){
                lattice[i][j][k] = 0;       // lattice in alle (=3) dimensies op 0 zetten
            }
        }
    }
    for(i = 0; i < MPL+1; i++){
        x[i] = 0;                           // x, y, z en gemw in 1 dimensie op 0 zetten
        y[i] = 0;
        z[i] = 0;
        teltak[i] = 0;
        gemw[i] = 0.0;
    }
    xmid = gridsize;                        // x-coordinaat van middelpunt lattice
    ymid = gridsize;                        // y-coordinaat van middelpunt lattice
    zmid = gridsize;                        // z-coordinaat van middelpunt lattice

  /**
   * Begin met groeien van M*S polymeren
   **/
    srandom(time(NULL));                    // 'randomize' random-number-generator
    while(m < M*S){
    x[0] = xmid;                            // polymeer start in midden
        y[0] = ymid;
        z[0] = zmid;
        lattice[xmid][ymid][zmid] = 1;      // die eerste site in rooster als 'bezet' merken
        poly_len = grow(0,0.25);            // Begin met groeien (lengte=0,
                                            // init.rosenbl.gew.=1, maar we houden 1/4*w bij)
        for(i = 0; i <= poly_len; i++){
            lattice[x[i]][y[i]][z[i]] = 0;  // 'Wis' het gelopen pad weer van lattice
        }
    }
  /**
   * Uitvoer van gegevens over alle M*S poly's
   **/
     /**
      * Telkens werd 1/4 rosenbl.gew. genomen. Dus (echte gewicht)=
      * (berekende gewicht) * 4^(MPL)
      * sumr2w moet dus met 4^(MPL) verm. maar dat wordt gedeeld door
      * sumw * 4^(MPL). (Netto * 1).
      **/
    double nu = log(sumr2w/sumw)/log(MPL)/2;  // 'nu' over alle poly's bepaald

  /**
   * Bepaling statistische fout m.b.v. standaarddeviatie van deelnu's
   * SD(deelnu) = sqrt{<nu^2>-<nu>^2}
   **/

    cout << "#Statistische fout na " << setnr << " sets van " << M << " polymeren is: " <<
             sqrt((somnu2/setnr - (somnu/setnr)*(somnu/setnr))/setnr) << endl;
    cout << MPL << " " << nu << endl;
  /**
   * Maak geheugen weer vrij
   **/
    delete[] x, y, z, teltak, lattice, *lattice, **lattice;
}
```

A.2 grow3.cpp

```
#include <iostream.h>
#include <math.h>

extern int *x, *y, *z;      // x-, resp y-coordinaten van polymeer
extern int *teltak;         // Hoe vaak er al een i-de tak is aangegroeid (i.v.m. gemw bijhouden)
extern char ***lattice;     // geeft voor iedere site aan of 'ie bezet(1) of onbezet (0) is
extern int nbx[6], nby[6], nbz[6];  // voor opslag van x-, resp. y-coordinaten van (max 4) buren
extern int MPL,             // Maximale Polymeer Lengte
    M,                      // Aantal polymeren per set
    m,                      // teller voor hoeveel poly's al zijn gegroeid (start met 0)
    setnr,                  // Aantal sets (van M poly's) dat reeds is gegroeid
    xmid, ymid, zmid,       // x- en y-coord. van midden van lattice
    pruntel,                // telt aantal pruningen
    enrichtel,              // telt aantal enrichments
    gridsize;               // Breedte/hoogte van het grid (vanaf middelpunt)

extern double *gemw;        // gemiddeld Rosenbluth gewicht bij bepaalde lengte van poly
extern double crit_prun,    // critische factor voor pruning (ong. 0.1)
       crit_enr,            // critische factor voor enrichment (ong. 10)
       sumw,                // som (over alle poly's) van Rosenbluthgewicht
       oldsumw,             // oude waarde van sumw (na vorige set)
       sumr2w,              // som (over alle poly's) van Rosenbluthgewicht*r^2
       oldsumr2w,           // oude waarde van sumr2w (na vorige set)
       rcorrection,         // voor random getal tussen 0 en 1
       somnu2,              // som nu^2 over alle sets
       somnu;               // som nu over alle sets

/**
 * groei een nieuw 'takje' aan polymeer
 * controleer op Pruning en Enrichment
 * Stop als poly lengte MPL heeft
 *
 * len = #takken,
 * w = rosenbl.gew. van laatste tak
 **/

int grow(int len, double w){
if(abs(x[len]-xmid)==gridsize||abs(y[len]-ymid)==gridsize||abs(z[len]-zmid)==gridsize){
    cout<<"# Out of gridbounds!"<<endl;
    return len;
}

 /**
  * Check huidige Rosenbluthgewicht voor Pruning
  **/
    if(w < crit_prun*gemw[len]){
        pruntel++;
        if(random()*rcorrection < 0.5)  // Kies met kans 1/2
            return len;                 // Voor polymeer beeindigen. Lattice schonen gebeurt elders
        else return grow(len, 2.0*w);   // Of huidig poly met dubbel gewicht verder groeien
    }

 /**
  * Check huidige Rosenbluthgewicht voor Enrichment
  **/
    if(w > crit_enr*gemw[len] && m > 0){         // Enrichment
        enrichtel++;
        int polylen1=grow(len, .5*w);            // Groei poly met half gewicht gewoon verder
        int tel;
        for(tel=len+1; tel <= polylen1; tel++)   // Wis dat laatste stuk
            lattice[x[tel]][y[tel]][z[tel]]=0;
        return grow(len, .5*w);                  // En groei nog zo'n stuk.
    }
 // Huidige w is OK: mee laten wegen in gemw[len]
    gemw[len] = (gemw[len]*teltak[len] + w)/(teltak[len]+1);
```

```
   teltak[len]++;

/**
 * Check of poly al juiste lengte heeft
 **/
   if (len==MPL) {                    // Polymeer heeft juiste lengte bereikt
       sumw += w;                     // sumw en sumr2w bijwerken
       sumr2w +=                      // w*(dx^2+dy^2+dz^2)
       w*((x[len]-xmid)*(x[len]-xmid)+(y[len]-ymid)*(y[len]-ymid)+(z[len]-zmid)*(z[len]-zmid));
       m++;                           // Aantal reeds gegroeide poly's met 1 ophogen

       if(m%M==0){                    // als een set afgerond is, bepaal 'nu' van die set
           double deelnu = (log((sumr2w-oldsumr2w)/(sumw-oldsumw))/log(MPL)/2);
           somnu += deelnu;
           somnu2 += deelnu*deelnu;
           setnr++;
           oldsumr2w = sumr2w; oldsumw = sumw;
       }
       return len;
   }

/**
 * Poly is nog niet af, dus buren zoeken
 **/
   int mynb,      // index (in nbx en nby) van gekozen buur
       numnb = 0; // aantal mogelijke buren

   if (lattice[x[len]][y[len]+1][z[len]]==0) { /*noorderbuur*/
     nbx[numnb]=x[len];
     nby[numnb]=y[len]+1;
     nbz[numnb]=z[len];
     numnb++;
   }

   if (lattice[x[len]+1][y[len]][z[len]]==0) { /*oosterbuur*/
     nbx[numnb]=x[len]+1;
     nby[numnb]=y[len];
     nbz[numnb]=z[len];
     numnb++;
   }

   if (lattice[x[len]][y[len]-1][z[len]]==0) { /*zuiderbuur*/
     nbx[numnb]=x[len];
     nby[numnb]=y[len]-1;
     nbz[numnb]=z[len];
     numnb++;
   }

   if (lattice[x[len]-1][y[len]][z[len]]==0) { /*westerbuur*/
     nbx[numnb]=x[len]-1;
     nby[numnb]=y[len];
     nbz[numnb]=z[len];
     numnb++;
   }

   if (lattice[x[len]][y[len]][z[len]-1]==0) { /*onderbuur*/
     nbx[numnb]=x[len];
     nby[numnb]=y[len];
     nbz[numnb]=z[len]-1;
     numnb++;
   }

   if (lattice[x[len]][y[len]][z[len]+1]==0) { /*bovenbuur*/
     nbx[numnb]=x[len];
     nby[numnb]=y[len];
     nbz[numnb]=z[len]+1;
```

```
      numnb++;
    }

 /**
  * Als er niet meer gegroeid kan worden (0 buren): stoppen
  **/
    if(numnb == 0){                     // geen onbezette buursites meer:
        return len;
    }

 /**
  * Verder groeien: Een buur kiezen kiezen voor de groeirichting
  **/
    mynb=(int)(random()*rcorrection*numnb);     // Kies 1 uit de <numnb> mogelijke buren

    lattice[nbx[mynb]][nby[mynb]][nbz[mynb]]=1; // Bezet de gekozen buursite
    x[len+1]=nbx[mynb];                         // Voeg nieuwe polymeerdeeltje aan keten toe
    y[len+1]=nby[mynb];
    z[len+1]=nbz[mynb];

    return grow(len+1, w*numnb/4.0);            // Groei recursief verder
}
```

B Programma-output in 3D

```
#Statistische fout na 500 sets van 500 polymeren is: 0.000480003
2 0.632279
#Statistische fout na 500 sets van 500 polymeren is: 0.000471128
2 0.631849
#Statistische fout na 500 sets van 500 polymeren is: 0.000460204
2 0.631945
#Statistische fout na 500 sets van 500 polymeren is: 0.000367007
4 0.618399
#Statistische fout na 500 sets van 500 polymeren is: 0.000368497
4 0.618087
#Statistische fout na 500 sets van 500 polymeren is: 0.000342434
4 0.619251
#Statistische fout na 500 sets van 500 polymeren is: 0.000337206
5 0.614615
#Statistische fout na 500 sets van 500 polymeren is: 0.000337671
5 0.614728
#Statistische fout na 500 sets van 500 polymeren is: 0.000278788
10 0.612969
#Statistische fout na 500 sets van 500 polymeren is: 0.00029035
10 0.612797
#Statistische fout na 500 sets van 500 polymeren is: 0.00027378
25 0.609843
#Statistische fout na 500 sets van 500 polymeren is: 0.000249979
25 0.609839
#Statistische fout na 500 sets van 500 polymeren is: 0.000277515
25 0.60937
#Statistische fout na 500 sets van 500 polymeren is: 0.000315063
50 0.608939
#Statistische fout na 500 sets van 500 polymeren is: 0.00030895
50 0.608822
#Statistische fout na 500 sets van 500 polymeren is: 0.000314143
50 0.609072
#Statistische fout na 500 sets van 500 polymeren is: 0.000319225
100 0.607718
```

```
#Statistische fout na 500 sets van 500 polymeren is: 0.000318252
100 0.608053
#Statistische fout na 500 sets van 500 polymeren is: 0.000333361
100 0.608225
#Statistische fout na 500 sets van 500 polymeren is: 0.000360756
250 0.605761
#Statistische fout na 500 sets van 500 polymeren is: 0.000396978
250 0.605692
#Statistische fout na 500 sets van 500 polymeren is: 0.000378308
250 0.604556
#Statistische fout na 524 sets van 500 polymeren is: 0.00136363
450 0.604556
#Statistische fout na 2293 sets van 500 polymeren is: 0.000876217
450 0.606173
#Statistische fout na 9826 sets van 50 polymeren is: 0.000476332
500 0.604644
#Statistische fout na 407962 sets van 50 polymeren is: 6.9404e-05
1000 0.596821
#Statistische fout na 10 sets van 10 polymeren is: 0.01668
1000 0.574342
#Statistische fout na 100 sets van 100 polymeren is: 0.003338884
1000 0.589475
```