Polymer simulation: Configurational entropy and mean end-to-end distance of chains via Monte Carlo method

Risto-Antti Paju, Queens' College, Cambridge

17th January 2001

Part II Physics, Computational Physics exercise

Except where specific reference is made to the work of others, this work is original and has not been already submitted either wholly or in part to satisfy any degree requirement at this or any other University.

Risto A Paju

Abstract

The number w of possible configurations of a polymer molecule of n monomer units has been investigated in a cellular cubic grid simulation. If the multi-occupancy of monomers in one cell were allowed, w would have the value $6 \times 5^{n-2}$; a Monte Carlo method has been implemented to test what fraction of these are possible when multi-occupancy is forbidden. In addition, the mean end-to-end distance $\langle r^2 \rangle$ has been computed.

These procedures have been performed with models where n ranges from 2 to 50, and the functional dependence of w and $\langle r^2 \rangle$ on n has been investigated. Theoretical considerations have suggested the models

$$w \propto n^g e^{\alpha n}$$
$$\langle r^2 \rangle \propto n^\theta$$

where g, α and θ are constants. The exponents of n were found to be

$$g = 0.21 \pm 0.05$$
$$\theta = 1.36 \pm 0.03$$

The value of θ obtained here gives substantial support for a theory which predicts $\theta = 4/3$.

Contents

1 Introduction **4**

2 Computational approach **4**

3 Implementation **5**
3.1 Chain propagation 5
3.2 Constructing complete chains 5
3.3 Repetition 5
3.4 Main program 6
3.5 Function fitting of the data 6
3.6 Performance optimizations 6

4 Results and discussion **7**
4.1 Execution performance 7
4.2 Debugging 7
4.3 Numerical results 7
4.4 Errors and possible improvements 8

5 Conclusions **11**

A Program source **12**

1 Introduction

The number w of possible configurations of a polymer molecule is essential for entropy calculations: $S = k_B \ln w$. Its dependence on the number n of monomer units is the subject of this simulation. In addition, the mean square end-to-end distances $\langle r^2 \rangle$ of the polymer chains are computed, and their relation to n is investigated.

Theoretical models suggest the functional relations

$$\begin{aligned} w &= an^g e^{\alpha n} \\ \langle r^2 \rangle &= cn^\theta \end{aligned} \tag{1}$$

where a, g, α, c and θ are constants. Different models have predicted θ to be either 1.17, $6/5$ or $4/3$. The simulation aims to determine which of these is closest to correct (within the scope of this type of simulation).

The question is approached with a simplified cellular model. Each monomer unit will occupy a cell in a cubic grid. A chain is built up starting from two units, by adding a unit to one of the adjacent face-sharing cells of the previously added monomer. This would give $w = 6 \times 5^{n-2}$, but the actual number must be less because a cell may only be occupied by one unit. A Monte Carlo method is implemented to find what fraction of the $6 \times 5^{n-2}$ configurations are allowed.

The FORTRAN programming language is used to implement the simulation. NAG routines are used to extract approximations of a, g, α, c and θ from the data obtained from the simulation.

2 Computational approach

The most basic starting point would be to build up all possible configurations and count the number. However, this is not computationally viable. Each polymer begins with a unit at the origin and another one in an adjacent face-sharing cell. From then on, there are five directions in which to continue after adding each monomer, so the total number would be $6 \times 5^{n-2}$. For $n = 50$ this would give a w in the order of 10^{34}. However, because the multi-occupancy of cells is forbidden, the actual number is considerably smaller. Nevertheless, it is too large to be computed in a reasonable time.

A Monte Carlo approach is used to circumvent this problem. A number of polymer chains are built up randomly, using the random number function to choose one of the five directions at each stage. When the polymer overlaps with itself, the chain is considered a failure. A certain fraction of all attempted chains will be successful, and it is expected that this fraction, multiplied by $6 \times 5^{n-2}$, gives a reasonable approximation of the number of all possible configurations.

The polymer chains are constructed in a 'space' of a $2n \times 2n \times 2n$ matrix whose elements are initially set to zero. A monomer unit in a cell is marked by setting that element of the matrix to one. Failure is indicated, if the system attempts to insert a monomer into a cell with the value one.

While there are many kinds of possibilities for the failure of building up a chain, depending on the number of overlaps, it suffices to stop the build-up and report failure when the first overlap is encountered. The correct number of successful chains is obtained in any case.

The number of trials for a given n is crucial to the success of the Monte Carlo method. Clearly, a larger number is required for larger n, but the relation between the two numbers is not obvious. The solution is to process sets of trials, keeping record of the total number of successes and trials. If, as the result of a further set of trials, the success fraction changes by less than a certain measure of precision, the number of total trials has been sufficient.

For each successful chain, the value of r^2, the squared end-to-end distance, is added to a cumulative sum, and the mean $\langle r^2 \rangle$ is easily calculated at the end.

After repeating the above procedures for different n, we have sets of $(n, w, \langle r^2 \rangle)$. Least-squares methods are then used to compute the constant parameters in eq. 1.

3 Implementation

The explanation of the program below does not attempt to follow the order of procedures in the code. Rather, it reflects the way in which the program was actually planned and written, from the simple, core routines to the higher-level wrapper procedures.

The construction of a polymer chain begins from the origin (at the centre of the grid) and the next unit is always placed at $(1, 0, 0)$. While this is only one of the six possible starting directions, the symmetry of the problem implies it is a correct choice, as long as the w for all (even overlapping) polymers is $6 \times 5^{n-2}$, the number obtained from considering all six starting directions.

3.1 Propagation of the chain by one unit: `subroutine propagate`

For the purpose of choosing the allowed direction of propagation, a record is kept of the last inserted monomer (`curpoint`) and the one before that (`prevpoint`).

The coordinates of the six closest neighbour cells of `curpoint` are recorded into a list. The one of these which coincides with `prevpoint` is removed from the list. One of the remaining five points is chosen at random, and it will become the new `curpoint`.

3.2 Constructing complete chains: `subroutine make_chains`

If this new point in the grid has the value zero, a new unit is inserted into it, by setting its value to one. The construction can then proceed. If an overlap is encountered, failure is reported.

The propagation routine and the above test are repeated until failure, or until n monomers have been used. In the latter case, r^2 is computed and added to a cumulative $\sum r^2$, and a `successes` counter is incremented. The construction is repeated by `trials`, the number of trials in a set.

3.3 Repetition: `subroutine get_data`

Initially, one set of trials is performed. A `do` loop is then used to repeat the sets until the required precision is reached; since the w is proportional to the success fraction, it suffices to compare the

fractions after i and $i-1$ sets of trials. Specifically, the condition for sufficient convergence is

$$\left|\frac{2(\text{fraction}_{\text{new}} - \text{fraction}_{\text{old}})}{\text{fraction}_{\text{new}} + \text{fraction}_{\text{old}}}\right| < \text{precision}$$

During all of the above subroutines, variables such as `successes`, total trials and $\sum r^2$ are accumulated. When the convergence limit has been reached, w and $\langle r^2 \rangle$ are computed.

3.4 Main program

The main program begins with questions of the number of different n to use, beginning from $n = 2$, and the intervals, allowing maximum $n = 50$. `precision` and `trials` are also queried. A `do` loop is used to `get_data` for the set values of n, and they are stored in an array.

3.5 Least squares fit of the data: `subroutine w_fit, r2_fit`

From eq. 1, we get the logarithmic dependences

$$\ln w = \ln a + g \ln n + \alpha n \tag{2}$$

$$\ln\langle r^2 \rangle = \ln c + \theta \ln n \tag{3}$$

Equation 3 suggests a simple linear regression. The NAG routine G02CAF is used to obtain c, θ and their standard errors.

Eq. 2 is more complicated, and it requires a generalized linear regression of the form $y = a + bx_1 + cx_2$. The suitable NAG routine for this is G02DAF.

The subroutines `w_fit, r2_fit` are wrapper scripts around the NAG routines G02DAF and G02CAF respectively, to make the main program cleaner. This is particularly important for the latter routine, which has more than twenty arguments, although only a few of them are directly shared with the main program.

3.6 Performance optimizations

Besides the obvious guideline of searching the shortest code for a given algorithm, there was another general idea that appeared to reduce processing time. Variables should not be initialized and deleted unnecessarily. For instance, the constant `modifier` array used in `propagate` is initialized outside the subroutine. This subroutine is being called the most often, so it was useful to reduce the number of variables initialized every time. This principle has, in many cases, been compromised by clarity; otherwise the number of variables passed to and from subroutines would grow too large to be handled conveniently. But since the calls to `propagate` are the most critical to performance, it was decided to initialize all its variables outside.

In addition, the `-O` compiler flag was used to add some umph to the binary. As a result, the execution time was approximately halved.

4 Results and discussion

4.1 Execution performance

After implementing the optimization techniques discussed in section 3.6, a significant increase in processing speed was observed. The processing time was reduced to roughly one quarter of the original. The optimum number of trials per set was found to be somewhere between 10 and 15. When $n = 2, 3, 4, ...50$, $\texttt{precision} = 10^{-3}$, the results were produced in about 30 minutes.

4.2 Debugging

Much of the debugging information was found by printing values of variables at several points of the program. At the very least, these would indicate the point in the code where the program crashes, and often provide specific information. For instance, the success fraction was printed out after each set of trials - its value should stay roughly constant and gradually converge until reaching the required precision. It could also be checked that the success fraction had a reasonable value, decreasing with n.

For $n = 2, 3, 4$ the success fraction has to be unity; therefore, the resulting w could be readily predicted. The values $6, 30$ and 150 were produced as expected, so the basic construction of the chains was shown to be correct. For $n > 4$ the code was, of course, the same but proceeded further. Reasonable values of the success fraction, combined with this fact, suggest that the code is basically correct.

Probably the most complete and direct proof of correct working is the plotting of graphs 1 - 4. There the data points and the resulting functions were combined. It could immediately be seen if, for example, two parameters had been swapped by accident. More importantly, the degree of correlation between the data points and the curves based on equations 1 could be quickly verified.

Using short scripts, the graphs were produced automatically from the output files, so this test could be repeated easily.

4.3 Numerical results

A snapshot of the results from one run of the program is provided here:

$$
\begin{aligned}
&n_{\text{min}} = 2 \\
&n_{\text{interval}} = 1 \\
&n_{\text{max}} = 50 \\
&\text{trials/set} = 13 \\
&\text{precision} = 10^{-3} \\
\\
&a = 0.23 \pm 0.02 \\
&g = 0.21 \pm 0.05 \\
&\alpha = 1.541 \pm 0.003 \\
&c = 0.6 \pm 0.8 \\
&\theta = 1.36 \pm 0.03
\end{aligned}
$$

The data, along with curves based on the resulting parameters, are plotted in figures 1 - 4 to illustrate the degree of correlation with the theory.

It should be noted that this is merely an example of the possible results; different test runs with the same input parameters give slightly varying results. However, they are mostly consistent within the uncertainty limits. It might be considered to run several instances of the simulation and average the results, but the same degree of accuracy should be reached with a single run of sufficient duration, i.e. using a small enough value of `precision`.

The correlation of $\langle r^2 \rangle$ vs n is apparent from figures 3 and 4. There is considerable scattering in the values of $\langle r^2 \rangle$ at high n, particularly in graph 3. Nevertheless, the computational implication is that the constant of proportionality c, not θ, is the one with relatively high uncertainty. This is well apparent from figure 4: the gradient θ is fairly precise with a 2% standard error, and this clearly singles out the theoretical model which predicts $\theta = 4/3$.

The correlation between w and n is slightly more difficult to assess. While the theoretical formula of $w(n)$ fits the data rather well, the apparent effect of g is suppressed by the exponential factor. Neither of the figures 1 or 2 can be used for the graphical evaluation of g. Computationally, this is reflected in the high relative uncertainty of g.

4.4 Errors and possible improvements

The relatively high level of scattering of data points relating $\langle r^2 \rangle$ and n may be an effect of the fact that we decided the sufficient number of repetitions on the basis of the precision of w, not $\langle r^2 \rangle$. One might want to modify the program to test the convergence of $\langle r^2 \rangle$ instead, perhaps even both at the same time. However, it was curious to note that improving `precision` had very little effect on the degree of scattering of $\langle r^2 \rangle$ - it was roughly the same with any `precision` less that 10^{-2}. Since the convergence test only applies to a single data point (i.e. value of n), the problem of scattering cannot be directly solved via this method. It might even be the case that the quirks in the distribution of $\langle r^2 \rangle$ are inherent in the cubic cellular model.

A further source of fluctuations between the results in separate runs may be the imperfection of the random number generator. Truly random numbers cannot, in principle, be generated by conventional computers.

Furthermore, it was occasionally the case that the success fraction converged very rapidly, only after a couple of iterations, even for large n which usually took hundreds of sets of trials to converge. The values of w and $\langle r^2 \rangle$ would then differ considerably from a 'good' value which would be obtained with many more iterations. Be it due to the poor quality of random numbers or some ofher fluke, the problem is difficult to tackle with the principles used here.

As an aside, the idea occurred whether the number of possible trials should be 5^{n-2} with or without the factor of six. It might be argued that the variations obtained by rotating one chain to give the six different directions are not physically distinct, as long as there is nothing else in the system, and therefore should be treated as the same state. Moreover, should the rotation of the cubic grid w.r.t our reference frame by, say, 45° be considered a separate state? However, for the purposes of this simulation is suffices to say that, provided the cubic grid is the reference frame, and the model will be applied to a system of several polymer chains (i.e. macroscopic systems), then the number of different orientations is precisely six. Besides, in this exercise

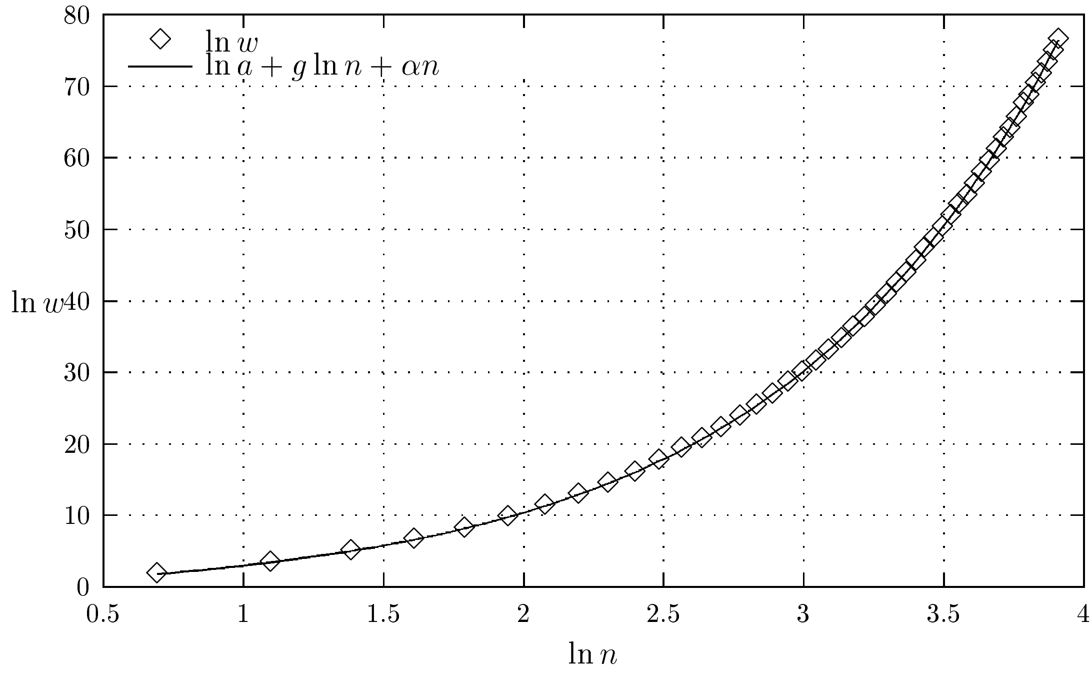

Figure 1: $\ln w$ vs. $\ln n$

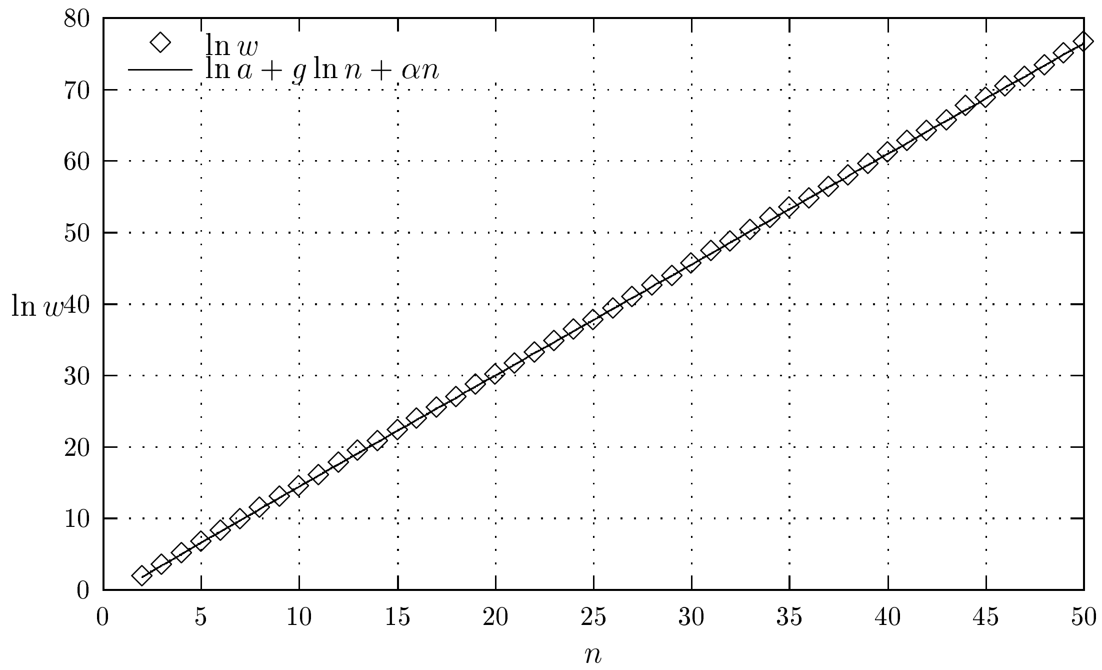

Figure 2: $\ln w$ vs. n

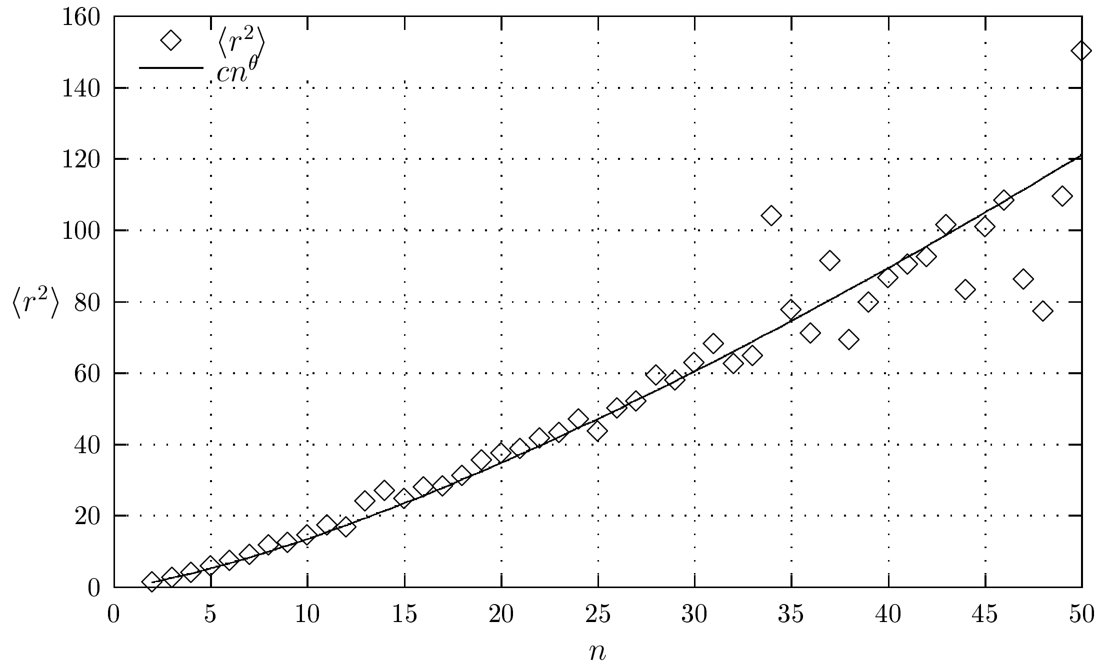

Figure 3: $\langle r^2 \rangle$ vs n

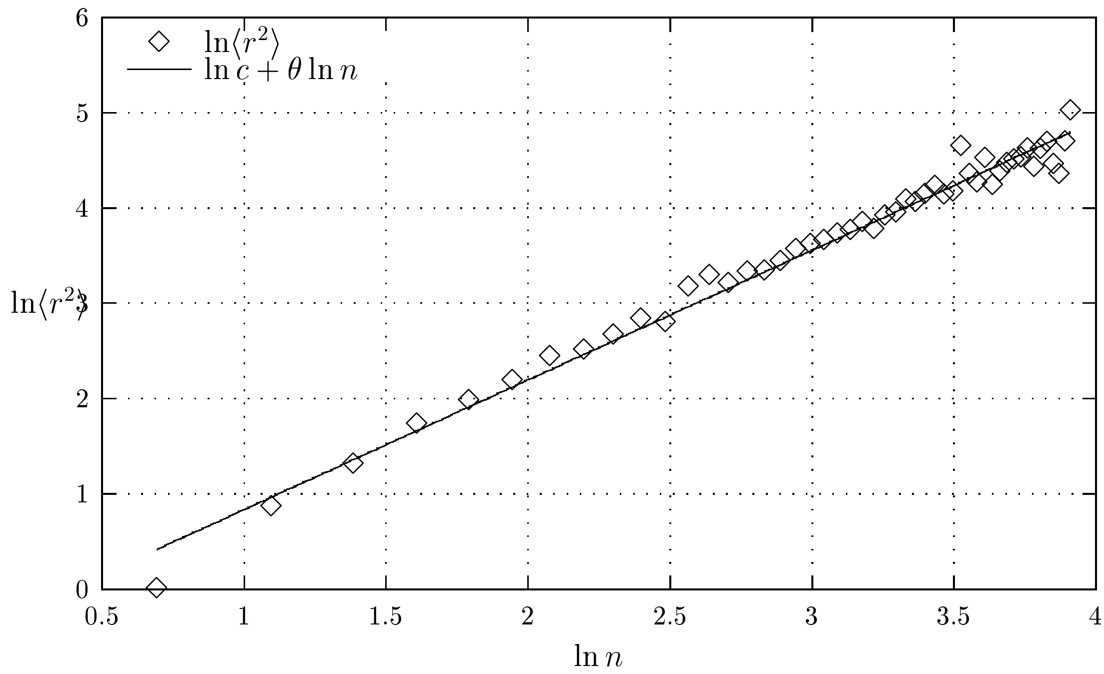

Figure 4: $\ln\langle r^2 \rangle$ vs. $\ln n$

we are primarily interested in the exponents of n in the expressions $w(n), \langle r^2 \rangle (n)$ on which the prefactor of 5^{n-2} has no effect.

5 Conclusions

The number of configurations and the mean end-to-end distance of a polymer chain of n units, $n = 2...50$, were investigated using a cellular based Monte Carlo approach. Analysis of the resulting data points $(w, n, \langle r^2 \rangle)$ support the theoretically predicted relations

$$w \propto n^g e^{\alpha n}$$
$$\langle r^2 \rangle \propto n^\theta$$

and the exponents of n were found to be

$$g = 0.21 \pm 0.05$$
$$\theta = 1.36 \pm 0.03$$

References

[1] Computational Physics, Course material; P. Alexander, Cavendish Laboratory 2000

[2] Fortran 90 Essentials, Cornell Theory Center, http://www.ccwr.ac.za/ccwr/users/kure/more.html

[3] NAG FORTRAN 77 Library Reference

APPENDIX

A Program source

```
program polymer
implicit none

!n = number of polymer units, between 2 and 50
!w = number of configurations (as in S = k ln w)
integer, parameter :: dp = kind(1.0d0)

integer n, i, j, n_interval, no_results, trials
real (kind=dp), dimension(:, :), allocatable :: results, w_reg_data, r_reg_data
real precision !relative precision required of w

integer, parameter :: n_min = 2

real (kind=dp) :: data_array(2), w_params(2, 3), r_params(2, 2)

1 print *, "n_min =", n_min,". Enter the interval and number of n-values:"
read *, n_interval, no_results
print *, "n_max is ", n_min + (no_results-1)*n_interval
if (n_interval .lt. 1) then
   print *, "n_interval should be an integer .ge. 1"
   goto 1
elseif(n_interval*(no_results-1) .gt. 48) then
   print *, "maximum n should be .le. 50"
   goto 1
endif

2 print *, "Enter the required precision (percentage, 0.1 to 10)"
print *, "used as the convergence limit of fraction:"
read *, precision
if (precision .ge. 0.1 .and. precision .le. 10) then
   precision = real(precision) / real(100)
else
   goto 2
endif

3 print *, "Enter the number of trials per set (1 to 100)"
read *, trials
```

```
if (trials .gt. 100 .or. trials .lt. 1) then
   goto 3
endif

print *, ""
print *, "  n                    w                    <r^2>"
print *, ""

allocate(results(3, no_results))
allocate(w_reg_data(3, no_results))
allocate(r_reg_data(2, no_results))

!obtain sets of w, r2 for each n
do i = 1, no_results
   n = n_min + n_interval*(i-1)
   data_array = get_data(n, trials, precision)
   results(1, i) = n
   results(2:3, i) = data_array !w, mean_r2
   print *, results(:, i)
enddo

!ln w = ln a + g ln n + alpha n
w_reg_data(1, :) = log(results(2, :)) !ln w
w_reg_data(2, :) = log(results(1, :)) !ln n
w_reg_data(3, :) = results(1, :) !n

!ln r2 = ln c + theta ln n
r_reg_data(1, :) = log(results(3, :)) !ln mean_r2
r_reg_data(2, :) = log(results(1, :)) !ln n

print *, "Parameters a, g, alpha, c and theta in"
print *, "w = a * n**g * exp(alpha*a)"
print *, "mean r**2 = c * n**theta"
print *, ""

w_params = w_fit(no_results, w_reg_data)
print *, "a = ", exp(w_params(1, 1)), "+/-", &
     exp(w_params(1, 1))*w_params(2, 1)
print *, "g = ", w_params(1, 2), "+/-", w_params(2, 2)
print *, "alpha = ", w_params(1, 3), "+/-", w_params(2, 3)

r_params = r_fit(no_results, r_reg_data)
print *, "c = ", exp(r_params(1, 1)), "+/-", &
     exp(r_params(1, 1))*r_params(2, 1)
print *, "theta = ", r_params(2, 1), "+/-", r_params(2, 2)
```

```
!output data points to files for gnuplot
open(1, file='r2.dat', status='replace')
do i=1, no_results
   !ln n, ln r2, n, r2 - gnuplot can then select appropriate pairs for plotting
   write (1, *) r_reg_data(2, i), r_reg_data(1, i), results(1, i), &
        results(3, i)
enddo
close(1)

open(1, file='w.dat', status='replace')
do i=1, no_results
   !n, log(n), log(w) - ditto for gnuplot
   write(1,*) w_reg_data(3, i), w_reg_data(2, i), w_reg_data(1, i)
enddo
close(1)

!write parameters to files for gnuplot curve drawing
open(1, file='w-params.plot', status='replace')
write(1, *) "a = ", exp(w_params(1, 1))
write(1, *)  "g = ", w_params(1, 2)
write(1, *) "alpha = ", w_params(1, 3)
close(1)

open(1, file='r2-params.plot', status='replace')
write(1, *)  "c = ", exp(r_params(1, 1))
write(1, *) "theta = ", r_params(2, 1)
close(1)

!write output of parameters for TeX
open(1, file='params.tex', status='replace')
write(1, *) "\[\begin{array}{l}"

write(1, *) "n_{\rm min} = ", n_min, "\\"
write(1, *) "n_{\rm interval} = ", n_interval, "\\"
write(1, *) "n_{\rm max} = ", n_min + (no_results-1)*n_interval, "\\"
write(1, *) "{\rm trials/set} = ", trials, "\\"
write(1, *) "{\rm precision} = ", precision, "\\ \\"

write(1, *) "a = ", exp(w_params(1, 1)), "\pm", &
     exp(w_params(1, 1))*w_params(2, 1), "\\"
write(1, *) "g = ", w_params(1, 2), "\pm", w_params(2, 2), "\\"
write(1, *) "\alpha = ", w_params(1, 3), "\pm", w_params(2, 3), "\\"
write(1, *) "c = ", exp(r_params(1, 1)), "\pm", &
     exp(r_params(1, 1))*r_params(2, 1), "\\"
```

```
write(1, *) "\theta = ", r_params(2, 1), "\pm", r_params(2, 2)
write(1, *) "\end{array}\]"
close(1)

deallocate(results)
deallocate(w_reg_data)
deallocate(r_reg_data)

contains
  function r_fit(n, r_reg_data)
    !get regression parameters c and theta in r2 = c * N**theta
    !notice here n = no_results..
    integer, parameter :: dp = kind(1.0D0)
    integer :: ifail = 0, n
    real (kind=dp), dimension(:), allocatable :: x, y
    real (kind=dp), dimension(:, :) :: r_reg_data
    real (kind=dp), dimension(20) :: result
    real (kind=dp), dimension(2, 2) :: r_fit

    allocate(x(n))
    allocate(y(n))

    x = r_reg_data(2, :) !ln n
    y = r_reg_data(1, :) !ln r2

    call g02caf(n, x, y, result, ifail)

    deallocate(x)
    deallocate(y)

    r_fit(1, 1) = result(7) !reg. constant, i.e. log(c)
    r_fit(1, 2) = result(9) !its std.dev.

    r_fit(2, 1) = result(6) !reg. coefficient, i.e. theta
    r_fit(2, 2) = result(8) !its std.dev.

  end function r_fit

  function w_fit(n, w_reg_data)
    !get regression parameters a, alpha, g in w = a * N**g * exp(alpha*N)
    !but here n = number of results... (for the NAG routine)
    integer, parameter :: dp = kind(1.0D0), ip = 3, m = 2
    integer, intent(in) :: n
    integer :: idf, irank, ldx, ldq, ifail = 0
    real (kind=dp) :: rss, tol = 0.000001
```

```
  real (kind=dp), dimension(2, 3) :: w_fit
  real (kind=dp), dimension(:, :), intent(in) :: w_reg_data
  integer, dimension(2) :: isx = 1
  real (kind=dp) :: b(ip), se(ip), cov(ip*(ip+1)/2), q(n, ip+1), x(n, m), &
       y(n), wt(n), res(n), h(n), p(2*ip+ip*ip), wk(5*(ip-1)+ip*ip)
  logical svd

  !m: use constant term; u: data not weighted
  character*1 :: mean = 'm', weight = 'u'

  y = w_reg_data(1, :)
  x = reshape(w_reg_data(2:3, :), shape = (/n, 2/), order = (/2, 1/))
  wt = 1
  ldx = n
  ldq = n

  call G02DAF(MEAN, WEIGHT, N, X, LDX, M, ISX, IP, Y, WT, RSS, IDF, &
       B, SE, COV, RES, H, Q, LDQ, SVD, IRANK, P, TOL, WK, IFAIL)

  w_fit(1, :) = b(1:3) !a, g, alpha
  w_fit(2, :) = se(1:3) !their standard errors

end function w_fit

function get_data(n, trials, precision)
  real (kind=dp), dimension(2) :: get_data
  integer, dimension(:, :, :), allocatable :: grid
  real (kind=dp) fraction_old, fraction_new, w, mean_r2
  real precision
  integer successes, n, trials, sum_r2

  !make space
  allocate(grid(-n:n, -n:n, -n:n))

  successes = 0
  sum_r2 = 0
  mean_r2 = 0
  call make_chains(grid, n, trials, successes, sum_r2)
  fraction_old = real(successes) / real(trials)
  !print *, fraction_old
  !letting successes cumulate, we compute the fraction of total
  !successes in total trials , and stop when an additional set of
  !trials makes a difference less than precision.
  do j = 1, 1000000
     call make_chains(grid, n, trials, successes, sum_r2)
```

```
      fraction_new = real(successes) / real(trials*(j+1))
      if (fraction_new + fraction_old == 0 .or. j < 2) then
         continue
      elseif(2*abs((fraction_new-fraction_old)/(fraction_new+fraction_old)) &
           < precision) then
         exit
      endif
      fraction_old = fraction_new
      !print *, fraction_old
   enddo

   deallocate(grid)

   w = fraction_new * real(6) * real(5)**real(n-2)
   mean_r2 = real(sum_r2) / real(successes)

   get_data(1) = w
   get_data(2) = mean_r2

 end function get_data

 subroutine make_chains(grid, n, trials, successes, sum_r2)
   !build a polymer chain <trials> times. Test for failure.
   !For successful chains, compute r2.
   logical :: failed
   integer, intent(in) :: n, trials
   integer k, l, r2, successes, sum_r2, i, j
   real rand
   integer, dimension(6, 3) :: modifier = reshape((/1, 0, 0, -1, 0, 0,  &
        0, 1, 0, 0, -1, 0, 0, 0, 1, 0, 0, -1/), (/6, 3/), order=(/2, 1/))

   !coordinates for new unit & the one before that
   integer, dimension(3) :: curpoint, prevpoint

   !six directions to go, but one of them (prevpoint) will be removed
   integer, dimension(6, 3) :: nearpoints
   integer, dimension(5, 3) :: newpoints

   !the space
   integer, dimension(-n:n, -n:n, -n:n) :: grid

   do l = 1, trials

      !fill space with zeros
      grid = 0
```

```
      !insert first unit
      grid(0, 0, 0) = 1

      !we start in the x-direction
      curpoint(1) = 1
      curpoint(2) = 0
      curpoint(3) = 0

      prevpoint = 0

      !insert the following unit, propagate to next coordinate,
      !check if we can insert a unit there
      failed = .false.
      do k = 1, n - 2
         grid(curpoint(1), curpoint(2), curpoint(3)) = 1
         call propagate(curpoint, prevpoint, modifier, nearpoints, &
              newpoints, rand, i, j)
         if (grid(curpoint(1), curpoint(2), curpoint(3)) == 1) then
            failed = .true.
            exit
         endif
      enddo

      if (.not. failed) then
         successes = successes + 1
         r2 = curpoint(1)**2 + curpoint(2)**2 + curpoint(3)**2
         sum_r2 = sum_r2 + r2
      endif

   enddo

end subroutine make_chains

subroutine propagate(curpoint, prevpoint, modifier, nearpoints, newpoints, &
      rand, i, j)
   !move curpoint to the next point

   real rand
   integer i, j
   integer, dimension(3) :: curpoint, prevpoint

   !six directions to go, but one of them (prevpoint) will be removed
   integer, dimension(6, 3) :: nearpoints, modifier
   integer, dimension(5, 3) :: newpoints
```

```
    do i=1, 6
       nearpoints(i, :) = curpoint
    enddo

    !turn 'nearpoints' into the actual points near curpoint
    nearpoints = nearpoints + modifier

    !copy these into newpoints, except the one which coincides with prevpoint
    j = 1
    do i = 1, 6
       if(any(nearpoints(i, :) /= prevpoint)) then
          newpoints(j, :) = nearpoints(i, :)
          j = j + 1
       endif
    enddo

    prevpoint = curpoint

    !choose the new point
    call random_number(rand)
    i = int(5*rand) + 1
    curpoint = newpoints(i, :)

  end subroutine propagate

end program polymer
```