The Working Man's Guide to Obtaining Self Diffusion Coefficients from Molecular Dynamics Simulations

David Keffer
Department of Chemical Engineering
University of Tennessee, Knoxville
Begun: November 11, 2001
Last Updated: March 3, 2002

Table of Contents

I. Introduction

The objective of these notes is to provide a method to obtain a self-diffusion coefficient from a molecular dynamics simulation.

Our molecular dynamics code, mddriver.f, periodically saves positions as a function of time. From these positions we can obtain a self-diffusion coefficient. The self-diffusivity is different than a transport-diffusivity, aka a Fickian diffusivity. The self-diffusivity describes the random motion of a molecule in the absence of any gradients that would cause a mass flux. The derivation of the self-diffusivity can be obtained from probability theory.[1] I recommend that you check it out. It is not too complicated. In the next lecture packet, we are going to relate the self-diffusivity to the transport diffusivity. Here, we simply obtain the self-diffusivity.

I learned the general algorithm for obtaining the self-diffusivity from Haile.[2] Other references also discuss the numerical procedure to obtain the self-diffusivity.[3-4]

II. Theory

There are two common ways to obtain a self-diffusion coefficient. The first is from molecule positions and the second is from velocities. Theoretically, both methods yield the same result. Obtaining the self-diffusivity from the velocities involve integrating the velocity auto-correlation function, an example from what is called Green-Kubo relations.[2] There is a "long-time" tail to this integral that can cause numerical problems. Therefore, in my experience, I have found that simulations have to be longer to obtain a reliable self-diffusivity form the velocity than from the positions. In this hand-out we discuss only obtaining self-diffusion coefficients from position data.

Einstein related the self-diffusion coefficient to the mean square displacement of a particle as a function of observation time. The mean square displacement is proportional to the observation time in the limit that the observation time goes to infinity. The proportionality constant that relates the MSD to the observation time is called the self-diffusivity. By convention, we

$$D \equiv \frac{1}{2d} \lim_{t\to\infty} \frac{\left\langle \left[\underline{r}(t_o + t) - \underline{r}(t_o)\right]^2 \right\rangle}{t} \tag{1}$$

where D is the self-diffusion coefficient, and d is the dimensionality of the system. The numerator of equation (1) is the mean square displacement. The angled brackets indicate an ensemble average has been taken. The ensemble average is an average over all molecules in the simulation and all origins. By origins we mean that any time step can be considered the time zero in equation (1), because equation (1) is only looking at observation times (relative times or elapsed times) rather than some absolute time.

We can see that by saving the positions as a function of time, we can calculate the mean square displacement and obtain a self-diffusion coefficient.

II. Numerical Considerations

Our molecular dynamics codes saved positions of all particles every k_{msd} steps during the data production phase of the simulation. The algorithm of the code which computes the self-diffusion coefficients, get_diff.f, is as follows.

1. Read necessary parameters
2. Read positions
3. Calculate mean square displacement
4. Perform a linear least squares regression to obtain mean and variance of slope and intercept
5. Report mean and standard deviations of self-diffusion coefficients

We now discuss each step.

Read the Parameters

The code used to generate self-diffusion coefficients is called get_diff.f (FORTRAN 90) or get_diff.m (MATLAB). In order to calculate the self-diffusion coefficient, this code requires four parameters from the molecular dynamics simulation. These parameters are the number of molecules in the simulation, N, the number of production steps, `maxstp`, the interval during which the mean square displacements were saved, k_{msd}, and the size of the time step, Δt.

```
integer, parameter :: maxstp = 100000        ! number of data production steps
integer, parameter :: kmsd = 100             ! sampling interval
integer, parameter :: N = 216                ! number of molecules
double precision, parameter  :: dt = 2.d0 ! timestep (fs)
```

The code also requires a data file which includes N_{data} data points.

$$N_{data} = N\left[\text{int}\left(\frac{\text{maxstp}}{k_{msd}}\right)+1\right] \tag{2}$$

The only other parameter that needs to be specified is how long should we ignore the data before we consider that it has reached the "long time" asymptotical behavior of Einstein's relation. This parameter is discussed in detail in the example. There is no general rule for how long a simulation has to be run before we reach the long time asymptotical behavior predicted by Einstein's relation. However, below we present two examples, where this determination is made. In the code, this parameter is the number of time origin to ignore and is called, N_{min}.

Read the Positions

The second step in the algorithm is to read the positions from the data file. There are a lot of positions because we stored the time, x, y, and z position for every molecule at every save interval. If we have a machine with a lot of memory, then we can simply read in all the data in a huge matrix. (This is what the code get_diff.f does.) If we have more data than can fit in our computer's RAM at a single time, then we need to read a part at a time, perform the mean square

displacement calculation on that data, read in more data, and continue in this fashion. Haile provides a code which performs this piecewise reading of data.[2]

Calculate the Mean Square Displacement

The mean square displacement is simply an average. We know that for sample averages we simply use the formula

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{3}$$

Since this is the definition of the sample average, we too will use this formula. Our x is the square of the displacement. In fact we will use this formula for every value of time, t, in equation (1). In that way we can obtain the mean square displacement as a function of time.

The average is over both molecules and "time origins". Any point during the simulation can be considered a time origin. However, we would like all times to be represented equally in terms of the number of data points used to generate the MSD vs. time plot.

Consider that we have saved positions of N molecules at $N_t = 101$ times. We will only evaluate the MSD for $\Delta t = 1$ to $N_{max} = \text{int}(N_t/2) = 50$. Only the first 50 points will be considered as time origins. The number of data points for the MSD at each Δt will be $N*N_{max}$. We stop with $\Delta t = N_{max}$ because, when the elapsed time is half the total time, then all the MSD have the same number of data points contributing to them, and thus they are weighted equally. It is true that we could have 100 time origins for $\Delta t = 1$, 99 time origins for $\Delta t = 2$, 98 time origins for $\Delta t = 3$, … 2 time origins for $\Delta t = 99$, and 1 time origins for $\Delta t = 100$. If we did this we have two problems. First, you can't get good statistics averaging over just a few time origins. Therefore, you need to set N_{max} substantially smaller than N_t; and $N_{max} = \text{int}(N_t/2)$ has proven to be a good choice. Furthermore, if you select $N_{max} = \text{int}(N_t/2)$, then all of the Δt has the same number of time origins, namely N_{max}. This is a bit confusing; it always is. The discussion in Haile, has some graphics to illustrate the idea, but it is still somewhat confusing.[1] The only solution is to examine the code yourself and see exactly what we are doing, in order to ensure good statistics in the results.

Perform Linear Least Squares Regression

Once we have the MSD as a function of time, we simply need to perform a linear least squares regression to obtain the slope and intercept. If we examine equation one, it would seem that the slope is 2dD and the intercept is zero. However, this behavior is the long time behavior. Because there is some different (unknown) short time behavior, the linear portion of the curve is shifted so that the intercept is not zero. So in fact we use linear least squares regression to fit the model

$$\left\langle \left[\underline{r}(t_o + t) - \underline{r}(t_o)\right]^2 \right\rangle = b_0 + (2dD)t \tag{4}$$

The value of b_0 has no physical significance. However, if we don't include it in the regression, it changes the value of the slope.

We have made it clear that the linear relationship is only true in long times. Therefore, we don't want to perform the regression over all our data, just data where the observation time is long enough that we have reached the long-time asymptotical behavior. How long does this time have to be? It depends on the particulars of the system. In the examples below, we determine this minimum time.

In practice, we examine the x, y, and z diffusivities independently. In that case d=1, and obtain a slope and intercept for each dimension. If the system is isotropic, we can obtain the average diffusivity by averaging the values of the x, y, and z components of the diffusivity obtained independently.

Report mean and standard deviations of self-diffusion coefficients

Once we have the slope of the curve, we can directly obtain the best fit values of the self-diffusivity in the x, y, and z components. We can also obtain variances and standard deviations of these coefficients, from basic statistics. For a review on how to obtain the mean and variance of coefficients in a linear regression fit, visit the ChE 301 course website and check out the lecture packet on Regression.[5] The equations are all given there.

In an isotropic medium, the x, y, and z diffusivities should all be the same. One can calculate an average diffusivity by the arithmetic average of the three components. In this way, one reclaims the 2d factor in equation (1). The standard deviation of the self-diffusivity can be obtained by using the ordinary formula for a standard deviation from the three x, y, and z diffusivities. (See the ChE 301 notes in the "Statistics and Sampling Estimation" lecture packet.[5])

III. Two Examples

In this section we calculate self-diffusion coefficients for pure methane in the isotropic bulk liquid and bulk vapor phases. In runs 1 and 2, we ran a simulation of the liquid and vapor phase respectively, for 200,000 fs of production. For program 3, we ran the vapor phase for 2,000,000 fs (2 ns) of production. The parameters for the molecular dynamics simulations that generated these results are given in Table 1. The MD simulation program outputs are given in Program Outputs 1 & 2. (The output of program 3 is not shown.)

Table 1. Molecular Dynamics Parameters

Program 1: Liquid Methane	integer, parameter :: N = 216 ! Number of molecules T = 150.0d0 ! Temperature (K) Vn = 1.1323d+2 ! Ang^3/molecule (liq at 150 K & 1 atm) maxeqb = 10000 ! Number of time steps during equilibration maxstp = 100000 ! Number of time steps during data production dt = 2.0d0 ! size of time step (fs) sig = 3.884d0 ! collision diameter (Angstroms) eps = 137.d0 ! well depth (K) MW = 16.0420d0 ! molecular weight (grams/mole) rcut = 15.d0 ! cut-off distance for potential (Angstroms) ksamp = 1 ! sampling interval knbr = 10 ! neighbor list update interval kwrite = 5000 ! writing interval kmsd = 100 ! position save for mean square displacement rnbr = rcut + 3.d0
Program 2: Vapor Methane	All parameters the same as program 1 except: T = 298.0d0 ! Temperature (K) Vn = 4.052d+4 ! Angstroms cubed / molecule (gas at 298 K & 1 atm)
Program 3: Vapor Methane	All parameters the same as program 2 except: maxstp = 200000 ! Number of time steps during data production dt = 10.0d0 ! size of time step (fs) kmsd = 200 ! position save for mean square displacement

```
 initially we have        17604 neighbor pairs
**********          equilibration Completed **********
property             instant       average       standard deviation
Kinetic Energy (aJ)      0.67100076E+00   0.67100076E+00   0.00000000E+00
Potential Energy (aJ)   -0.14486458E+01  -0.14455271E+01   0.31280345E-01
Total Energy (aJ)       -0.77764502E+00  -0.77452638E+00   0.31280345E-01
Temperature (K)          0.15000000E+03   0.15000000E+03   0.00000000E+00
x-Momentum              -0.65851077E-13  -0.14548397E-13   0.36523879E-13
y-Momentum              -0.90140882E-13  -0.31955860E-13   0.26774874E-13
z-Momentum               0.15271385E-12   0.89562192E-13   0.92071565E-13
Pressure aJ/Angstorm^3  -0.30974931E-05  -0.46256192E-05   0.75595203E-05
**********          production    Completed **********
property             instant       average       standard deviation
Kinetic Energy (aJ)      0.66119487E+00   0.67063331E+00   0.20432851E-01
Potential Energy (aJ)   -0.14387403E+01  -0.14482998E+01   0.20474032E-01
Total Energy (aJ)       -0.77754540E+00  -0.77766651E+00   0.11492026E-03
Temperature (K)          0.14780793E+03   0.14991786E+03   0.45676962E+01
x-Momentum               0.33849764E-12   0.28152058E-12   0.18043265E-12
y-Momentum              -0.47732644E-12  -0.22460318E-12   0.16430488E-12
z-Momentum              -0.63552066E-12  -0.13384061E-12   0.27130997E-12
Pressure aJ/Angstorm^3  -0.10317413E-05  -0.40672303E-05   0.49255311E-05
 Program has used   664.876074671745     seconds of CPU time.
```

Program Output 1: liquid methane MD simulation results.

```
 initially we have           0 neighbor pairs
**********          equilibration Completed **********
property             instant       average       standard deviation
Kinetic Energy (aJ)      0.13330548E+01   0.13330548E+01   0.00000000E+00
Potential Energy (aJ)   -0.26195503E-02  -0.25881818E-02   0.19531099E-02
Total Energy (aJ)        0.13304353E+01   0.13304667E+01   0.19531099E-02
Temperature (K)          0.29800000E+03   0.29800000E+03   0.00000000E+00
x-Momentum              -0.98083446E-13   0.28504112E-13   0.88736947E-13
y-Momentum               0.93635610E-13   0.50912990E-14   0.35509657E-13
z-Momentum               0.23278933E-13   0.29206684E-13   0.42819568E-13
Pressure aJ/Angstorm^3   0.10350216E-06   0.10146422E-06   0.97710909E-09
**********          production    Completed **********
property             instant       average       standard deviation
Kinetic Energy (aJ)      0.13375729E+01   0.13338900E+01   0.20282717E-02
Potential Energy (aJ)   -0.71625785E-02  -0.34632438E-02   0.20315744E-02
Total Energy (aJ)        0.13304103E+01   0.13304268E+01   0.14073403E-04
Temperature (K)          0.29900999E+03   0.29818671E+03   0.45341344E+00
x-Momentum              -0.90171208E-12  -0.36633471E-12   0.27337420E-12
y-Momentum              -0.11067602E-12   0.29115486E-12   0.14100432E-12
z-Momentum              -0.27495713E-12  -0.12811391E-12   0.16597744E-12
Pressure aJ/Angstorm^3   0.10153409E-06   0.10146684E-06   0.10622152E-08
 Program has used   144.187327086926     seconds of CPU time.
```

Program Output 2: vapor methane MD simulation results.

In order to obtain self-diffusion coefficients, we ran the code get_diff.f on the mean square displacement output file of simulations 1-3, and generated the output given in get_diff.f outputs 1-3.

We do not know what value to set for the amount of data points to skip before the MSD behavior is satisfactorily close to the long time limit. Therefore, we can set the low limit of time origins, N_{min}, to 1. This will include all the data. Using this value, we run get_diff.f. The output for the self-diffusion coefficient is meaningless, but we have generated a data file, get_diff.out, which has the mean square displacements as a function of time. We plot this data in Figure 1. It appears linear. However, if we plot it on a log-log scale, we can get a better feeling for the data.

In Figure 2, we show the log-log behavior of the mean square displacement data in Figure 1. We can see three regions of behavior. In region one, we have "free motion". This regime occurs at very short observations times before any collisions have occurred. Here the mean square displacement is proportional to the observation time squared.

The second regime shown in Figure 2, is an intermediate time regime, where the mean square displacement is proportional to the observation time raised to some power between 1 and 2. The third regime is the long time behavior where the mean square displacement is proportional to the observation time.

In fitting the mean square displacement data to Einstein's relation, we only want to use data points in region 3. Thus we use a plot like Figure 2 to determine the value of N_{min}.

For program 1, where we simulated liquid methane, we see clearly that by 10,000 fs we are in the long time limit. We ran our simulation for 50,000 steps @ 2 fs/step, or 100,000 fs. So we see that 90% of our simulation can yield reasonable mean square displacements. It doesn't look like this from the plot but that is because the plot is on a log scale.

We obtain N_{min} by converting the minimum observation time to saved data intervals, where

$$N_{min} = \frac{time_{min}}{k_{msd}\Delta t} \tag{5}$$

In this example, our time step was 2 fs and MSD were saved every 100 steps, so for a minimum time of 10,000 fs, we have $N_{min} = 50$. The output for the diffusivity calculation is shown below.

```
 ntime =       1001 ndata =      216216
 read all the data
x  slope =  0.54545284E-02 y-intercept =  -0.13798473E+02 A^2/fs
y  slope =  0.48132025E-02 y-intercept =  -0.10418087E+02 A^2/fs
z  slope =  0.49110663E-02 y-intercept =  -0.53264983E+01 A^2/fs
x  diffusivity avg =  0.27272642E-07 stand dev =  0.18144673E-10 m^2/sec
y  diffusivity avg =  0.24066012E-07 stand dev =  0.33133902E-10 m^2/sec
z  diffusivity avg =  0.24555332E-07 stand dev =  0.10730716E-10 m^2/sec
avg diffusivity avg =  0.25297995E-07 stand dev =  0.17275064E-08 m^2/sec
```

get_diff.f Output 1.a: liquid methane diffusion results. $N_{min} = 50$.

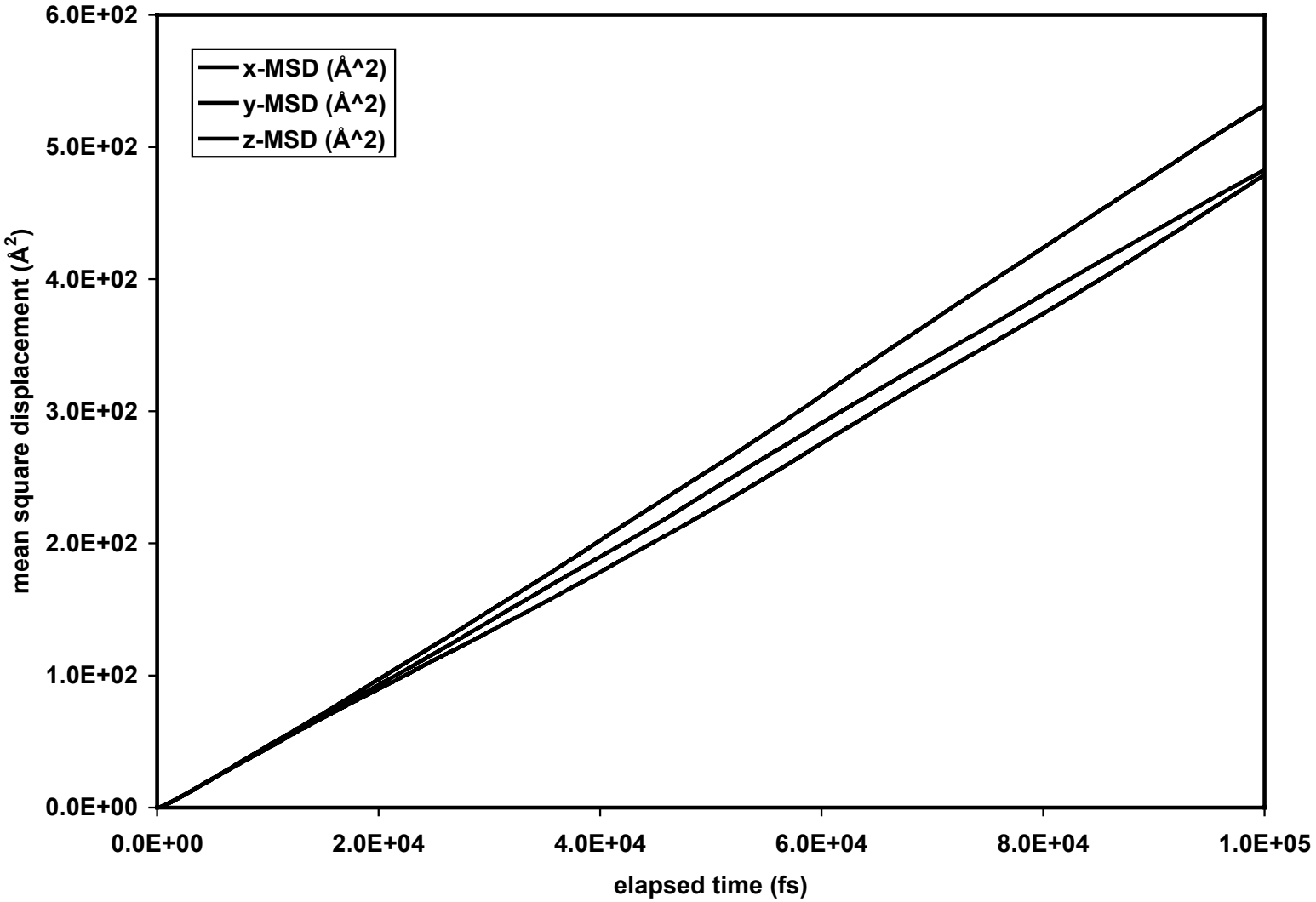

Figure 1. Mean square displacement as a function of observation time for liquid methane of program output 1.

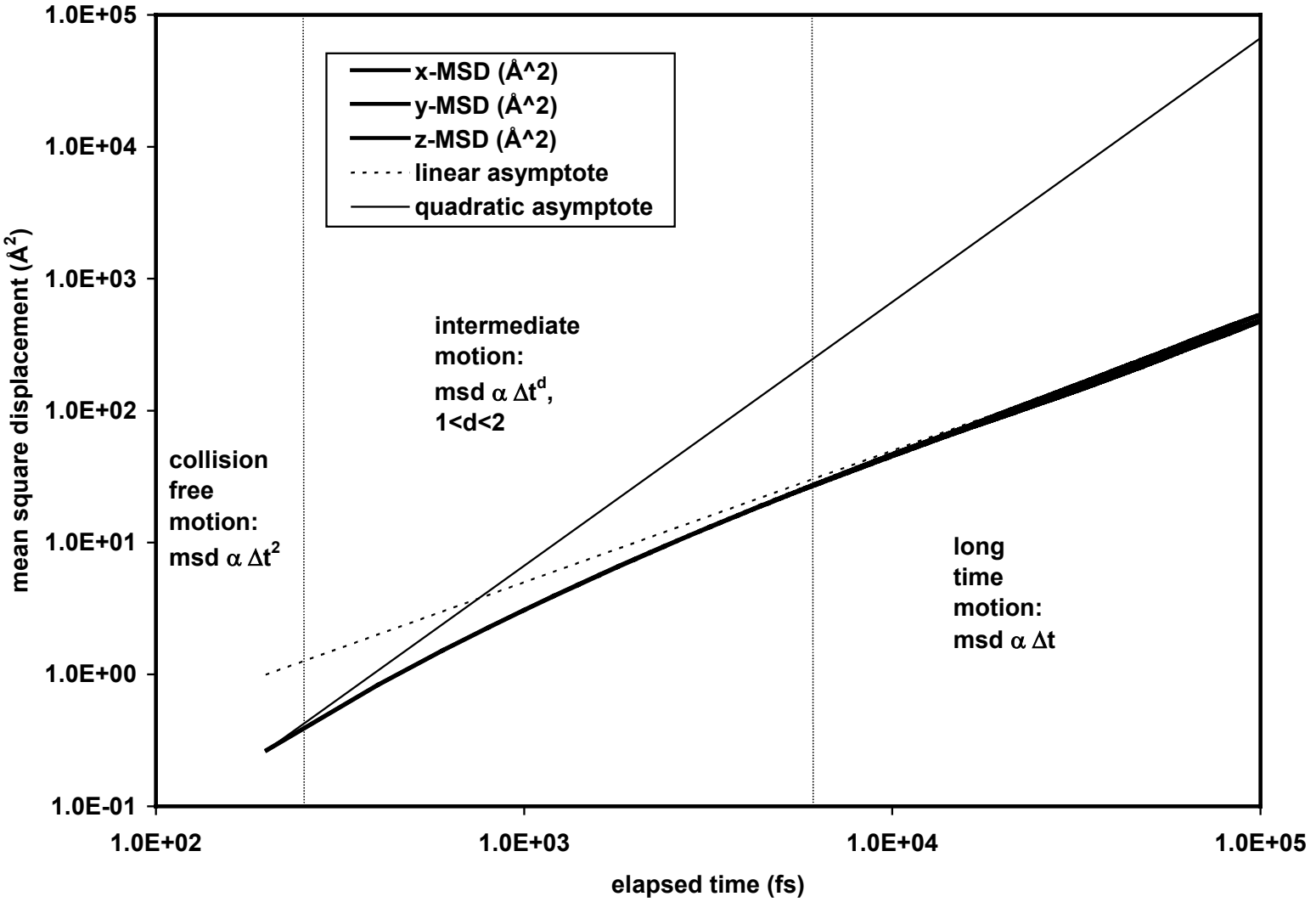

Figure 2. Log-log plot of mean square displacement as a function of observation time for liquid methane of program output 1.

We can make several observations from this program output. The first observation is that the average self-diffusion coefficient is 2.53×10^{-8} m^2/sec. The standard deviation of the self-diffusion coefficient is 1.7×10^{-9} m^2/sec or 6.7 %.

The second observation we can make is that the fluid is supposed to be isotropic. Therefore, the x, y, and z components of the diffusion should all be equal. Their variation is used to calculate the standard deviation.

To determine the importance of the selection of N_{min}, we can run the program where we use all of the data points to generate the self-diffusion coefficient (e.g. $N_{min} = 1$). This output is shown below. In this case we find that the average self-diffusion coefficient is 2.51×10^{-8} m^2/sec. The standard deviation of the self-diffusion coefficient is 1.7×10^{-9} m^2/sec. The difference in the diffusion coefficients calculated using two different values of N_{min} is 0.8 %. This difference is not so big here but it can become larger for gases, as we shall see below.

```
 ntime =       1001 ndata =      216216
 read all the data
x  slope =  0.54074507E-02 y-intercept =  -0.10580693E+02 A^2/fs
y  slope =  0.47705983E-02 y-intercept =  -0.74992496E+01 A^2/fs
z  slope =  0.48971462E-02 y-intercept =  -0.43751128E+01 A^2/fs
x  diffusivity avg =  0.27037254E-07 stand dev =  0.23114807E-10 m^2/sec
y  diffusivity avg =  0.23852992E-07 stand dev =  0.31271477E-10 m^2/sec
z  diffusivity avg =  0.24485731E-07 stand dev =  0.10210160E-10 m^2/sec
avg diffusivity avg =  0.25125325E-07 stand dev =  0.16857229E-08 m^2/sec
```

get_diff.f Output 1.b: liquid methane diffusion results. $N_{min} = 1$.

We can repeat these calculations for the gas phase methane. The mean square displacement as a function of time is shown in Figure 3. The log-log plot is shown in Figure 4. The MSD is clearly a nonlinear function of time in Figure 3. This difference is magnified in Figure 4. The conclusion is that we need a longer time to capture the long time diffusive behavior of a gas than we do for a liquid. This is because the mean free path of a gas is much longer than that of a liquid.

We can run get_diff.f on this simulation output, even though we know that it is not meaningful. It may be useful to compare with longer simulations.

```
 ntime =       1001 ndata =      216216
 read all the data
x  slope =  0.12306509E+01 y-intercept =  -0.18700921E+05 A^2/fs
y  slope =  0.12518016E+01 y-intercept =  -0.19691092E+05 A^2/fs
z  slope =  0.12775410E+01 y-intercept =  -0.20132713E+05 A^2/fs
x  diffusivity avg =  0.61532545E-05 stand dev =  0.58786747E-07 m^2/sec
y  diffusivity avg =  0.62590082E-05 stand dev =  0.64033762E-07 m^2/sec
z  diffusivity avg =  0.63877050E-05 stand dev =  0.65689699E-07 m^2/sec
avg diffusivity avg =  0.62666559E-05 stand dev =  0.11740431E-06 m^2/sec
```

get_diff.f Output 2: vapor methane diffusion results (short simulation).

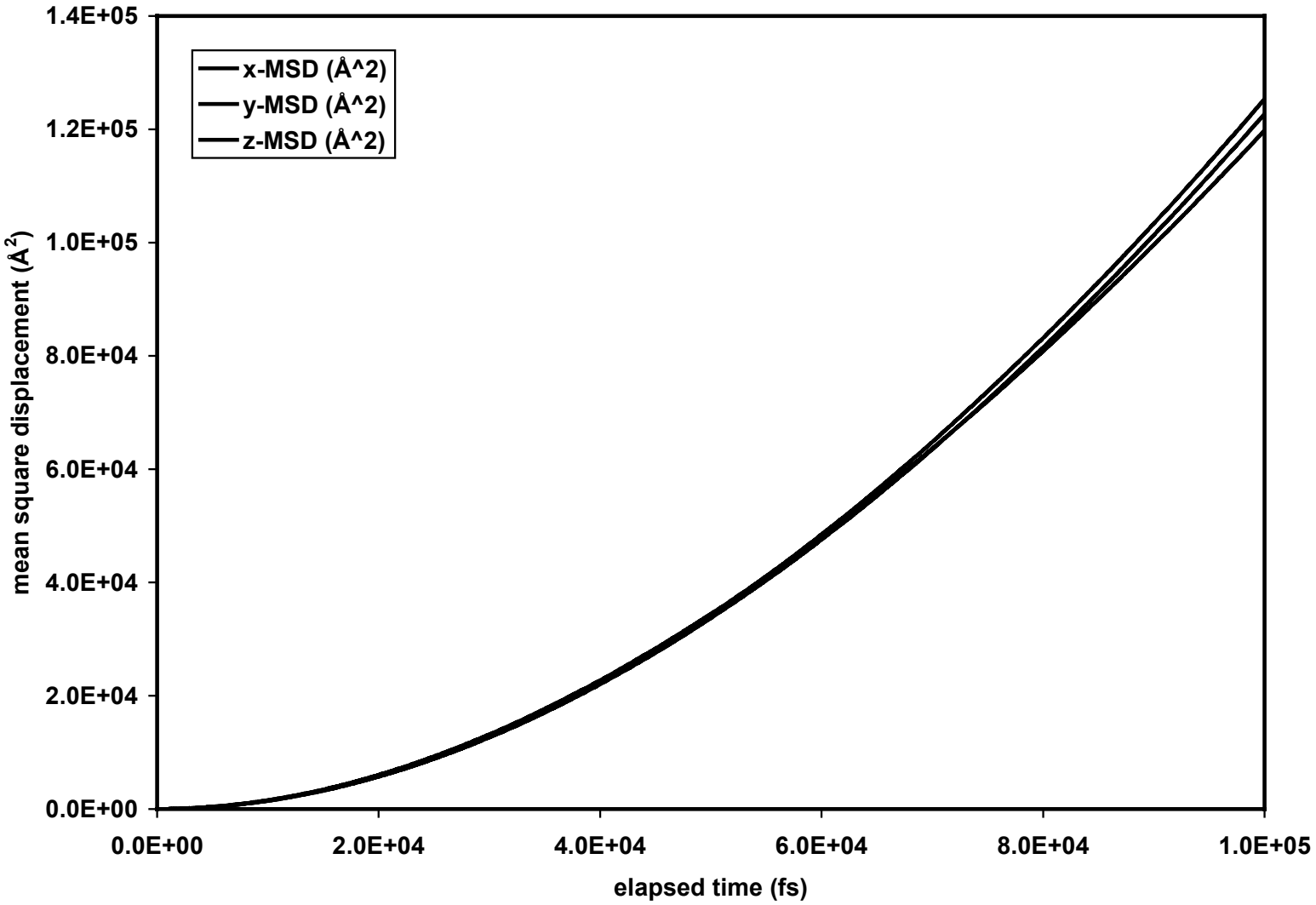

Figure 3. Mean square displacement as a function of observation time for vapor methane of program output 2.

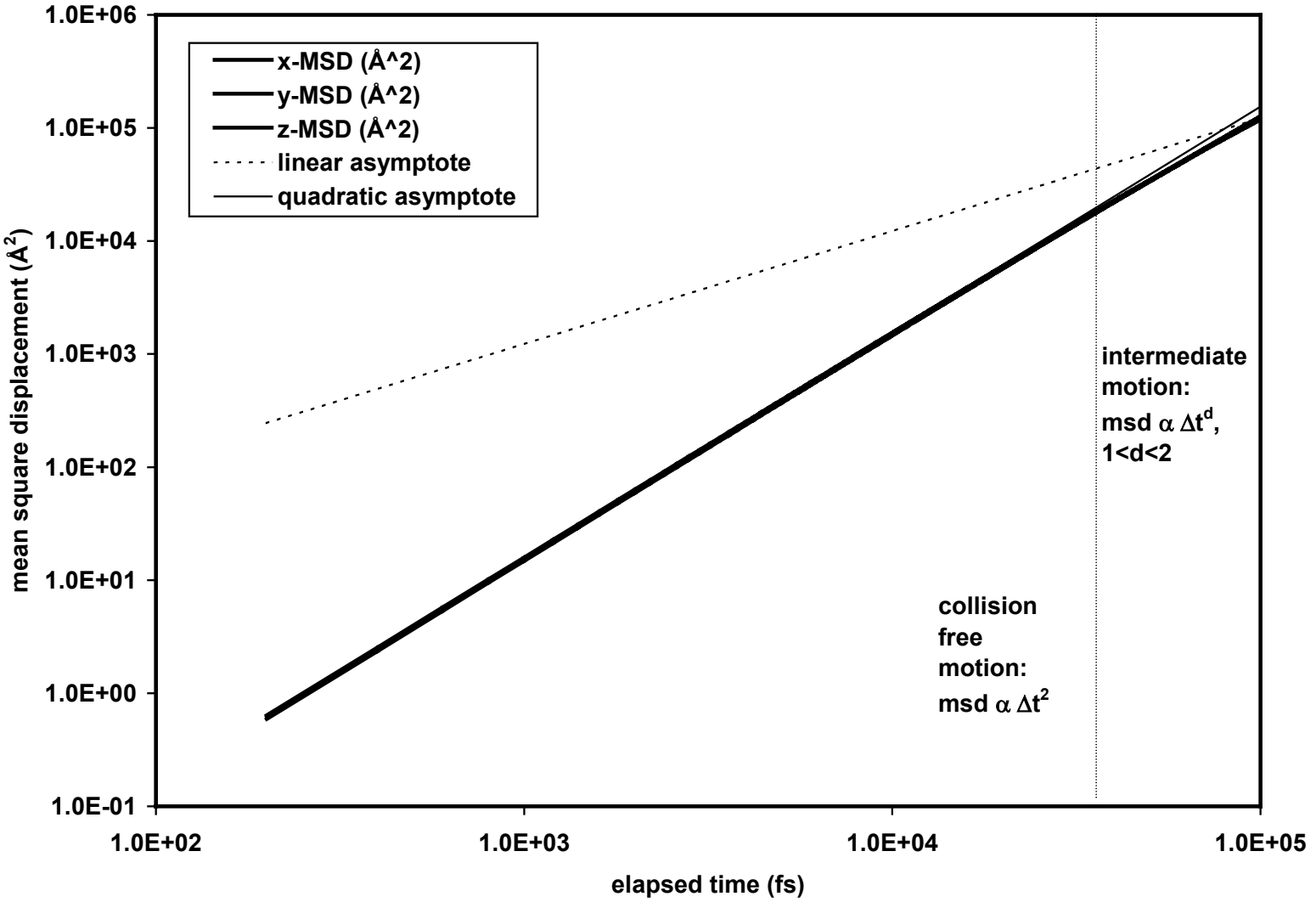

Figure 4. Log-log plot of mean square displacement as a function of observation time for vapor methane of program output 2.

We repeat the gas phase simulation for 2 ns instead of 0.2 ns. The mean square displacement as a function of time is shown in Figure 5. The log-log plot is shown in Figure 6. We see that the MSD has reached long time behavior at 500,000 fs. In this example, our time step was 10 fs and MSD were saved every 200 steps, so for a minimum time of 500,000 fs, we have $N_{min} = 250$.

```
 ntime =       1001 ndata =      216216
 read all the data
x  slope =  0.42801023E+01 y-intercept =  -0.49714112E+06 A^2/fs
y  slope =  0.43171521E+01 y-intercept =  -0.47808680E+06 A^2/fs
z  slope =  0.39884563E+01 y-intercept =  -0.26316716E+06 A^2/fs
x  diffusivity avg =  0.21400512E-04 stand dev =  0.23095574E-07 m^2/sec
y  diffusivity avg =  0.21585761E-04 stand dev =  0.90838446E-08 m^2/sec
z  diffusivity avg =  0.19942281E-04 stand dev =  0.44566287E-07 m^2/sec
avg diffusivity avg =  0.20976185E-04 stand dev =  0.90014598E-06 m^2/sec
```

get_diff.f Output 3: vapor methane diffusion results (long simulation).

In this case we find that the average self-diffusion coefficient is 2.10×10^{-5} m^2/sec. The standard deviation of the self-diffusion coefficient is 9.0×10^{-7} m^2/sec. The short duration run of the same simulation produced a self-diffusion coefficient is 6.27×10^{-5} m^2/sec. The relative error is 199%. It is important to make sure that you run the simulations long enough to reach the long time behavior. Moreover, it is important that you only include sufficiently long observation times in the linear regression of the self-diffusion coefficient.

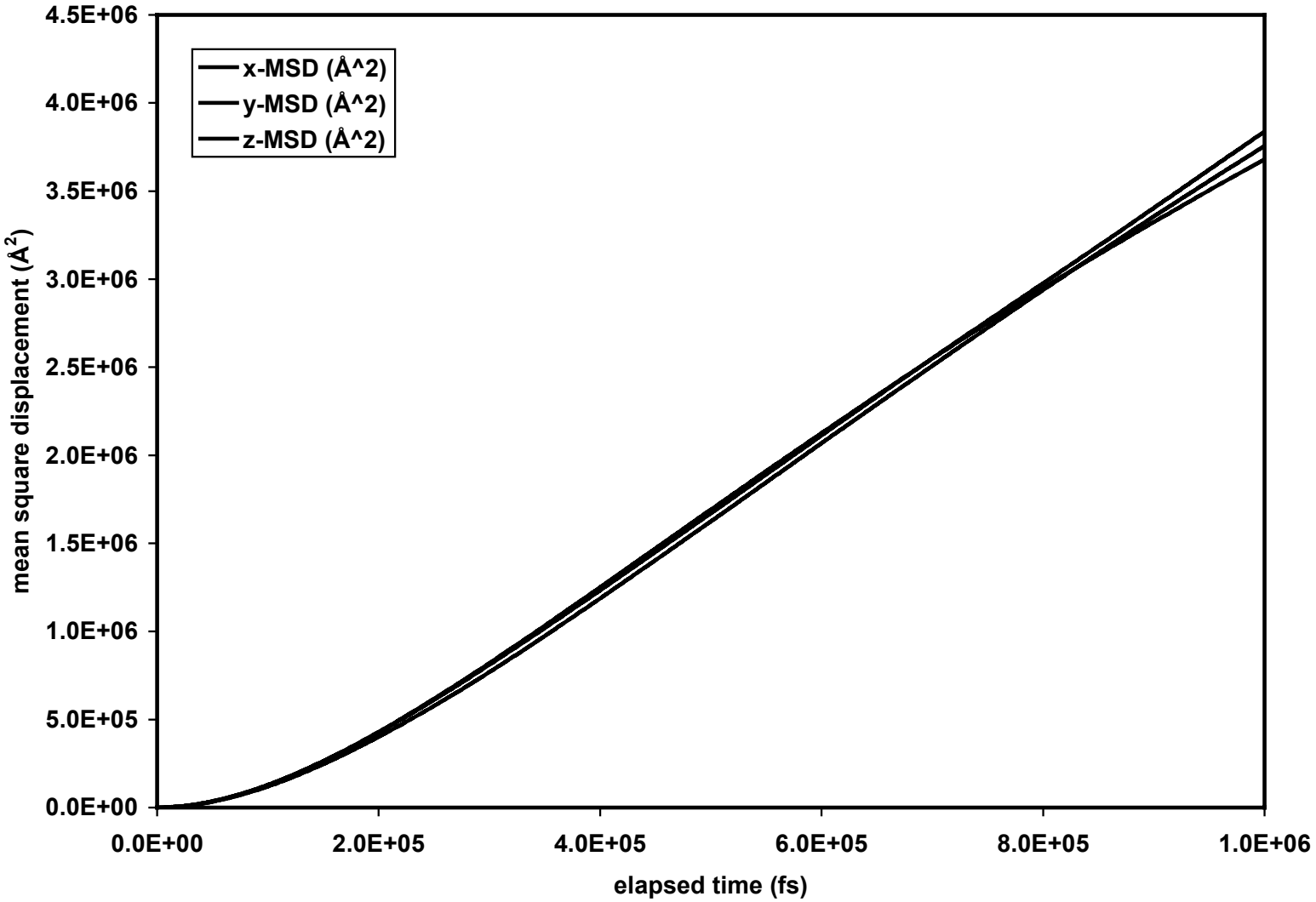

Figure 5. Mean square displacement as a function of observation time for vapor methane of program output 3.

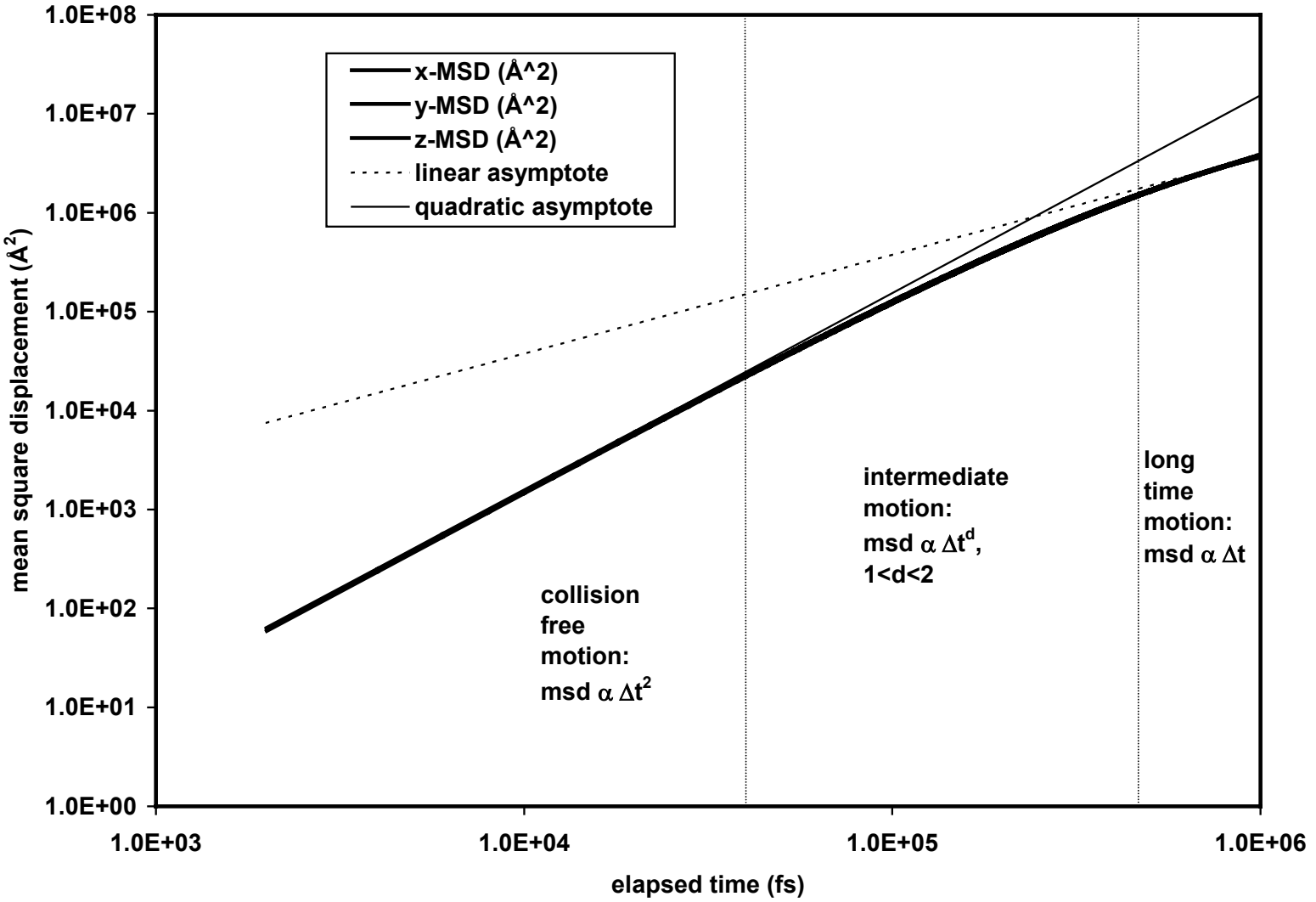

Figure 6. Log-log plot of mean square displacement as a function of observation time for vapor methane of program output 3.

References.

1. Kärger, J., Ruthven, D.M., "Diffusion in Zeolites and Other Microporous Solids", John Wiley & Sons, Inc., New York, 1992.
2. Haile, J.M., "Molecular Dynamics Simulation", John Wiley & Sons, Inc., New York, 1992.
3. Allen, M.P., Tildesley, D.J., "Computer Simulation of Liquids", Oxford Science Publications, Oxford, 1987.
4. Frenkel, D., Smit B., "Understanding Molecular Simulation", Academic Press, San Diego, 1996.
5. Keffer, D., "Applied Statistical and Numerical Methods for Engineers", Course Website, http://clausius.engr.utk.edu/che301, University of Tennessee, Knoxville, 1998-2001.
6. de Groot, S.R., Mazur, P., "Non-Equilibrium Thermodynamics", North Holland Publishing Co., Amsterdam, 1962.

Appendix A. Self Diffusion Code in Fortran

```
      program get_diff
c
c     This program will calculate diffusivities
c    from mean square displacement data
c
c     author  David Keffer
c     Department of Chemical Engineering
c     University of Tennessee, Knoxville
c     last updated  October 6, 2001
c
      integer, parameter :: maxstp = 100000      ! number of data production steps
      integer, parameter :: kmsd = 100           ! sampling interval
      integer, parameter :: N = 216                    ! number of molecules
      double precision, parameter  :: dt = 2.d0 ! timestep (fs)
      character*12 :: cmsd, cout          ! character variables
      character*3, dimension(1:4) :: cname
      double precision, dimension(1:4) :: Dav, Dsd
      double precision, dimension(1:3) :: slope,slopesd,yinter,yintersd
      double precision, allocatable :: md_msd(:,:), time_vec(:),
   & xmsd(:,:)
c
      cout = 'get_diff.out'
      cmsd = 'md_msd.out'
c     number of times represented in data
      ntime = maxstp/kmsd + 1
c     number of rows of data
      ndata = N*ntime
      allocate (md_msd(1:ndata,1:3))
      open(unit=1,file=cout,form='formatted',status='unknown')
      open(unit=2,file=cmsd,form='formatted',status='old')
      print *, ' ntime = ', ntime, ' ndata = ', ndata
      do i = 1, ndata, 1
            read(2,*) md_msd(i,1:3)
      enddo
      print *, ' read all the data'
c     number of origins is half number of time steps
      norigin = (ntime-1)/2
c     minimum number of intervals to contribute to diffusivity
      nmin = 50
c     maximum number of intervals to contribute to diffusivity
      nmax = norigin
c
      if (nmin .gt. nmax) then
            print *, ' We have a problem. '
            print *, ' nmin = ', nmin, ' nmax = ', nmax
            stop
      endif
c     store mean square displacements in xmsd
      allocate (time_vec(1:norigin), xmsd(1:norigin,1:3))
      do i = 1, norigin, 1
            time_vec(i) = dfloat(i*kmsd)*dt
      enddo
```

```
      xmsd = 0.d0
      do i = 1, N, 1
            do j = 1, norigin, 1
                  jstart = (j-1)*N + i
                  do k = nmin, nmax, 1
                        kend = jstart + k*N
                        xmsd(k,1:3) = xmsd(k,1:3) +
     &                  (md_msd(kend,1:3) - md_msd(jstart,1:3) )**2
                  enddo
            enddo
      enddo
      xmsd = xmsd/dfloat(N*norigin)
c
c     perform a linear least squares regression
c
      do i = 1, 3, 1
            call dllsr(slope(i), slopesd(i), yinter(i), yintersd(i),
     &    nmax-nmin+1, time_vec(nmin:nmax), xmsd(nmin:nmax,i))
      enddo
c
c     report results
c
      cname(1) = 'x  '
      cname(2) = 'y  '
      cname(3) = 'z  '
      cname(4) = 'avg'
      do i = 1, 3, 1
            write(6,1007) cname(i), slope(i), yinter(i)
            write(1,1007) cname(i), slope(i), yinter(i)
      enddo
 1007 format(a3, ' slope = ', e16.8, ' y-intercept = ', e16.8,
     & ' A^2/fs')
      Dav(1:3) = 0.5d0*slope(1:3)*1.0d-5 ! convert to m^2/sec
      Dsd(1:3) = 0.5d0*slopesd(1:3)*1.0d-5 ! convert to m^2/sec
      Dav(4) = sum(Dav(1:3))/3.d0
c     standard deviation of average diffusivity
      term1 = 3.d0*(Dav(1)*Dav(1) + Dav(2)*Dav(2) + Dav(3)*Dav(3) )
      Dsd(4) = sqrt( (term1 - Dav(4)*Dav(4)*9.d0) /6.d0 )
      do i = 1, 4, 1
            write(6,1006) cname(i), Dav(i), Dsd(i)
            write(1,1006) cname(i), Dav(i), Dsd(i)
      enddo
 1006 format(a3, ' diffusivity avg = ', e16.8, ' stand dev = ', e16.8,
     & ' m^2/sec ')
c
c     write xmsd vs time data for later plotting
c
      do i = 1, norigin, 1
            write(1,1008) time_vec(i), xmsd(i,1:3)
      enddo
 1008 format(4(e16.8,1x))
      close (unit=1,status='keep')
      close (unit=2,status='keep')
      stop
      end
```

```
subroutine dllsr(slope, slopesd, yinter, yintersd, n, x, y)
implicit double precision (a-h, o-z)
double precision, intent(out) :: slope, slopesd, yinter, yintersd
integer, intent(in) :: n
double precision, intent(in), dimension(1:n) :: x, y
xn = dfloat(n)
xavg = sum(x)/xn
yavg = sum(y)/xn
sumxy = 0.d0
sumxx = 0.d0
sumx2 = 0.d0
do i = 1, n, 1
        sumxy = sumxy + (x(i) - xavg)*(y(i) - yavg)
        sumxx = sumxx + (x(i) - xavg)*(x(i) - xavg)
        sumx2 = sumx2 + x(i)*x(i)
enddo
slope = sumxy/sumxx
yinter = yavg - slope*xavg
sse = 0.d0
do i = 1, n, 1
        sse = sse + (y(i) - slope*x(i) - yinter)**2.d0
enddo
sig2 = sse/dfloat(n-2)
slopesd = dsqrt(sig2/sumxx)
yintersd = dsqrt(sig2/dfloat(n)*sumx2/sumxx)
return
end
```

Appendix B. Self Diffusion Code in MATLAB

```
function get_diff

maxstp = 2000; % number of data production steps
kmsd = 10;     % sampling interval
N = 27;        % number of molecules
dt = 1.0;      % timestep (fs)

load('md_msd.out')
ndata = length(md_msd)

% number of times represented in data
ntime = ndata/N;

% number of origins is half number of time steps
norigin = floor(ntime-1)/2;

% minimum number of intervals to contribute to diffusivity
nmin = 50;
% maximum number of intervals to contribute to diffusivity
nmax = norigin;
if (nmin > norigin)
   nmin = nmax;
end

% store mean square displacements in xmsd
time_vec= [dt:dt:norigin*dt]';
xmsd = zeros(norigin,3);
for i = 1:1:N
   for j = 1:1:norigin
      jstart = (j-1)*N + i;
      for k = nmin:1:nmax
         kend = jstart + k*N;
         %term = (md_msd(kend,1) - md_msd(jstart,1) ).^2;
         %fprintf(1, 'i %i j %i jstart %i k %i kend %i term %e  %e %e\n',i,j,jstart,k,kend, term,
md_msd(kend,1), md_msd(jstart,1));
         xmsd(k,1:3) = xmsd(k,1:3) + (md_msd(kend,1:3) - md_msd(jstart,1:3) ).^2;
      end
   end
end
xmsd = xmsd/(N*norigin);
```

```
[Px,Sx] = POLYFIT(time_vec(nmin:1:nmax),xmsd(nmin:1:nmax,1),1);
[Py,Sy] = POLYFIT(time_vec(nmin:1:nmax),xmsd(nmin:1:nmax,2),1);
[Pz,Sz] = POLYFIT(time_vec(nmin:1:nmax),xmsd(nmin:1:nmax,3),1);
fprintf(1, 'x:  slope = %e intercept = %e \n',Px);
fprintf(1, 'y:  slope = %e intercept = %e \n',Py);
fprintf(1, 'z:  slope = %e intercept = %e \n',Pz);

D(1) = 0.5*Px(1)*1.0e-5; % convert to m^2/sec
D(2) = 0.5*Py(1)*1.0e-5; % convert to m^2/sec
D(3) = 0.5*Pz(1)*1.0e-5; % convert to m^2/sec
Dav = sum(D)/3;
sd = zeros(3,1);
sdav = 0.0;
fprintf(1, 'x: diffusivity  average = %e stand dev = %e  m^2/sec\n',D(1),sd(1));
fprintf(1, 'y: diffusivity  average = %e stand dev = %e  m^2/sec\n',D(2),sd(2));
fprintf(1, 'z: diffusivity  average = %e stand dev = %e  m^2/sec\n',D(3),sd(3));
fprintf(1, 'avg: diffusivity  average = %e stand dev = %e  m^2/sec\n',Dav,sdav);


xmod(:,1) = Px(2) + Px(1)*time_vec;
xmod(:,2) = Py(2) + Py(1)*time_vec;
xmod(:,3) = Pz(2) + Pz(1)*time_vec;
figure(1)
plot(time_vec,xmsd(:,1),'k-');
hold on;
plot(time_vec,xmsd(:,2),'r-');
hold on;
plot(time_vec,xmsd(:,3),'b-');
hold on;
plot(time_vec,xmod(:,1),'k:');
hold on;
plot(time_vec,xmod(:,2),'r:');
hold on;
plot(time_vec,xmod(:,3),'b:');
hold off;
legend('x: sim', 'y: sim', 'z: sim','x: mod', 'y: mod', 'z: mod');
ylabel('mean square displacement (Angstroms^2)');
xlabel('time (fs)');


figure(2)
time_vec2= [0:dt:(ntime-1)*dt]';
plot(time_vec2,md_msd(1:N:ndata,1),'k-');
```

```
hold on;
plot(time_vec2,md_msd(1:N:ndata,2),'r-');
hold on;
plot(time_vec2,md_msd(1:N:ndata,3),'b-');
hold off;
legend('x ', 'y ', 'z ');
ylabel(' atom 1 position (Angstroms)');
xlabel('time (fs)');
```