Lawrence Berkeley National Laboratory
Recent Work

Title
Vibrational Relaxation in Liquids: Comparisons between Gas Phase and Liquid Phase Theories

Permalink
https://escholarship.org/uc/item/9bb475fr

Author
Russell, D.J.

Publication Date
1990-11-01

eScholarship.org Powered by the California Digital Library
University of California

LBL-30000

Lawrence Berkeley Laboratory

UNIVERSITY OF CALIFORNIA

Materials & Chemical Sciences Division

Vibrational Relaxation in Liquids: Comparisons between Gas Phase and Liquid Phase Theories

D.J. Russell
(Ph.D. Thesis)

November 1990

LOAN COPY
Circulates
for 2 weeks
Bldg. 50 Library.
LBL-30000
Copy 2

Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

LBL-30000

Vibrational Relaxation in Liquids: Comparisons between Gas Phase and Liquid Phase Theories

Daniel John Russell

Ph.D. Thesis

Department of Chemistry
University of California

and

Chemical Sciences Division
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720

December 1990

This work was supported by the National Science Foundation, the San Diego Supercomputer Center, and the U.S. Department of Energy, Office of Basic Sciences, Chemical Sciences Division under Contract No. DE-AC03-76SF00098.

Vibrational Relaxation in Liquids:
Comparisons between Gas Phase and Liquid Phase Theories

By

Daniel John Russell

Abstract

The vibrational relaxation of iodine in liquid xenon was studied to understand what processes are important in determining the density dependence of the vibrational relaxation. This examination will be accomplished by taking simple models and comparing the results to both experimental outcomes and the predictions of molecular dynamics simulations. The vibration relaxation of iodine is extremely sensitive to the iodine potential. The anharmonicity of iodine causes vibrational relaxation to be much faster at the top of the iodine well compared to the vibrational relaxation at the bottom. For this reason, models that use theories such as Schwartz, Slawsky, and Herzfeld equation to describe vibrational relaxation can not be expected to describe the relaxation in the top of the iodine well. Models that incorporate the anharmonicity by actually calculating the probability of relaxation vs. vibrational level are qualitatively insensitive to the actual potential used and reproduce the actual experimental results faithfully.

A number of models are used in order to test the ability of the Isolated Binary Collision theory's ability to predict the density dependence of the vibrational relaxation of iodine

in liquid xenon. The models tested vary from the simplest incorporating only the fact that the solvent occupies volume to models that incorporate the short range structure of the liquid in the radial distribution function. None of the models tested do a good job of predicting the actual relaxation rate for a given density. This may be due to a possible error in the choice of potentials to model the system. The models tested do a reasonable job predicting the density dependence given the relaxation rate at one density. The reason for this discrepancy between the error in predicting the actual rate vs the positive results in predicting density dependencies suggests that the rate depends strongly on potentials while density dependence is determined by the structure of the liquid, which for high densities is only sensitive to the size of the solvent.

Acknowledgements

The time I have spent at Berkeley has been filled with many knowledgeable and friendly people. Without their help and friendship, life as a graduate student would have been brutish and long. I can't thank everyone here, so I will thank the people who have been most essential to my happiness and graduate training.

I would like to thank my parents, who have had the strongest influence on my life. My brother Gerard has also helped me in every way, from moral to financial support when needed. Without my family I probably would have not gone to graduate school.

Life is not all books and experiments. John and Tina have not let me lose contact with my friends in Chicago, and are always willing to listen to my complaints and make holidays happy by sending goofy packages.

My life has been very happy for the last few years due to Eva. She provided me with a constant link to the non-scientific world and has given me much joy and happiness with her understanding and ability to ignore my argumentativeness. The days have gone by much faster since I met Eva. I also want to thank Justine and Tom, who make every Friday a rousing political debate, and Vince and Susan who have given me some mini vacations, and tried to teach me bridge.

The Harris group has been a very good place to learn and work. I have learned a lot from Keenan Brown, who through

intuition and mathematical ability has always been helpful, even when I have bothered him at his new job. Mark Paige taught me an experimental style that I think will be useful in any problem in life. Dave Smith was also willing to discuss this work at any time and was quite helpful in my work generally and in particular the papers we worked on together. Karen Schultz has always been willing to read something for me and give good advice on the many mistakes I make scientifically or otherwise. I also appreciate my office mates over the years for the good working and clothes changing environment they provided. I would also like to thank Walter, Eric, Jennifer, Jason, Steve, David, Jin, Robert, and Dee for helping me make Roma's rich.

Of course if I wanted to get anything done here it would have been impossible without Vijaya Narasimhan. Vijaya knew all the ins and outs of dealing with the bureaucratic monolith of Berkeley and was also a friend.

Finally I would like to thank Charles Harris for providing intellectual and financial support for my graduate training. I would not have studied any of the models discussed here without his suggestions. Finally this work was supported by the National Science Foundation. The calculations were performed at the San Diego Supercomputer Center. I would also like to acknowledge the U.S. Department of Energy, Office of Basic Sciences, Chemical Sciences Division under Contract No. DE-AC03-76SF00098, for some

specialized equipment used in this research.

Table of Contents

I. Introduction 1

II. Review of Isolated Binary Collision Theory and Iodine Geminate Recombination 2

A. Introduction 2

B. Iodine Recombination 3

C. Isolated Binary Collision Theory 8

III. Application of Isolated Binary Collision Theory to the Vibrational Relaxation of Iodine in Liquid Xenon 16

A. Calculation of Relaxation Probabilities using SSH. 16

B. Taking account of anharmonicity. 24

C. Calculations of the Density Dependence. . . . 40

IV. Summary and Conclusions 76

References 80

V. APPENDIX 83

A. PROGRAM LISTING I2IBC 83

B. PROGRAM LISTING IBCENE 101

C. PROGRAM LISTING IIBC 105

D. PROGRAM LISTING MAIN 127

E. PROGRAM LISTING MULTI 162

F. PROGRAM LISTING CAMAC 170

G. PROGRAM LISTING CONST 172

H. PROGRAM LISTING ERROR 178

I. PROGRAM LISTING PLOT5 181

J. PROGRAM LISTING COMM 188

K. PROGRAM LISTING DATA 191

L. PROGRAM LISTING LOOK 195

M. PROGRAM LISTING NICE 206

N. PROGRAM LISTING SAVE 227

I. Introduction

Chemistry has been divided into many subfields of interest from biochemistry to physical chemistry. Although the specific areas of interest are different, the bond between all of the subfields is the direct or indirect study of chemical reactions. Physical chemistry is more the study of how or why certain reactions occur, and not the inventory of the reactions. Since the advent of spectroscopy there have been many advances in the realization of many static properties of molecules, from the electronic states it possesses, to the relative coordinates of the nuclei. This allows one to predict the products of reactions using thermodynamics. However this does not provide enough information at this time to predict reaction rates or how to control which products are produced.

The introduction of lasers that produced very short laser pulses allowed physical chemists to trace the actual reaction and through what states the molecule went. With the information of how the molecule moves along a reaction coordinate it may be possible to change the final outcome of the reaction by perturbing the path of the wave packet or which potential a wave packet is on. This will only be possible, if it is possible at all, by investigating the time

scale of these processes and understanding the mechanisms that determine the pathways.

II. Review of Isolated Binary Collision Theory and Iodine Geminate Recombination

A. Introduction

The photodissociation and geminate recombination of iodine in a liquid has been studied since Rabinowitch and Wood in 1936.[1,2,3] In the photodissociation of iodine the molecule is electronically excited from the ground state to mainly the bound B state. The molecule then collisionally predissociates, at this point the atoms may lose energy and recombine geminately or they may diffuse away. After geminate recombination, vibrational relaxation returns the molecule to it's initial equilibrium state. This simple reaction incorporates three relatively fast processes that are not well understood. The three processes are curve crossing, geminate recombination, and vibrational relaxation. These processes are not well understood for a number of reasons. Because of the fast time scale for geminate recombination, experiments could not be performed until laser technology had produced lasers which probed this time scale with picosecond or shorter pulses. Secondly only with the advent of supercomputers, could people model liquids on the short time scales that were

needed to describe geminate recombination. Before supercomputers most people used hydrodynamic models which treat the liquid as a continuum. However, on very short time scales a liquid does not behave as a continuum and the frequency dependent behavior of the liquid becomes important. This would make a big difference in predicting geminate recombination rates. The process of curve crossing presents the difficult problem of trying to describe a quantum system coupled, sometimes strongly, to a bath with many degrees of freedom. Also the curve crossing process can vary by an order of magnitude in it's rate in the same molecule depending on the states coupled. The problem of vibrational relaxation on the other hand has been studied for many years. The systems studied were usually polyatomics in polyatomic solvents using ultrasound. This allowed people to make qualitative validations of their expectations, but predictions were out of the question due to the complexity of the systems. Only recently have simpler molecules in simple solvents been studied showing that the time scale for relaxation varies over an enormous range. The question of simple models for iodine's vibrational relaxation is the main topic of the following text.

B. Iodine Recombination

The study of geminate recombination yields of iodine

was begun in the 1950's and 1960's by Noyes and co-workers.[4] The attempts to model the geminate recombination yields as two iodine atoms in a continuum liquid, using Brownian motion models or hydrodynamic systems, were not very successful. The first time dependent studies of the geminate recombination of iodine in a liquid, CCl_4, were performed by Eisenthal.[5] They excited the iodine with 532 nm light to the bound B state and watched the transient bleach decay in 150 ps approximately. The data were interpreted in terms of the dissociated iodine atoms diffusing in the solvent and then geminately recombining. Curve crossing from the excited electronic state to the ground state and vibrational relaxation were considered to be very fast. This interpretation was questioned when early molecular dynamics results predicted that recombination did not behave like a diffusional process and should be very fast, less than 5 ps. rather than the 150 ps proposed.[6, 7] Nesbitt and Hynes advanced that the 150 ps. time scale was not geminate recombination but the time scale for vibrational relaxation of the newly recombined iodine.[8] They based there proposal on the calculation that for a bleach at 532 nm., as Eisenthal had performed, that the recombined iodine molecules would absorb light at 532 only after the iodine molecule had vibrationally relaxed. Secondly Nesbitt and Hynes made an order of magnitude calculation of what the time scale of the vibrational relaxation should be, they found that vibrational relaxation could be the time scale associated with the bleach.

To perform their calculation of the vibrational relaxation rate, the assumptions of the Isolated Binary Collision theory were employed. Using the calculation of vibrational relaxation rates and Frank Condon factors, Nesbitt and Hynes predicted that probing at longer wavelengths should probe higher vibrational states, and if the 150 ps. time scale was due to vibrational relaxation, the transient absorptions at longer wavelength would appear sooner and decay faster.

The prediction of fast geminate recombination is consistent with experiments performed by Smith, which are most simply interpreted as a recombination time of less than 1 ps.[9] The prediction of slow vibrational relaxation was confirmed by Harris et al,[10,11,12] however a clear understanding of vibrational relaxation was not forthcoming. No clear trends could be observed that would predict vibrational relaxation rates. The solvents studied at the time were molecular solvents, and there were three possible contributions to the relaxation, vibration to translation, vibration to vibration, and vibration to rotation. Nesbitt and Hynes had to invoke a large rate of iodine vibrational energy relaxing into the vibrational modes of CCL_4 to explain the vibrational relaxation times of iodine in CCl_4.[8] Justifying this by noting the lowest frequency vibrational mode for CCl_4 is 217 cm^{-1} and close enough to iodine's 214 cm^{-1} vibrational mode to couple significantly. Unfortunately this does not seem to be emulated in other chlorinated solvents. In $CHCl_3$ and CH_2Cl_2

the lowest vibrational frequencies are 261 cm^{-1} and 282 cm^{-1} yet CCl_4 has the slowest relaxation time while the vibrational frequency is closest, and Ch_2Cl_2 is the fastest where vibrational coupling arguments indicate it should be the slowest. In order to come to a firmer understanding of vibrational energy transfer this group undertook a three phase study of the relaxation of iodine in liquid xenon, a simple monatomic solvent. A simple monatomic solvent was chosen to eliminate any difficulties due to vibration to vibration coupling. The first phase was the actual experiment,[13] for which Nesbitt and Hynes had predicted approximately one nanosecond for vibrational relaxation due to the lack of vibrational modes in the solvent to couple to. The second phase was a classical molecular dynamics simulation of iodine photodissociating and vibrationally relaxing in liquid xenon.[14] And finally the application of simpler models to try and understand the vibrational relaxation process.

The experimental study of iodine in liquid xenon had been reported by Kelly,[15] but the results were on the same time scale as relaxation in CCL_4. At the time the results were reported it was not surprising, considering the then current interpretation that this was the recombination time. However, after Nesbitt and Hynes' interpretation became the consensus, these results were considered too fast to be vibrational relaxation and therefore surprising. Paige et al reexamined the iodine photodissociation and recombination in liquid xenon

with a laser system that had lower noise and better time resolution, and found the vibrational relaxation to be on the time scale of approximately 5 ns. This was the same order of magnitude as the predictions of Nesbitt and Hynes.[13] The molecular dynamics simulations however gave a time scale of relaxation more in line with Kelly's experimental observations, if interpreted as vibrational relaxation. However the molecular dynamics simulations are sensitive to the interaction potential used for iodine and xenon. If the slope of the potential between iodine and xenon is in reality smaller than the slope of the potential in the simulation the results would be comparable to experiments. This problem can also be seen by the fact that real Xe does not solidify until $\rho^* \approx .94$ even though a Lennard-Jones system solidifies at $\rho^* \approx .9$. Another problem with the molecular dynamics being compared to the experiment is that the system studied by molecular dynamics is not at constant temperature, being a finite volume with periodic boundary conditions, and the temperature rises as the iodine molecule vibrationally relaxes. The temperature of the system has been shown to be a sensitive variable in vibrational relaxation. In order to test a less computationally intensive model a generalized Langevin Equation was also used to model the relaxation, and performed quite well in modeling the classical simulation of the relaxation.[16,17]

Because of the problems described above with regards to

temperature, all the implementations of the IBC models discussed here will be compared primarily with the molecular dynamics simulations and only secondarily with the actual experiments.

C. Isolated Binary Collision Theory

It has been 30 years since the Isolated Binary Collision (IBC) theory was proposed to describe vibrational relaxation in liquids.[18,19] Ultrasonic absorption experiments on various liquids were performed and the results were related to the vibrational relaxation of an oscillator. Experimentally it was found in many cases that the relaxation rate increased linearly with density at low density. As the density increased however, the rate increased nonlinearly with density. Litovitz believed that the results could be explained by making two assumptions. The first assumption was that the relaxation could be explained by the gas phase relaxation equation

$$K(\rho,T)_{ij} = P(T)_{ij} * Z(\rho,T) \qquad (1)$$

Where K_{ij} is the rate of relaxation from vibrational state i to j. P_{ij} is the gas phase probability of changing from vibrational state i to j given a collision with one molecule averaged over oscillator phase, impact parameter and a Maxwell Boltzmann distribution of velocity. Z is the collision frequency. Notice that P_{ij} is only temperature dependent and

Z is density and temperature dependent. The second assumption Litovitz made was that Z is not given by the ideal gas collision rate in the dense phase, but that the volume the molecules take up must be taken into account in calculating the collision rate. An example of one of the collision rate formulas used at the time is

$$Z \approx \frac{V}{\rho^{-\frac{1}{3}} - \sigma} \tag{2}$$

Note that this is the average velocity (V) divided by the mean free path of a molecule in a moving wall cage. One problem with this formulation is that the mean free path should be proportional to ρ^{-1} as $\rho \rightarrow 0$, but where this transition occurs is not defined.[18]

According to early IBC proponents one only had to correctly calculate the collision frequency in order to predict the density dependence of vibrational relaxation. However there were more assumptions than Litovitz and Madigosky stated in the original papers. Fixman and Zwanzig objected that IBC neglected the constant collective random force on the vibration, the possibility that in the liquid the collisions may not be independent due to a non random phase of the oscillator during collisions, and that collisions may overlap in time.[20,21] Fixman modeled the collective random force due to the solvent as a force with a white noise frequency spectrum. His calculation showed that the white noise random force is very efficient at relaxing the

oscillator. However the white noise random force overestimates the high frequency random forces in a liquid, as Fixman noted, and the model's criticism of IBC was rebuffed by Herzfeld.[22] Zwanzig approached the problem from a time correlation function perspective where the rate should be proportional to

$$\int dt \exp(i\omega t) \langle F(t)\ F(0) \rangle \tag{3}$$

where ω is the frequency of the oscillator and F(t) is the total force on the oscillator at time t. Zwanzig then assumed that the force on the oscillator could be decomposed into isolated events at some time t_k.

$$F(t) = \sum_k f(t-t_k) \tag{4}$$

$$\langle F(t)\ F(t+\tau) \rangle = \sum_j \sum_{k \neq j} \langle f(t-t_k)\ f(t+\tau-t_j \rangle + \sum_k \langle f(t-t_k)\ f(t+\tau-t_k) \rangle \tag{5}$$

Where $f(t-t_k)$ is the force during the kth event. Zwanzig defined the second part of the equation as the binary part of the force on the oscillator. He found that the binary part dominates relaxation when

$$\omega\tau_c \gg 1 \tag{6}$$

where ω is the oscillator frequency and τ_c is the time between events. Herzfeld convinced Zwanzig that this was consistent with IBC if τ_c is the time between effective events.[23]

According to Herzfeld, the molecules that were studied were high frequency oscillators and relaxed very slowly, therefore the time between effective events was very long. Zwanzig agreed that IBC was internally consistent.[24]

IBC remained stable until 1971, when Davis and Oppenheim used a master equation approach to describe vibrational relaxation in a liquid in the weak coupling limit.[25,26] Again their theory, as in earlier ones, applies only to high frequency oscillators. Even though they assumed weak coupling, they pointed out that using weak coupling theories may not be appropriate, because even though relaxation is slow, the forces that cause the relaxation are strong. They derived an equation that was forced into a binary form and found that

$$\frac{K_l}{K_g} = \left(\frac{\rho_l}{\rho_g}\right) \frac{g_l(R^*)}{g_g(R^*)} \qquad (7)$$

K_l is the rate for the liquid where the ij subscript has been dropped, K_g is the gas rate, ρ_l is the liquid density, ρ_g is the gas density, $g_l(R^*)$ is the radial distribution function for that liquid density evaluated at some R^*, and $g_g(R^*)$ is the gas radial distribution function evaluated at R^*. R^* is the turning point for the most effective collisions and it is assumed that this region is small. This is only valid for spherical molecules with small amplitude vibrations. Equation 7 could have been derived by incorporating into IBC, Einwohner and Alders' models for collision rates in a liquid.[27] A very

intuitive development of this is given by Delalande and Gale.[28] Notice that unlike the earlier equation for the rate by Litovitz, this incorporates the structure of the liquid. At this point experimentalists had started to look at vibrational relaxation with more specific techniques than ultrasound. Unlike the ultrasound studies, the use of lasers allowed experimentalists to study the relaxation of diatomics. The first experiments by Calaway and Ewing of the vibration to translation relaxation of N_2 in liquid N_2 served not only as a simple system to test the above ideas but showed the enormous range over which vibrational relaxation takes place.[29,30]

After 1984 IBC was applied to the relaxation of many simple molecules, however most of the experiments were vibration to vibration relaxations and not as simple to model as vibration to translation relaxations. In many experiments, IBC was used to explain the data and the basic theory was usually not questioned, only the effect of anisotropy, how hard or soft potentials affected the relaxation, how to calculate $g(R^*)$ and what R^* to use. One major change was a paper by Chesnoy and Weis[31] studying the density dependence of relaxation times. They performed a molecular dynamics simulation of a Lennard-Jones fluid and calculated two force autocorrelations functions as a function of density

$$F(t) = \langle \sum_b f(r_b(t)) \sum_c f(r_c(0)) \rangle \tag{8}$$

$$F_b(t) = \langle \sum_b f(r_b(t))\ f(r_b(0)) \rangle \qquad (9)$$

Where F(t) is the total force autocorrelation, $F_b(t)$ is the binary force autocorrelation and f(t) is the coupling from the Lennard-Jones liquid to the oscillator at time t. From these correlation functions and the Golden Rule they calculated the relaxation rate

$$\frac{1}{T_1} \approx \int dt\ e^{i\omega t} F(t) \qquad (10)$$

Basically the component of the force autocorrelation spectrum at the oscillator frequency determines relaxation. They found that the binary force autocorrelation function frequency spectrum was very similar to the total force autocorrelation function frequency spectrum, all the way to frequencies of ≈ 10 cm^{-1}. This would extend the validity of IBC calculations to near resonant vibration to vibration relaxation and perhaps to dephasing.

The most recent theoretical consideration of IBC was by Dardi and Cukier.[32,33,34] They calculated the relaxation of a dilute diatomic in a structureless fluid and discuss explicitly all approximations in their calculations and an IBC approach. They examine interference effects as other authors have and assumed again that for high frequency oscillations this is not a problem. Another assumption is that dynamically correlated collisions are not important. Dynamic correlations are the correlations between successive elastic collisions.

The nondynamic effects of correlated elastic collisions should be taken into account by the radial distribution function. No one has examined the effects of dynamic correlations on vibrational relaxation, although by it's very definition it shouldn't be important, since they are elastic collisions and may only contribute to a change in the equilibrium vibrational frequency. Another assumption implicit in all IBC models is that vibrational relaxation is a Markov process. Finally they propose that a weak coupling assumption can be made, unlike Davis and Oppenheim. Gas phase calculations have shown the weak coupling approximation to work well, if the inelastic cross section is much smaller than the elastic cross section, and the elastic and inelastic potentials are chosen correctly.[35,36] Cukier et al's final paper attacks the scaling of vibrational relaxation by the radial distribution function. They calculate the relaxation of an oscillator in a dilute gas using their formalism and show that the result is the standard dilute gas rate constant. They perform the same calculations for a liquid and find that to do the correct averaging the R^* of $g(R^*)$ must be chosen so large that $g(R^*) \approx 1$. They believe that there is no basis for using the ratio of $g_l(R^*)$ to $g_g(R^*)$ to explain the nonlinearity of relaxation vs. density. One possible criticism of their calculation is that they must assign a transition probability R, as a function of P(momentum) and b(impact parameter), R(P,b). Unfortunately they approximate R(P,b) as a constant in b up to b_{max}, where it

drops to zero. This functional form is highly unlikely for vibrational relaxation and most previous authors have assumed that R(p,b) is sharply peaked at b = 0. It is generally accepted that hard direct collisions are responsible for the majority of relaxation.

At this point in time Isolated Binary Collision theory is still in use even though the attack on the premise continues. Most of the more current studies are not on simple systems although Knudtson et al and Chesnoy have both studied HCl in liquid Xe, and Chesnoy has had good results interpreting the data in terms of the Isolated Binary Collision theory.[37,38] For this reason and the fact that the experimental and molecular dynamics data was available for iodine relaxing in xenon, the theory was put to a quantitative test, in order to find out if it would apply to such a low frequency oscillator (214 cm^{-1}) and if it could reproduce a system where the potential was known.

III. Application of Isolated Binary Collision Theory to the Vibrational Relaxation of Iodine in Liquid Xenon

A. Calculation of Relaxation Probabilities using SSH.

The first calculation of P_{ij} performed is one of the simplest approximations that could be made. The Schwartz, Slawsky, and Herzfeld (SSH) theory is the quantum mechanical successor to Landau theory.[39,40] The theoretical derivation has been performed by a number of people to different degrees of exactness.[41,42,43] The theory assumes an exponential potential, a harmonic oscillator and spherical symmetry. A good discussion of the calculation of relaxation probabilities can be found in Lambert's book or the review paper by Rapp and Kassal.[44,45] The calculation using SSH was performed more as an exercise to prove that it would not work and to check the results given by Kelly et al. Kelly et al proposed that SSH theory reproduced their experimental results quite well for the vibrational relaxation of iodine in weakly interacting solvents such as $CFCl_3$, $C_2Cl_3F_3$ and CCl_4.[46] This was quite surprising since a theory for a harmonic oscillator should not be accurate for iodine, which is fairly anharmonic especially high in the vibrational well. The calculation used the same formulas as Kelly et al, but the implementation must have been different due to the different conclusions reached. The calculation of P_{ij} was performed as follows.

$$P_{n+j,n} = [P_{1,0}]^j \frac{(n+1)(n+2)\ldots(n+j)}{(j!)^2} \quad (11)$$

The matrix must obey two constraints, population conservation and the Onsager equations of detailed balance that define the temperature of the system.

$$1 = \sum_j P_{ij} \text{ for all } i \quad (12)$$

$$P_{ij} = P_{ji} \exp\left(\frac{E_{ij}}{kT}\right) \quad (13)$$

where E_{ij} is the energy of state i minus the energy of state j.

The first test was to see if Kelly's results could be reproduced. Kelly stated that $1/P_{10}$ = 550 was a good fit for their experimental data for CCl_4 and was very close to the value determined by ultrasound data. In this group Harris et al had performed basically the same experiment with a shorter pulse laser with lower noise, and using the Frank Condon principle inverted the data to get the vibrational population as a function of time (see figure 1).[12] In order to compare to the experimental data the above equations were implemented in the following way. The iodine well was assumed to consist of 58 harmonic vibrational levels with 214 cm^{-1} frequency. Using $1/P_{10}$ = 550, the half above the diagonal of the matrix was calculated using equation 11. Next the half below the diagonal of the matrix was calculated using equation 13. Finally the diagonal was calculated using equation 12. This

gave the probability matrix, one could now apply this matrix to an initial vector and find out the time dependence of the vibrational relaxation. Only an initial population vector is now needed. At the time this work was originally completed it was thought that the iodine only geminately recombined after predissociation from the bound B state occurred and only then could vibrational relaxation begin. Predissociation was thought to take 15 ps., and the calculation was performed that way. This was implemented by having vibrational state 58 of the artificially harmonic iodine molecule populated with a time constant of 15 ps. Then a time step is taken every .1 ps (that is a collision rate of 10/ps was assumed, as in Kelly et al's paper) and the transition matrix is applied and more population is put in state 58 to simulate the time for predissociation. The complete relaxation is shown in figure 2 as a three dimensional plot of the population as function of energy and time.

Since that time work by Smith et al has shown that the predissociation takes place in less than 1 ps.[9] The same calculation is repeated as in figure 3 except that the recombination time is 1 ps instead of 15 ps. This changes the early time distribution of the relaxation but does not change the conclusion that SSH is not a correct theory for predicting the relaxation of iodine. There are two possible reasons for the discrepancy. The real iodine relaxation occurs much faster in the high energy part of the well. The SSH model does

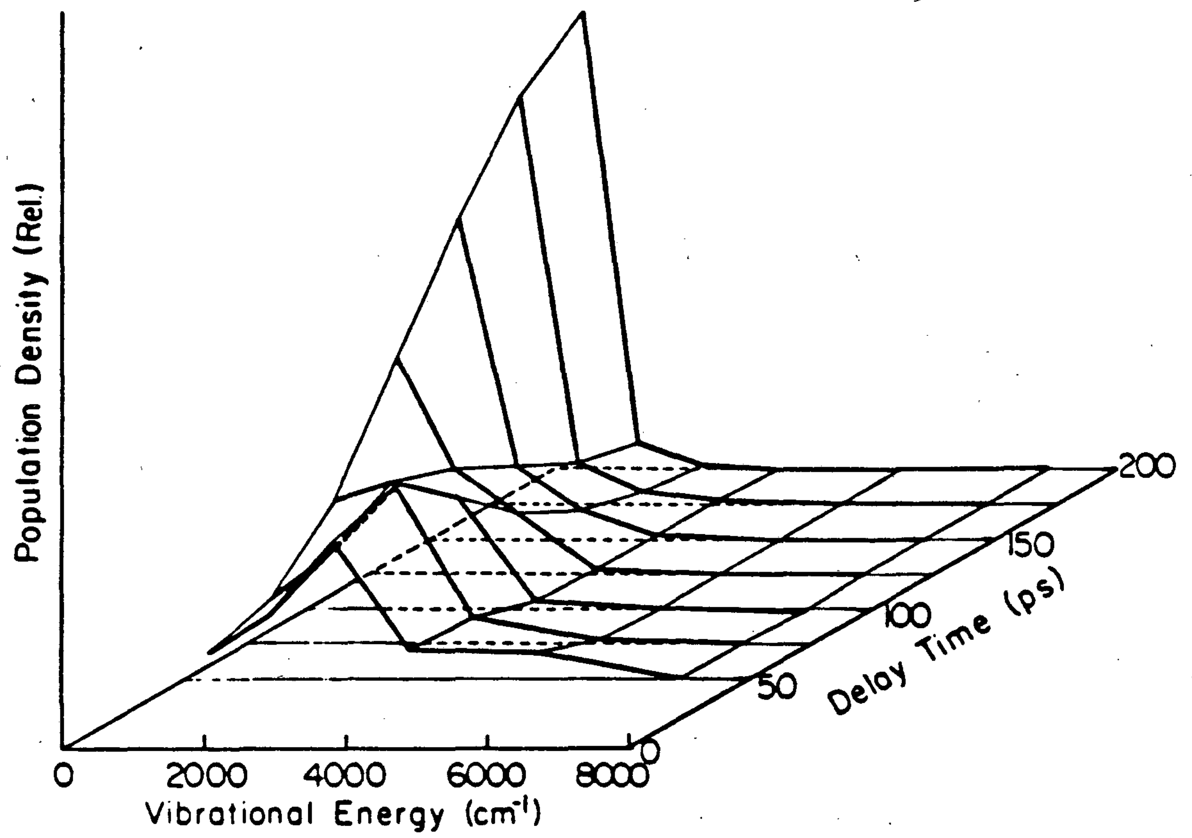


Figure 1 Experimental results of I_2 in CCl_4

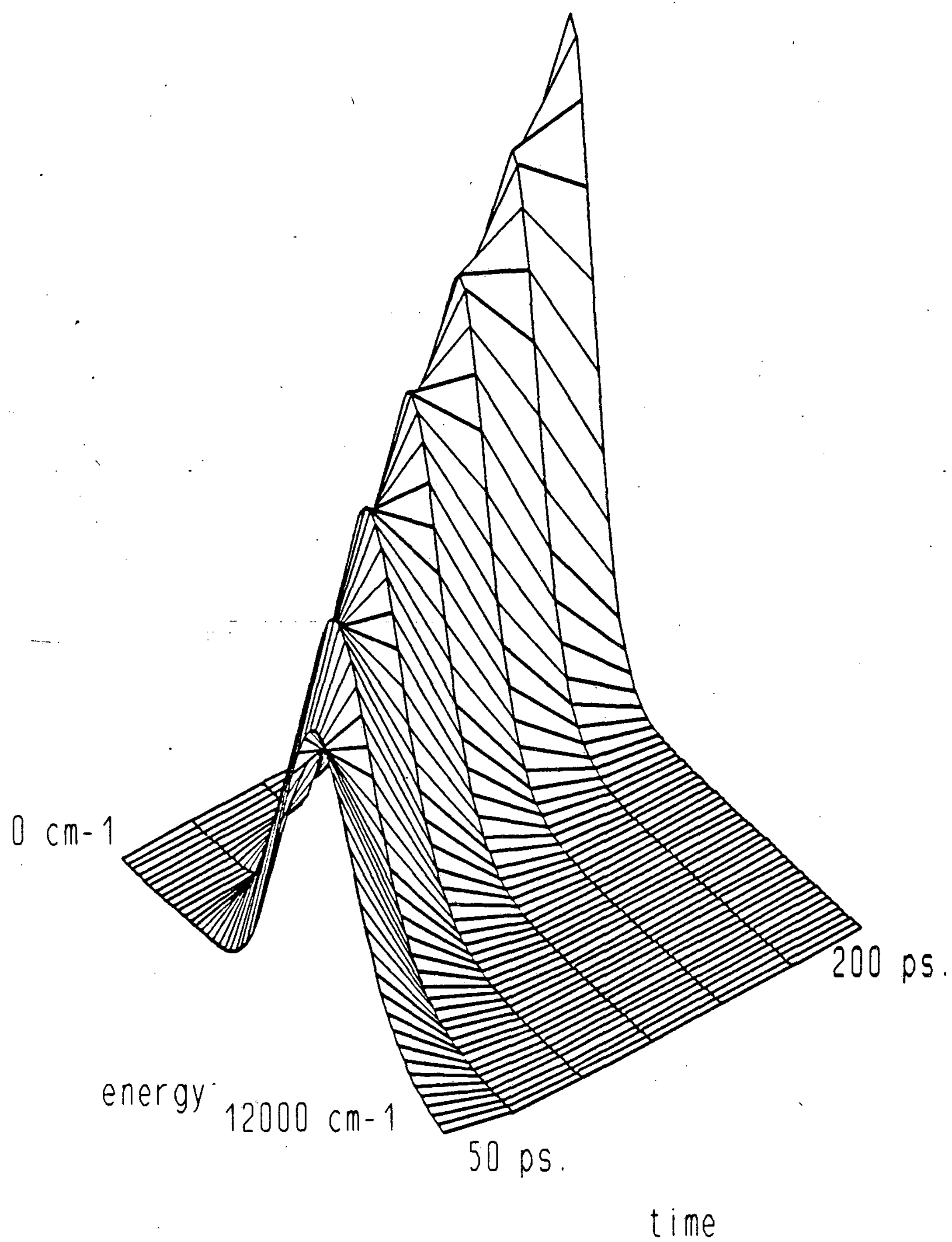


Figure 2 SSH Calculation with 15 ps. predissociation.

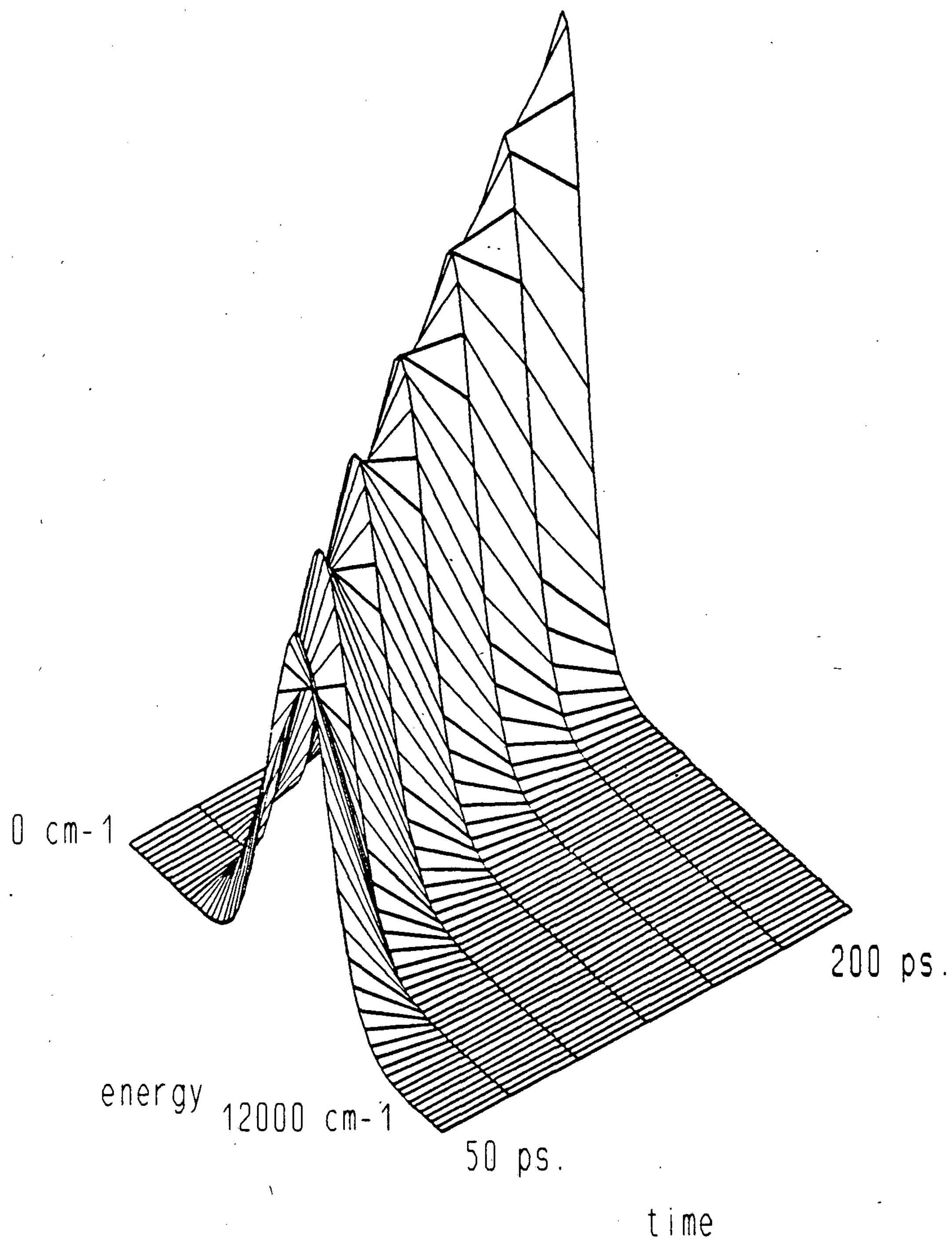


Figure 3 SSH Calculation with 1 ps. predissociation.

not reproduce this. This may be due to a relaxation channel that is open in the higher part of the well that is not open in the lower part of the well. For example perhaps the iodine molecule couples better to the vibrational and rotational modes of CCl_4 in the upper part of the well. This discrepancy may also be due to the fact that the SSH theory assumes a harmonic oscillator and iodine is certainly not that. The first possibility can be tested by seeing if SSH theory can predict the vibrational relaxation of iodine in liquid xenon modeled by Brown et al.[14] In this system there are no vibrational modes for the iodine to couple to. There are two parameters to fit, the collision frequency which is about 5 collisions/ps for iodine in liquid xenon at 1.8 g/cm^3,[14] and the value for P_{10}. In figure 4 a value of 1/735 was used for P_{10}. Also because of the way Brown et al created their plots of time dependence it is assumed that the recombination in this simulation is immediate or .1 ps. In Brown et al's plots they plotted the average vibrational energy loss where all trajectories started at 2000 cm^{-1}, below the dissociation limit. In other words there is no recombination dynamics in those plots. This was not to be a test of quantitative predictions but a test of qualitative usefulness. Figure 4 shows the same problems in fitting the vibrational relaxation as figure 2, and 3. If a value of P_{10} is chosen to simulate the relaxation in the lower part of the vibrational well, the upper part is too slow, and if a P_{10} is picked that gives fast

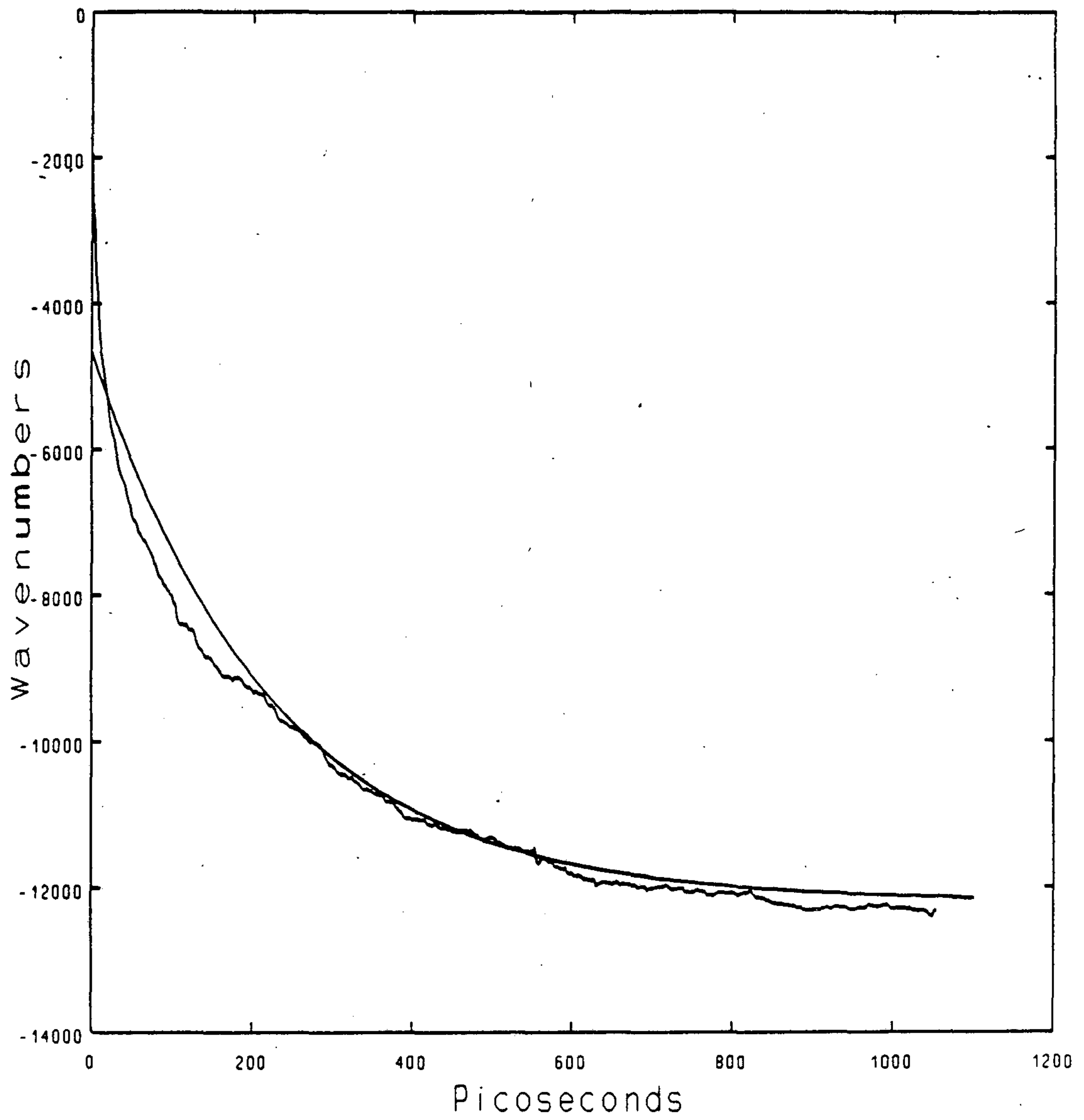
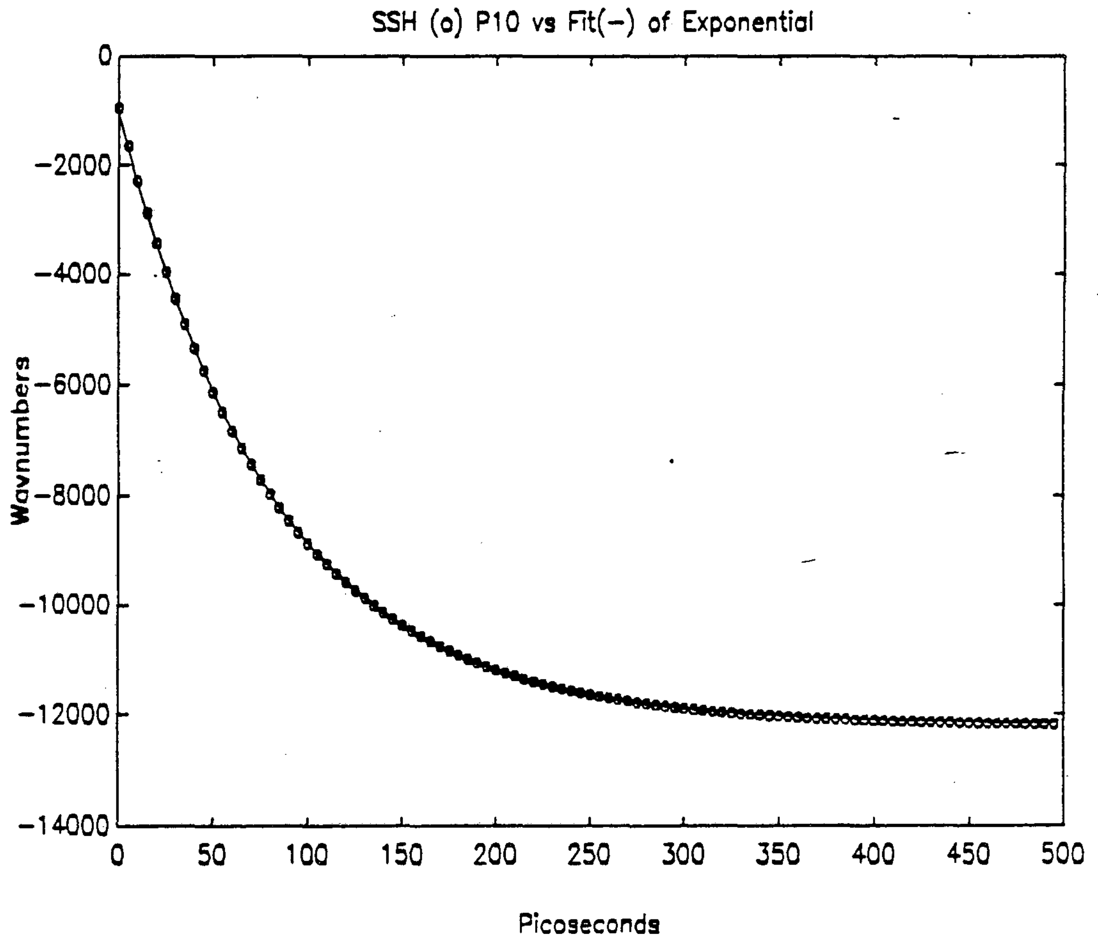


Figure 4 Molecular Dynamics (1.8 gm/cc^3) vs. SSH

relaxation at the top of the well the relaxation predicted by SSH theory is much too fast. This is because SSH theory implemented this way gives very exponential decays and the decay produced from molecular dynamics is not exponential. In figure 5 an exponential fit to the SSH data from figure 2 is performed. It does a very good job. Surprisingly the Molecular Dynamics data can also be fit with an exponential for the last 4000 cm^{-1}. This may be due to the relative harmonicity in the lower part of the well or the relative insensitivity of fitting the data.

In conclusion the application of SSH theory to the vibrational relaxation of iodine from the top of the well to the bottom is inappropriate. Iodine is a very anharmonic oscillator and the P_{ij} in the upper part of the well should be much larger than the P_{ij} in the lower part of the well. This can be seen by examining figure 7 which is the average energy loss from each state i for SSH theory. The average energy loss is linear as a function of vibration level. Note also that the large increase of energy loss at the vibrational energy $\approx$ 12000 cm^{-1} is an artifact of the calculation. The calculation did not allow for the upward transitions to dissociative states above the dissociation limit, and therefor the highest vibrational level can only relax or remain at the same energy.

B. Taking account of anharmonicity.


SSH (o) P10 vs Fit(–) of Exponential
Wavnumbers
Picoseconds
0
–2000
–4000
–6000
–8000
–10000
–12000
–14000
0
50
100
150
200
250
300
350
400
450
500

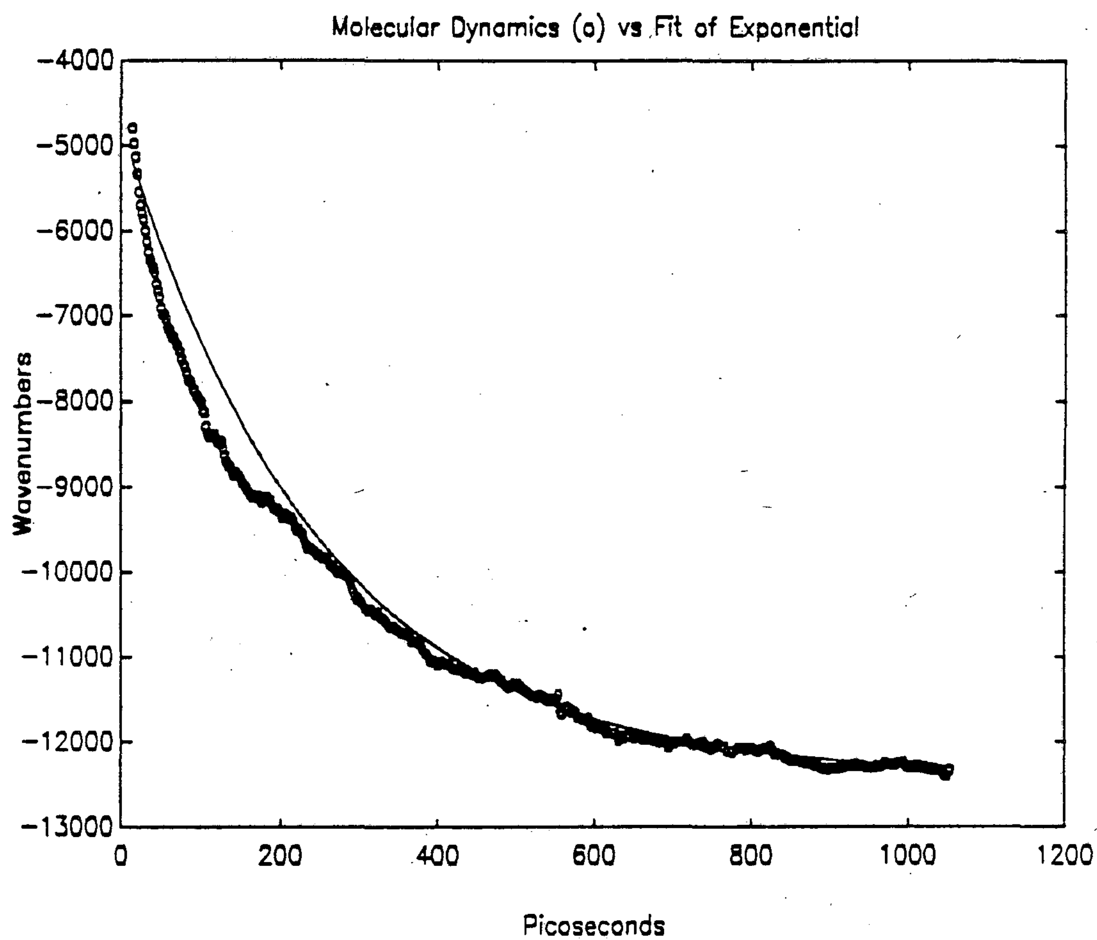


Figure 6 Molecular Dynamics fit to 247 ps. decay.

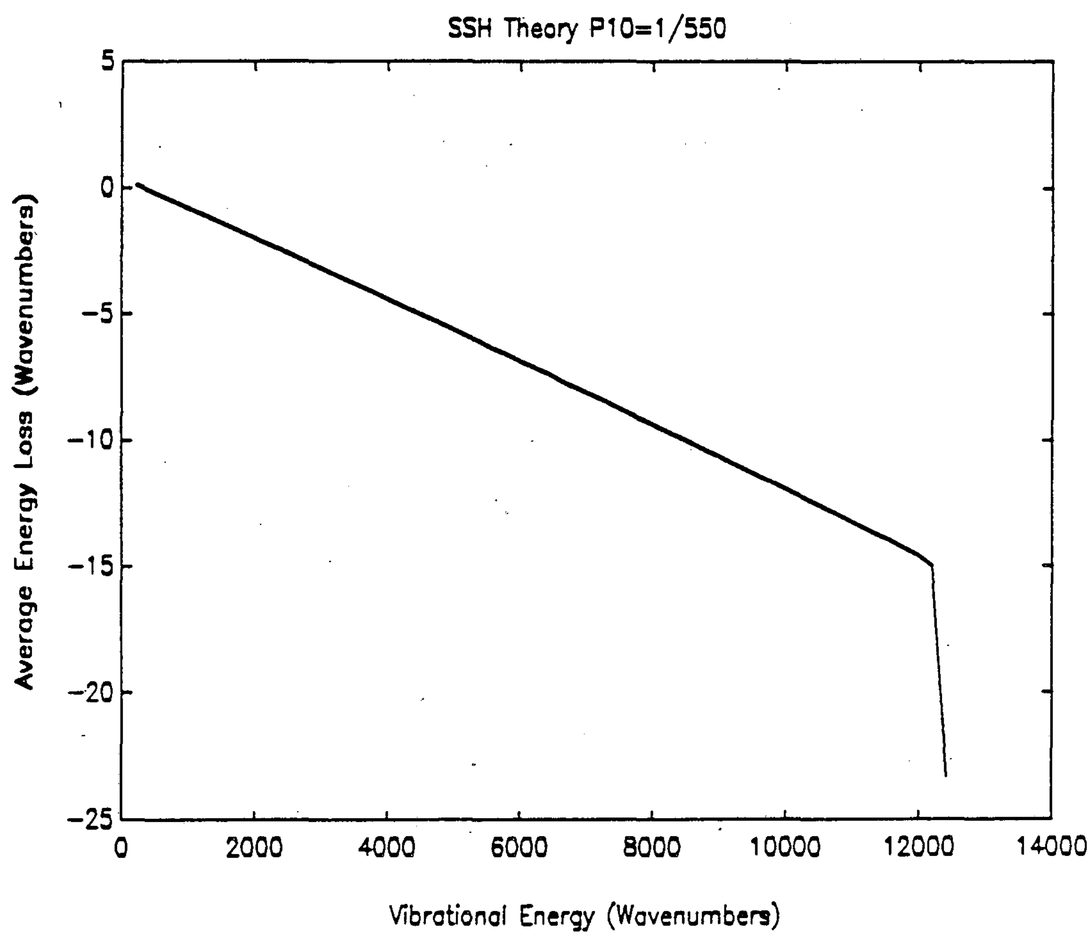


Figure 7 Average energy loss per vibrational level

The SSH calculation failed to model the vibrational relaxation of iodine even qualitatively, due to the neglect of anharmonicity. In this section a discussion of a simple model which includes anharmonicity is discussed. The calculation of P_{ij} performed for iodine and xenon is a one dimensional classical calculation of energy loss. The system can be treated classically because the de Broglie wavelength of both xenon and iodine is smaller than the length scale of interaction. Calculations were done in one dimension for two reasons. If three dimensional calculations were required the calculation would have become more time consuming and more complex. Secondly studies have shown that the one dimensional calculation is a reasonable substitute for three dimensions if the constraints described in paper by McKenzie are realized.[47]

The calculation of P_{ij} follows closely the calculations of Nesbitt and Hynes. There are changes from the potentials that they use in order to check how sensitive the calculations are to interaction potentials, and to compare more closely to the molecular dynamics simulations of Brown et al. The iodine-iodine potential used is the RKR surface of LeRoy. This potential is slightly steeper in the upper part of the well than the morse potential used by Nesbitt and Hynes. The potential between iodine and xenon is a Weeks Chandler Anderson (WCA) decomposition of a Lennard-Jones potential.[48]

	Potential	Form
One D Calculation	I-Xe	WCA $\epsilon = 225\ cm^{-1}$ $\sigma = 3.94\ \AA$
	I-I	RKR
Molecular Dynamics Calculation[14]	Xe-I	Lennard-Jones $\epsilon = 225\ cm^{-1}$ $\sigma = 3.94\ \AA$ $c = .84\ cm^{-1}$ [49]
	I-I	RKR

Table 1

Note that the WCA decomposition was originally intended to explain liquid structures for reduced densities greater that 0.6 . Even though most of the comparisons to molecular dynamics will be in this range, WCA was not chosen for this reason. The WCA decomposition was chosen for three reasons. When a gas liquifies, energy is released, the heat of vaporization, due to the solvent atoms spending most of their time in the bottom of the well where the potential is repulsive. The attractive part of the potential is defined as the part of the potential where the accelerations are negative. Note that in the Lennard-Jones potential, the potential energy may be negative for $r < \sigma 2^{1/6}$, but the accelerations are not negative. Since the liquid samples the attractive part of the potential, but this part of the potential does not contribute strongly to relaxation, it was thought that the IBC simulation would be more realistic if it also did not sample that part of the potential. The turning point in the gas phase will also be on average at a smaller radius than in the liquid due to the heat of vaporization, however the one D model's turning point should be comparable to the molecular dynamics simulation due to the use of the WCA potential. Secondly not having an attractive section of the potential makes the integration of the trajectory much quicker since there is less distance to integrate over and there is no possibility of forming a long lived complex. Thirdly the

molecular dynamics trajectory simulation that the trajectories

$$V(r) = 4\varepsilon[(\frac{\sigma}{r})^{12} - [(\frac{\sigma}{r})^{6}] + c \quad where\ r<r_c$$

$$c = 4\epsilon[(\frac{\sigma}{r_c})^{12} - (\frac{\sigma}{r_c})^{6}] \quad where \quad r_c = 2^{(\frac{1}{6})}\sigma \qquad (14)$$

$$V(r) = 0. \quad where\ r>r_c$$

would be compared to used a Lennard-Jones potential and the WCA decomposition is the closest approximation to the Lennard-Jones potential within the above constraints. This potential is steeper than the exponential potential used by Nesbitt and Hynes.

A one dimensional trajectory is calculated where the xenon collides with the iodine collinearly, this is averaged 100 times over iodine's vibrational phase at 15 different velocities chosen from a Maxwell distribution at 300 K. This is done for 82 vibrational levels of the RKR potential. The higher levels where dissociation is significant, were not studied because the purpose of the calculation was to try and understand the slow vibrational relaxation and not the fast recombination dynamics.

The actual code used to calculate the trajectories are very similar to Brown's code (see program listing I2IBC). Beeman's method was used for the integrator.[50] This was the same integrator used in the molecular dynamics simulations of Brown. This is a second order integrator. For the one dimensional trajectories a time step of .1 fs was used. This is a smaller time step than used in the molecular dynamics

simulations. This was due to the fact that at low vibrational energies, iodine lost very little energy and the energy conservation should be at least an order of magnitude better than the energy lost. The code was run on a Cray X-MP at the University of San Diego Supercomputer Center. Since there were only three bodies interacting in this model the code would not take advantage of the speed gains from vectorization without modification. This was accomplished by running 15 trajectories simultaneously, which could be written as vector code with vectors long enough to vectorize efficiently.

The trajectories show qualitatively what you would expect keeping in mind iodine's anharmonicity. Vibrational energy transfer increases non linearly as a function of v, the vibrational energy quantum number (see figure 8). This is at variance to Landau Teller theory, which predicts a linear increase in v, and the SSH results which show a linear increase as a function of v. Of course Landau Teller theory is based on a harmonic oscillator and iodine is most definitely not a harmonic oscillator. The shape of the curve is qualitatively the same as that which Nesbitt and Hynes found and even follows their power law dependence (see figure 9). Nesbitt and Hynes found that

$$\ln\left[\frac{\Delta E}{\omega(E)}\right] \ vs \ \ln[v] \qquad (15)$$

is linear with a slope of 4.3 . The results for the WCA potential are also linear for the above function, although the

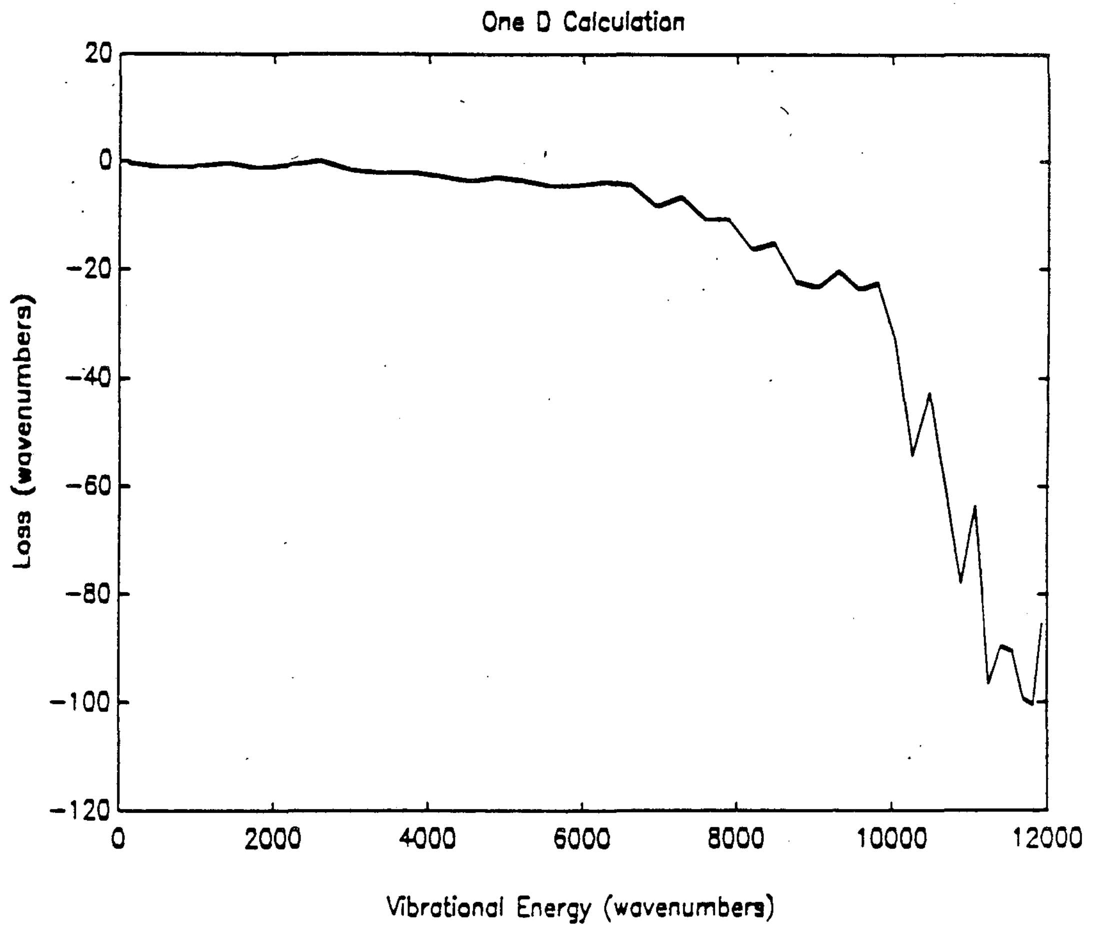


Figure 8 One D average energy losses vs vibrational energy.

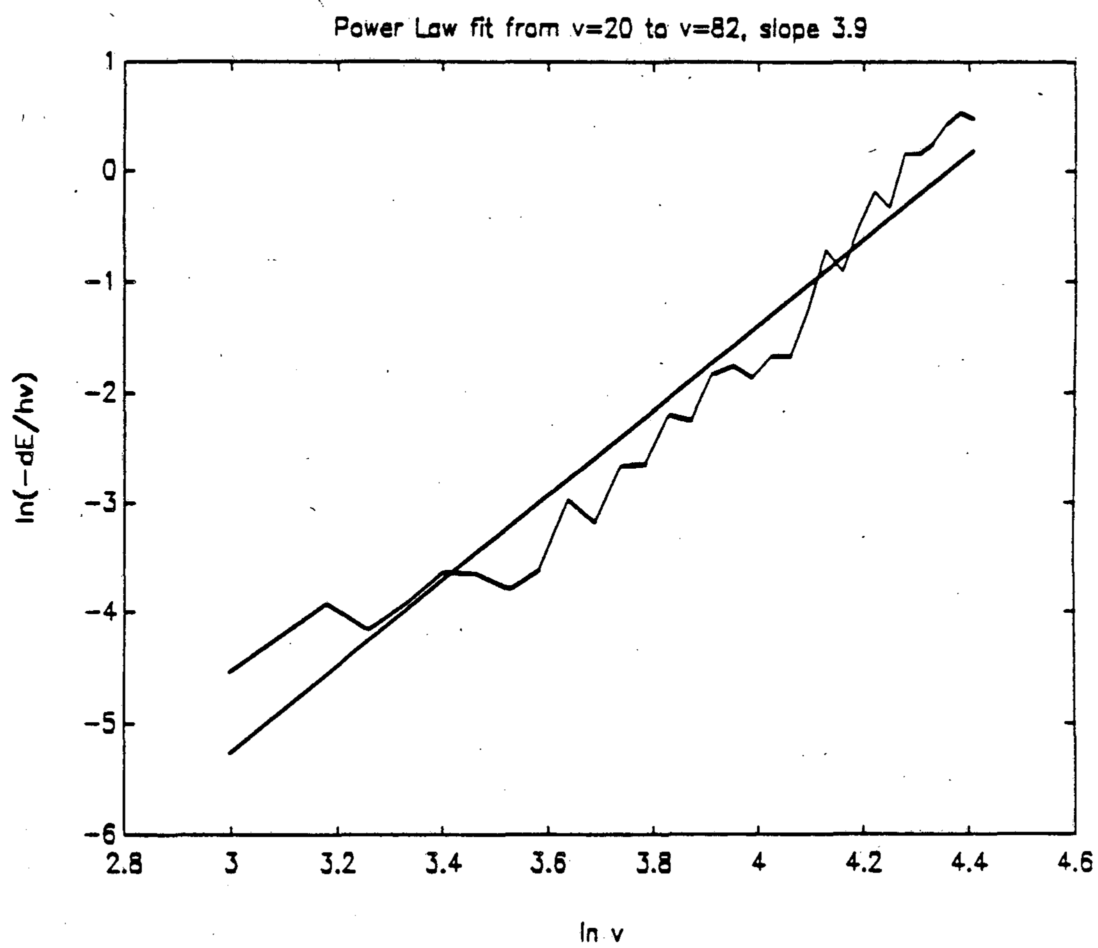


Figure 9 Power Law of Average Energy Loss

slope was only 3.9. This is probably due to the steeper interaction potential between iodine and xenon in this study. Nesbitt and Hynes studied the dependence of equation 15 as a function of the steepness of the interaction potential. They found that the slope of the above equation decreased as a function of increasing interaction. Most interesting is that the current study follows the same power law even though the interaction potential is a completely different functional form. This seems to indicate that the most important physical aspects of vibrational relaxation for this system are explained by Nesbitt and Hynes's hard sphere model description. They assumed that since the iodine molecule in the upper part of the well spends most of its' time at the outer turning point, where the potential is flat, the dynamics "ought to resemble unbound particles", and therefore can be approximated in the hard sphere impulsive limit. The second important consideration in their model is which parts of the oscillator phase may be accessed by the colliding xenon. Using only these two assumptions they were able to reproduce their results for an exponential interaction with a very steep potential. The phase of the oscillator that is sampled in their model is determined by the oscillator potential, which is approximately the same for both studies, and the relative speed of the iodine and xenon, which is determined by the temperature, which is the same for both studies. The dynamics of collision are not exactly the same, due to the different

interaction potentials, however the slope of the potential seems to determine the slope of the power law dependence and not the functional form.

Given the similarity between the results of the two models, Nesbitt and Hynes and this calculation, it seems to be a good assumption that the P_{ij} are qualitative good approximations for the energy loss given their relative insensitivity to potential changes. The problem of using the right potential to map the potential that iodine feels from the xenon in the liquid to some appropriate gas phase potential is not resolved. Within the framework of IBC theory though, it must be a small effect or IBC theory will be essentially invalid. If there is a way to map the average forces the iodine feels in liquid xenon to the gas, the result will certainly be density dependent. For example if the density of xenon is increased, the xenon spend more time on average at a smaller distance from the iodine molecule, and therefore will sample on average a different part of the potential than a less dense liquid xenon atom would. If the average interaction is density independent, the $\bar{P}_{ij}$ will also be density independent and that is the main foundation of IBC theory.

Assuming that the WCA potential is a good approximation to the average potential of interaction for the iodine-xenon system, the next question is what the vibrational energy as a function of time is. This calculation was performed two ways

to make sure that the approximations used were valid. First the iodine molecule was placed in vibrational state 82 and randomly one of the 1500 energy losses for the 82nd vibrational level was chosen (see program IBCENE). If it was an increase in energy the iodine molecule remained at level 82. If it was a decrease in energy the iodine molecule lost that amount of energy. If the energy of the iodine molecule was in between vibrational levels at this step, a linear combination of two randomly chosen energy losses from the level above and below the iodine's current energy were chosen. The two randomly chosen energy losses were weighted according to their distance from the iodine's current energy. This would provide a new energy loss. This was repeated until the iodine relaxed. The whole process was repeated 100 times for good statistics.

Another calculation of the energy vs time was performed by modeling the distribution of energy losses for each P_{ij} as a gaussian. The distributions of energy losses for each P_{ij} was very close to a gaussian and this allowed for a very efficient calculation. Basically the same program as above was used, except that the random energy losses chosen were picked from a gaussian distribution that modeled the energy loss distribution. If the energy of the iodine molecule was in between vibrational levels a linear combination of the gaussian parameters was used weighted by proximity to the iodine energy. This gave results that were indistinguishable

from the first calculation.

The more important question for IBC is the calculation of Z. If IBC theory is an appropriate theory, the value of P_{ij} could be found experimentally, although it is not really needed to test the theory because if P_{ij} was not known, IBC could still be tested by it's prediction of the density dependence of the relaxation. Because the relaxation of iodine by xenon is affected quantitatively by the potentials used, the calculation of Z will be made with respect to molecular dynamics simulations and only qualitatively to the experiments of Paige et al.

In the molecular dynamics simulation of Brown et al the relaxation of the iodine at the densities of 1.8 g/cc and 3.0 g/cc could be overlapped on top of each other by scaling the time axis of the decays. Brown et al found that the 3.0 g/cc relaxation was approximately 4 times faster than the 1.8 g/cc. This seemed to indicate that the relaxation may be described by IBC. This was not expected to occur although work by Chesnoy seems to indicate that it might be possible.[31] From the calculation of P_{ij} and the calculation of energy vs time described above, the vibrational relaxation of iodine vs time can be calculated assuming a collision frequency. For a collision frequency of 4.5 per ps. the one dimensional calculation does a very good job reproducing the energy relaxation (see figure 10).

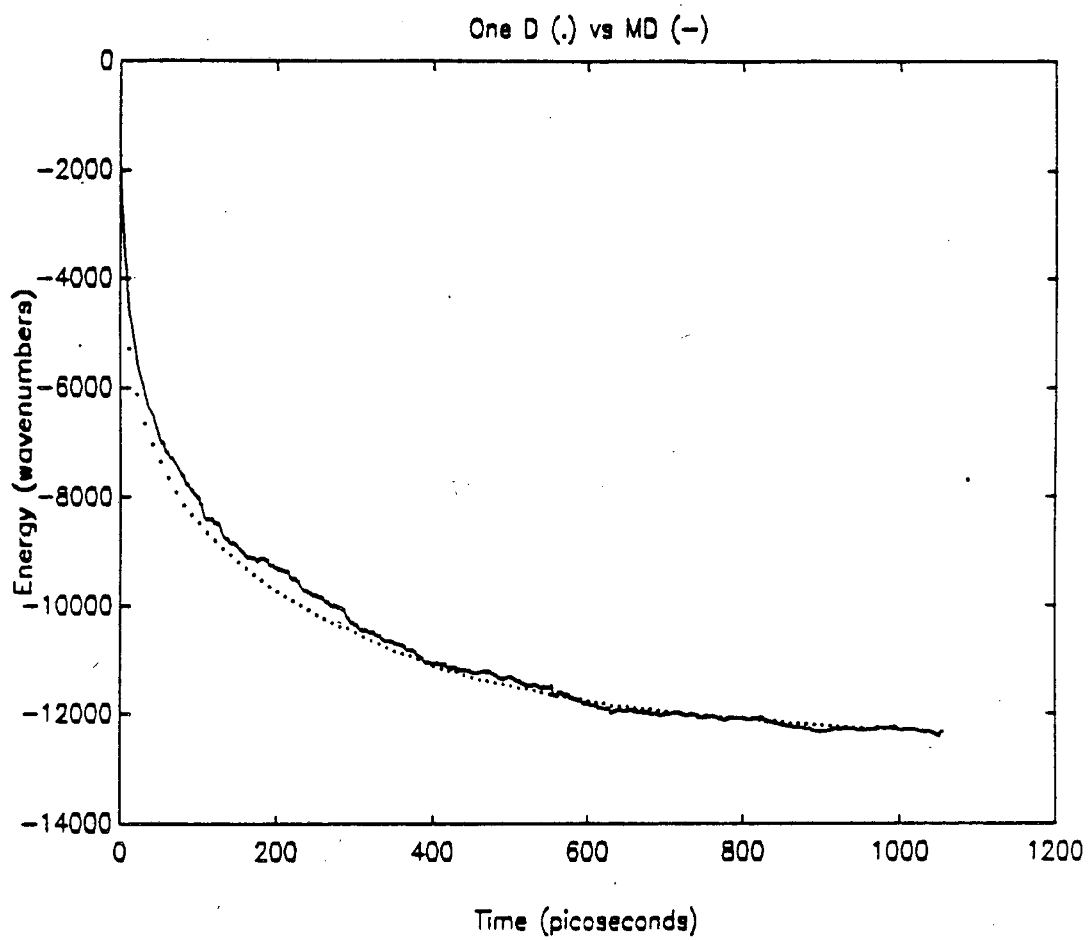


Figure 10 One D vs MD Xe 1.8 g/cc

C. Calculations of the Density Dependence.

The one dimensional calculation performed very well with respect to qualitative aspects of modelling the vibrational relaxation, especially at incorporating the anharmonicity of iodine. It also seems that the choice of the WCA decomposition of the potential may be appropriate. Since the trajectory calculations were one dimensional there must be some weighting factor to take into account that some collisions are not collinear. It is not clear what that factor should be, for example Nesbitt and Hynes selected 1/2 for their calculation.[8] The value for the steric factor could range from all the way from one to less than 1/3.[51,52] In fact Shin has calculated steric factors for iodine-argon and found 1/8 to be appropriate. However, if there are any three dimensional effects that were missing they are effects that are constant throughout the 82 vibrational levels. If the steric factor was dependent on the vibrational levels, the one dimensional calculation probably would not have fit the three dimensional MD simulation.

In order for IBC theory to be a useful theory it must also be able to make quantitative predictions of the relaxation for a particular density and predictions of the density dependence. The collision rate of 4.5 per ps. is a quite reasonable first order guess of what the collision rate should be for xenon at 1.8 g/cc[3]. An estimate of the

collision frequency can be found using equation 2. This is the simples treatment for finding the collision frequency, given the non-linear increase in relaxation with increasing solvent. The average velocity is

$$V = \left(\frac{8kt}{\pi\mu}\right)^{\frac{1}{2}} \qquad (16)$$

where μ is the reduced mass of the I_2- Xe system. The value to use for σ is unclear due to it's ambiguous definition. From the one dimensional calculation, the turning point for the most effective collisions was 3.7 - 3.8 Å. This number comes can come from qualitatively considering that P_{ij} is an increasing function of velocity and the magnitude of the velocity distribution is rapidly decreasing for high velocities. Therefore, the turning point of the most effective collisions will be a trade off between high velocities for large P_{ij} and the fact that for a given temperature the number of particles with a high velocity decreases with increasing velocity. This is the R^* that is defined in equation 7 earlier. If this value is used for σ in equation 2, the collision rate is calculated to be ≈ 2 ps^{-1}. Unfortunately, to have any faith in this equation it would have to predict a larger collision rate than 4.5 ps^{-1} to be acceptable. If equation 2 had predicted a collision frequency higher than 4.5 ps^{-1} , the steric factor could be used to explain the discrepancy.

The problem of predicting the correct collision frequency

to reproduce the MD or the experimental result is probably intractable. The mapping of the potential that is sampled in the liquid phase to the gas phase is probably not quantitatively correct. It is close enough to reproduce the relaxation qualitatively, but any shifts up or down on the potential will probably change the rate over all. Comparisons to the experimental results also suffer the same problems along with uncertainty in potential parameters. The uncertainty in potential parameters may cause only quantitative problems and not qualitative problems if the basic shape of the potential is correct and only a linear change in the slope is needed. For example from the Golden Rule, the relaxation rate is

$$\tau_{ij}^{-1} = \frac{2\pi}{\hbar}\sum_{\alpha}\sum_{\beta} P_{\alpha}|V_{i\alpha,j\beta}|^2\ \delta(E_i - E_j + E_{\alpha} - E_{\beta}) \tag{17}$$

where P_{α} is the probability of the bath being in state α, and V is the part of the hamiltonian which couples the vibrator to the bath. Suppose for example coupling is due to the Lennard-Jones potential, and ϵ is the parameter that has the most error in going from the one dimensional simulation to the real potential. Then ϵ^2 can be pulled out of equation 17 and the qualitative shape of the relaxation vs time will not change and only the time scale will change. The change will be proportional to ϵ^2. For these reasons the rest of the comparisons will be made to density dependencies and not actual collision frequencies.

In figure 11 the density dependence of equation 2 is plotted. All the rates are scaled relative to the relaxation in Xe at density 1.8 g/cc. The σ chosen for the plots are 3.7 Å and 3.8 Å. The results are quite encouraging with regards to the good agreement with the experimental results. The comparison to the molecular dynamics is not good and this is quite disconcerting. Since the molecular dynamics relaxes 10 times as fast as the actual experiment, it should be closer to a model which treats the system as hard spheres. There is one more reason why equation 2 should fail to model the molecular dynamics correctly. Equation 2 does not take into account the short range structure of the liquid. Equation 2 treats the liquid as a system where the particles are evenly distributed and for dense fluids this is not true.

Equation 7 does include the liquid structure in it's calculation of relaxation rates, although the use of this equation has many severe restraints. The molecule must be nearly spherical because there is no way to incorporate simply any angle dependent relaxation rate. Secondly the trajectories that lead to most of the relaxation must have a narrow distribution in velocity space or equation 7 will not be valid. The final problem of equation 7 is calculating the radial distribution function. It is only fairly recently that the computer power needed for this calculation was distributed widely enough for it to be used on problems such as this. Before that, approximations to the radial distribution

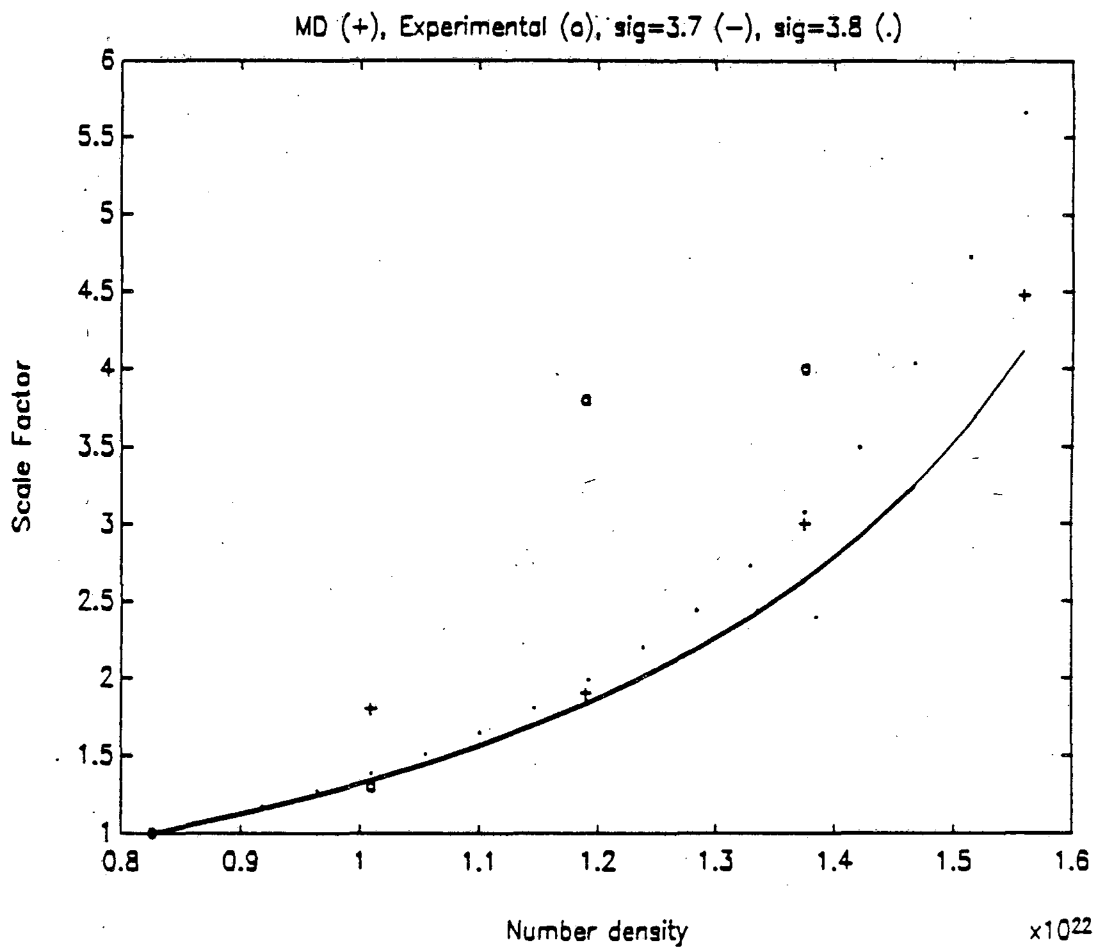


Figure 11 Scaling Predicted by Equation 2

function were used, sometimes correctly and sometimes without a clue.

One approach to this problem was to use the attractive hard spheres pair distribution model by Delalande and Gale.[53] This model assumes that the collision rate should be calculated at the hard sphere radius. One then assumes the radial distribution function at R^* can be approximated by the Carnahan and Starling approximation,[54]

$$g(R^*) \approx g_{HS}(sig) = \frac{(1-\frac{\eta}{2})}{(1-\eta)^3}; \quad \eta=\frac{\pi}{6}\rho\sigma^3 \tag{18}$$

where σ is the hard sphere contact distance and ρ is the number density. The problem with this approximation is that the hard sphere radius which provides the best model for the radial distribution is not necessarily the correct radius at which to evaluate R^* (see figure 12). A more sophisticated version of this theory was employed by Madden and van Swol.[55] They used WCA theory to calculate the cavity distribution function, which was then related to the ratio of vibrational relaxation rates in a dilute gas and a dense liquid. This assumed that g(R) could be approximated by a properly chosen hard sphere fluid of the same density. They did not equate R^* with the hard sphere diameter used to calculate the radial distribution function.

We calculated the radial distribution of an iodine atom in liquid Xe directly. Again this makes the approximation that the iodine molecule is spherical and the additional

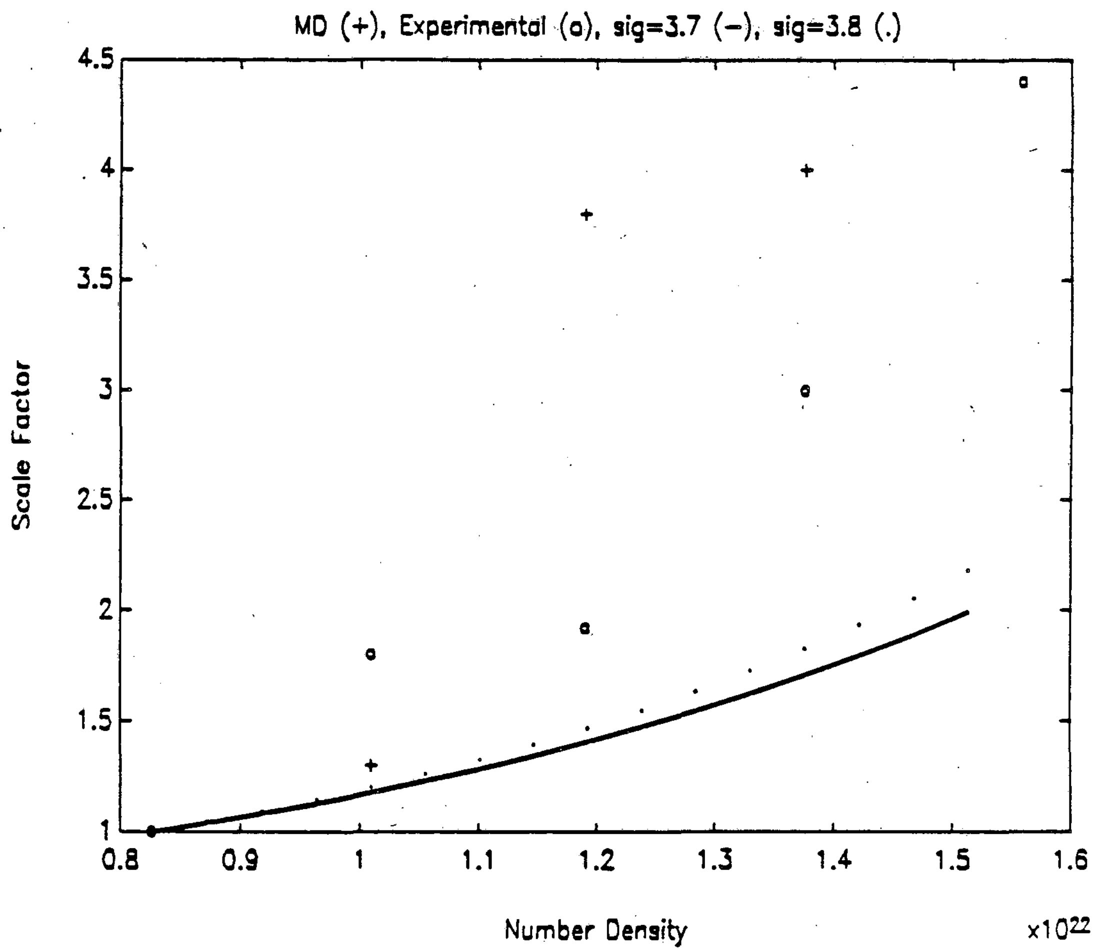


Figure 12 Scaling Predicted by Equation 18

approximation that the radial distribution around the iodine molecule can be approximated by the radial distribution around an iodine atom. Both of the assumptions are not true, although the differences may not be consequential. Due to the assumption that all forces in the liquid were pair wise additive and Lennard-Jones potentials, there is a large potential well between the two iodine atoms that causes the radial distribution to be higher there than on the ends of the iodine molecule. This can be seen in the MD calculations of Brown et al. This may not be a problem because it is not expected that the trajectories of the xenon atoms between the two iodine atoms will contribute significantly to vibrational relaxation.

The radial distribution function is calculated in the program IIBC. The simulation puts one iodine atom in a liquid of 107 xenon atoms with periodic boundary conditions. The question to be asked now is what potential to use in the density comparison? The MD was run using a Lennard-Jones potential but the one D trajectories used a WCA decomposition. For high density, where the WCA potential provides a good substitute for the Lennard-Jones potential, there is no discernable difference between the radial distribution functions produced (see figure 13). However, at 1.8 g/cc the differences are quite obvious (see figure 14). Since the quantity needed is the flux of xenon atoms that is colliding with the iodine molecule in the MD, it seems that the best

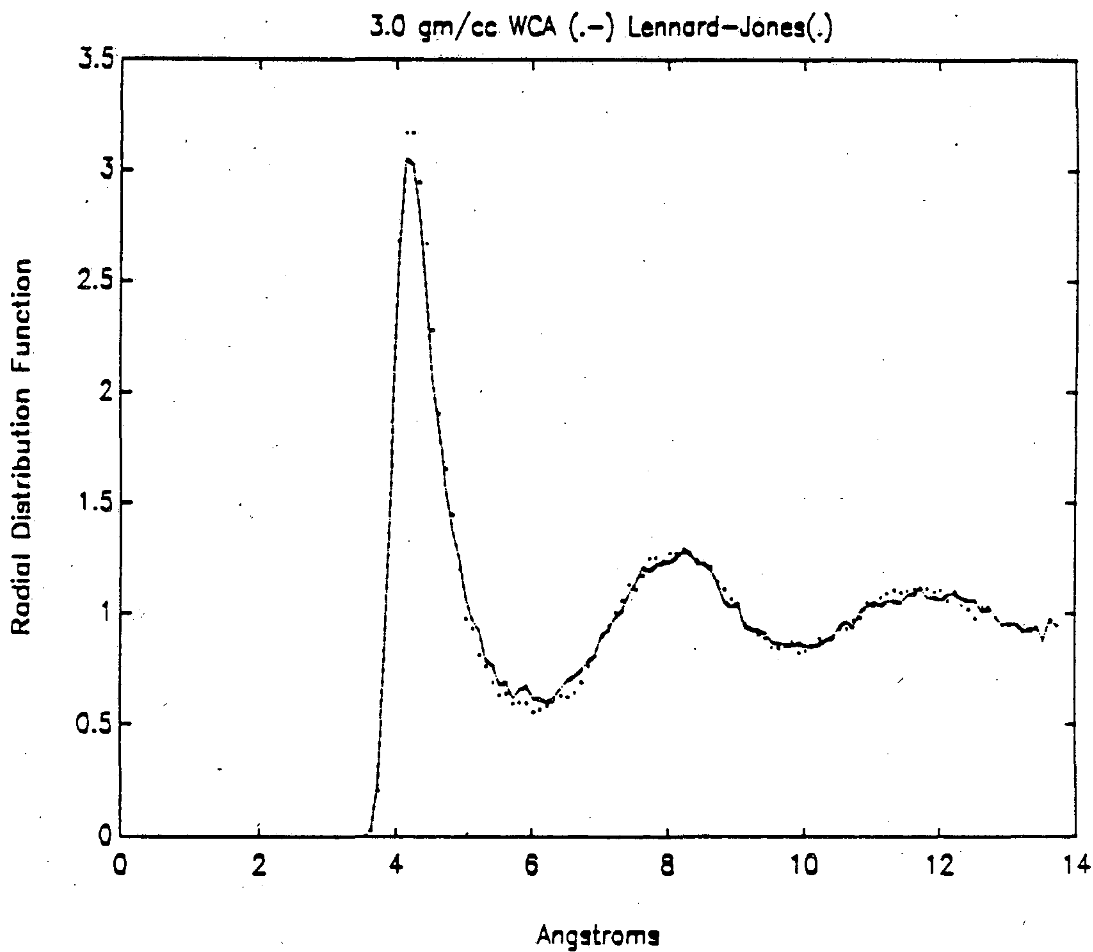


Figure 13 One Iodine Atom in Liquid Xenon

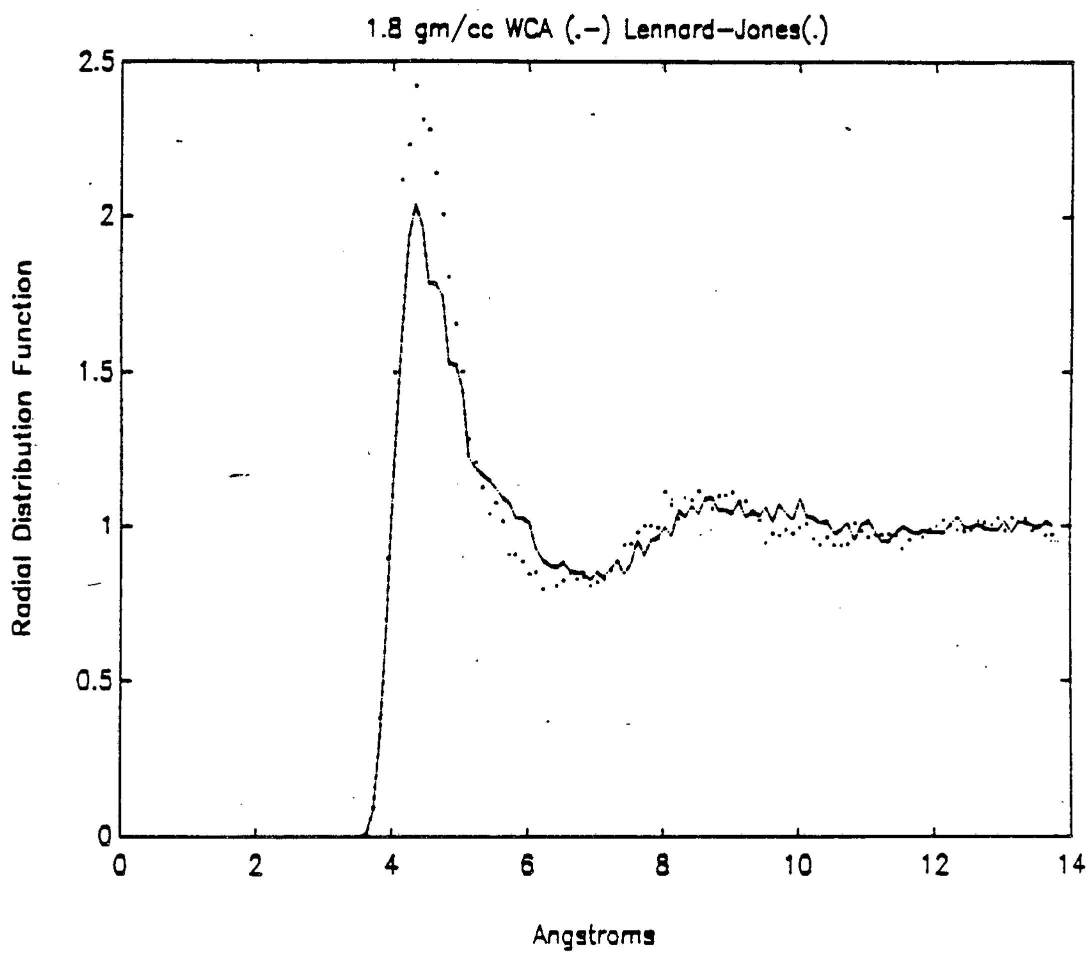


Figure 14 One Iodine Atom in Liquid Xenon

potential to use would be the Lennard-Jones potential used in the MD. Figure 15 shows the radial distribution functions using Lennard-Jones potentials for all 4 densities studied by the molecular dynamics. The highest density (3.4 g/cc) studied experimentally can not be studied this way due to the fact that it is a Lennard-Jones solid at that density even though in reality xenon is still a fluid at that density. Note that the area of interest in the radial distribution functions is the radii between 3.7 Å and 3.8 Å where the radial distribution is changing quite rapidly. That region is of most interest due to the calculation that showed that 3.7 Å to 3.8 Å is the region where the most efficient trajectories have their turning point. Figures 16-18 show that using the R^* calculated from the one dimensional trajectories, 3.7-3.8 Å, good agreement was found for the scale factors given by equation 19 for the different densities.

$$Scale\ Factor = \frac{K_1}{K_2} = \left(\frac{\rho_1}{\rho_2}\right) \frac{g_1(R^*)}{g_2(R^*)} \tag{19}$$

In figures 16-18 equation 19 is plotted for the densities 3.0 g/cc, 2.6 g/cc and 2.2 g/cc scaled by the density 1.8 g/cc. The straight solid line is the prediction of the MD calculations with the dashed line indicating a standard deviation of the MD calculated scale factors. The only major disagreement comes at the density of 2.6 g/cc. However this density had the fewest trajectories averaged and there may be a systematic error for this trajectory. In figures 19-21 the

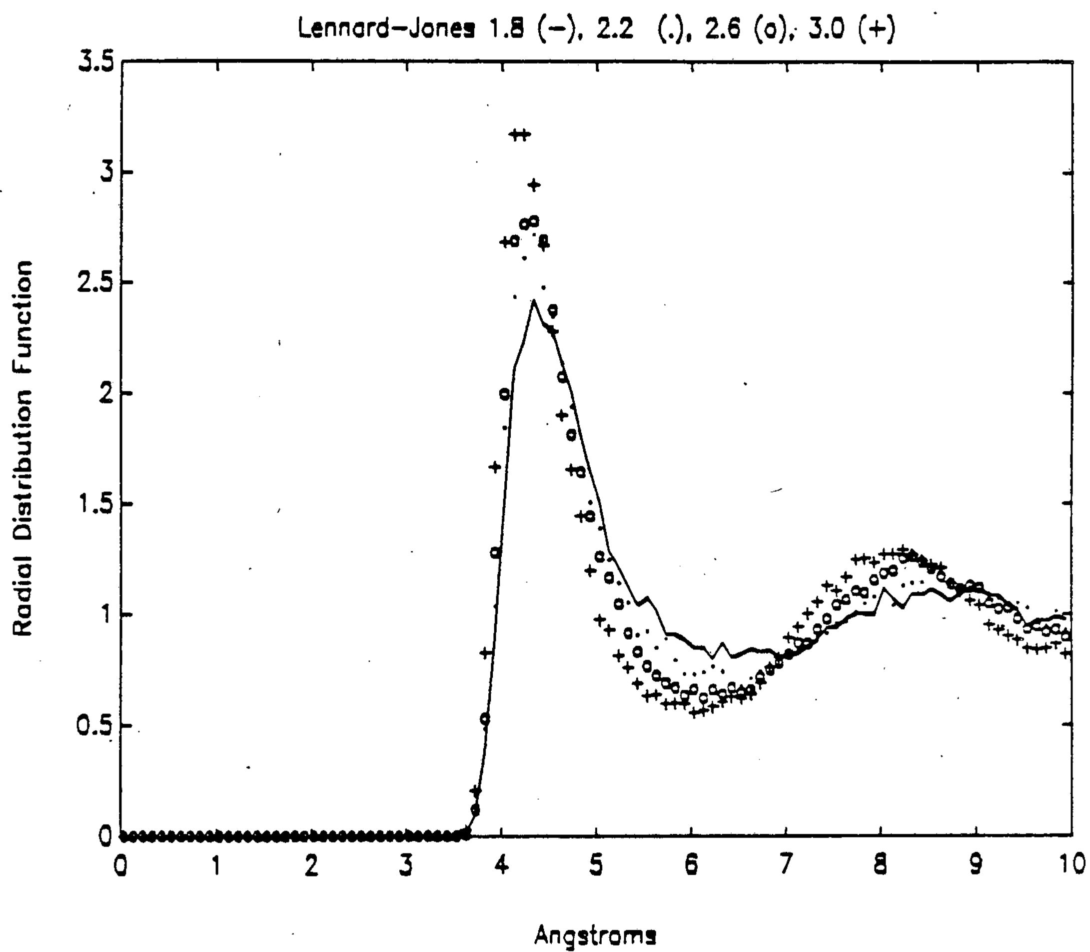


Figure 15 One Iodine Atom in Liquid Xenon

same calculation is plotted, except the comparison is made to experimental results. There is even better agreement with the experimental results. All the scale factors were within the error of the measurement. This was unexpected due to the difference in the potentials that are used and the real potentials which are unknown.

The scale factors could also have been calculated using continuum theories. As pointed out earlier, the vibrational relaxation rate is affected by

$$F(t) = \langle \sum_b f(r_b(t)) \sum_c f(r_c(0)) \rangle \qquad (20)$$

$$F_b(t) = \langle \sum_b f(r_b(t))\ f(r_b(0)) \rangle \qquad (21)$$

Where F(t) is the total force autocorrelation, $F_b(t)$ is the binary force autocorrelation and f(t) is the coupling from the liquid to the oscillator at time t. Oxtoby has also considered this type of division of the forces.[56] From these correlation functions and the Golden Rule, the relaxation rate is

$$\frac{1}{T_1} \approx \int dt e^{i\omega t} F(t) \qquad (22)$$

Basically, the magnitude of the force autocorrelation spectrum at the oscillator frequency determines the relaxation rate. Figure 22 shows the total force autocorrelation and the binary force autocorrelation functions for an Iodine atom in liquid Xe at 1.8g/cc. This was calculated in the program IIBC. The

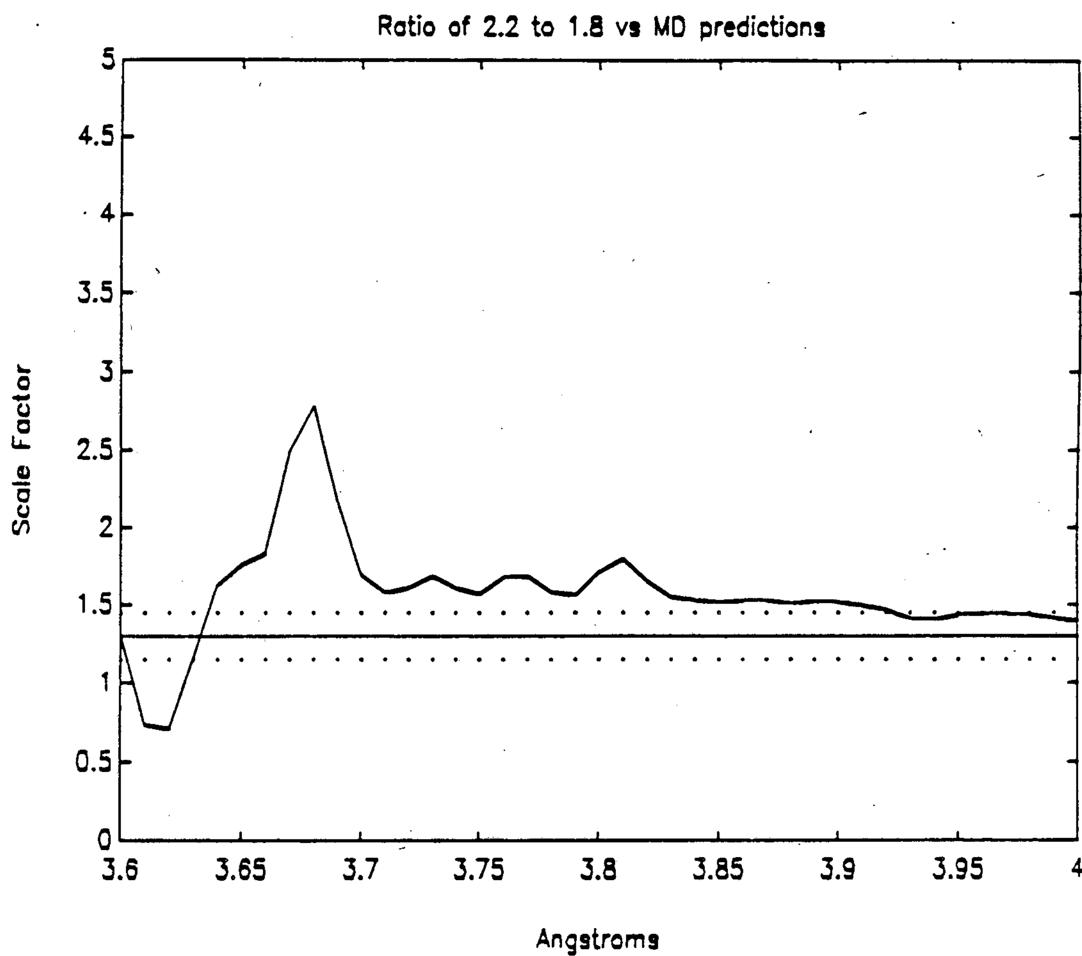


Figure 16 Scaling Predicted by Equation 19

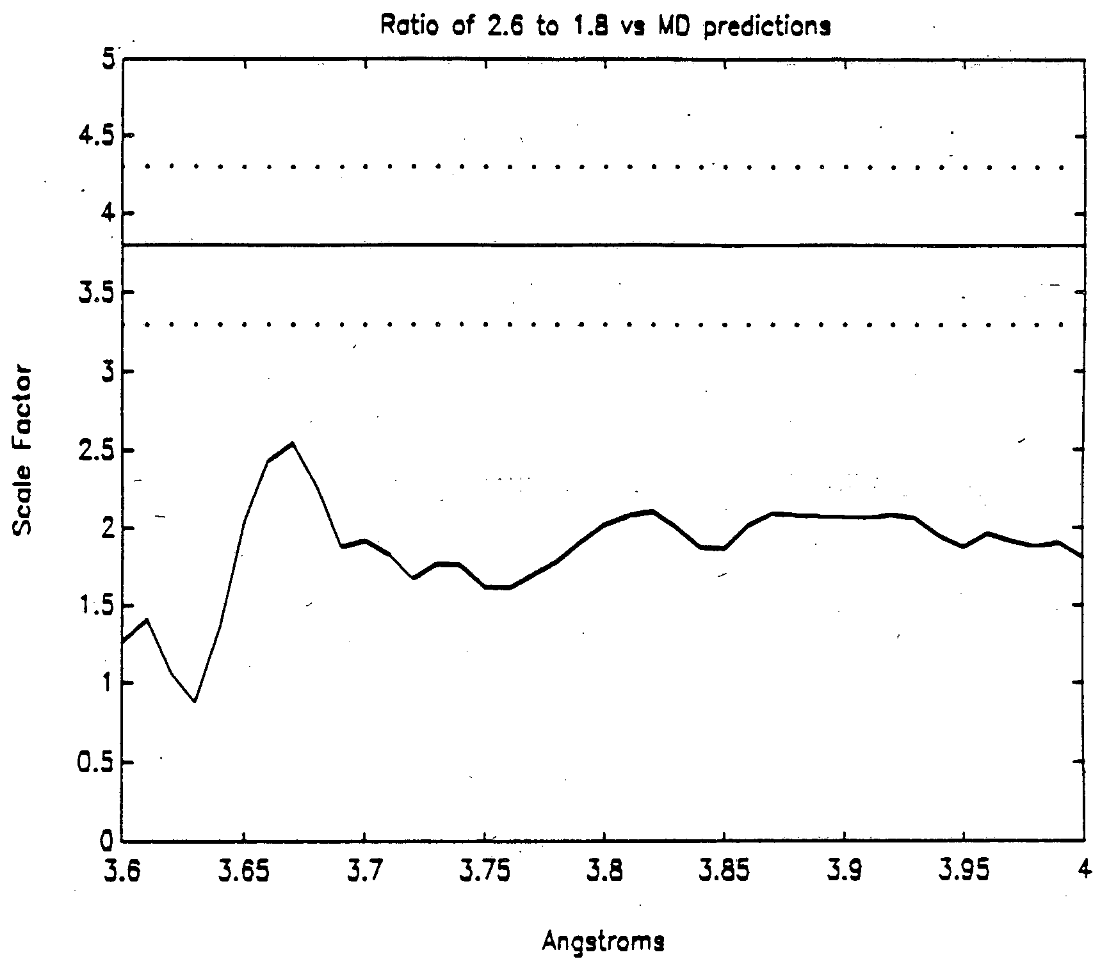


Figure 17 Scaling Predicted by Equation 19

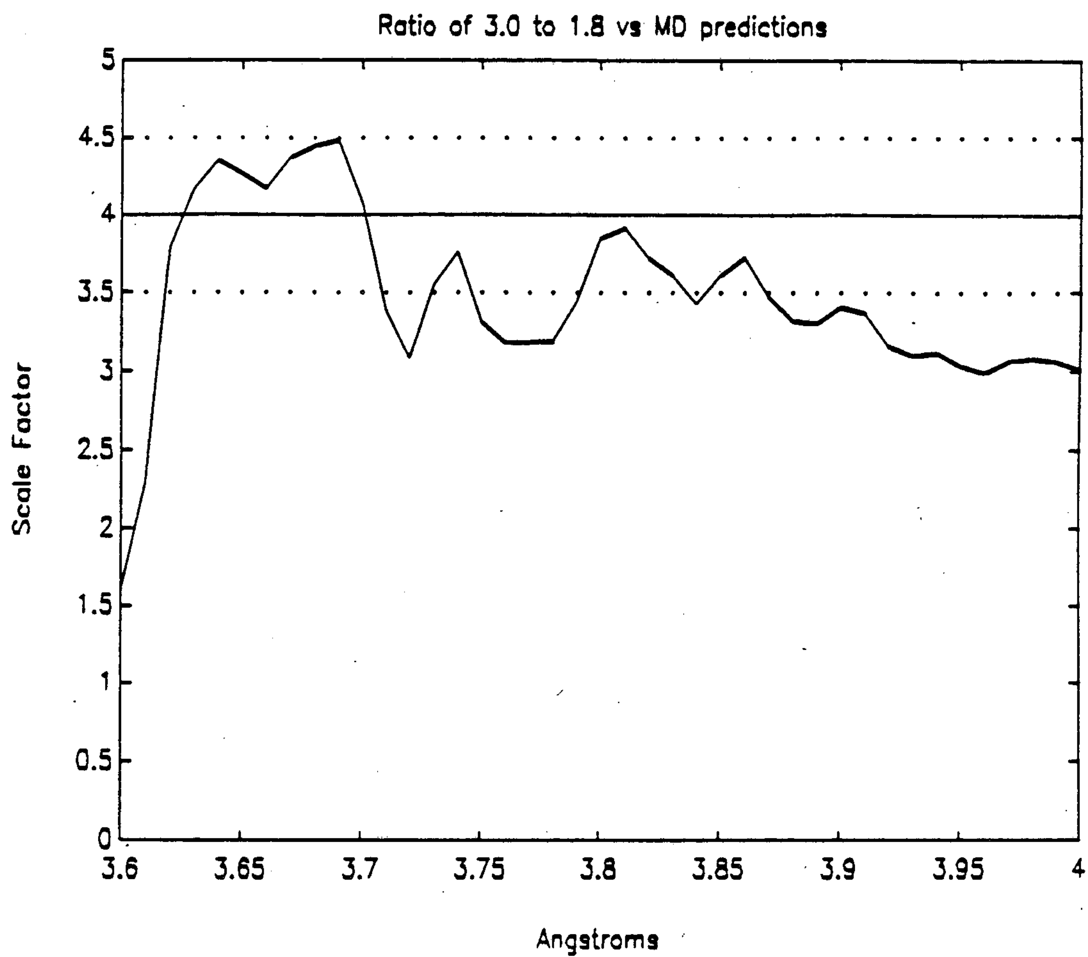


Figure 18 Scaling Predicted by Equation 19

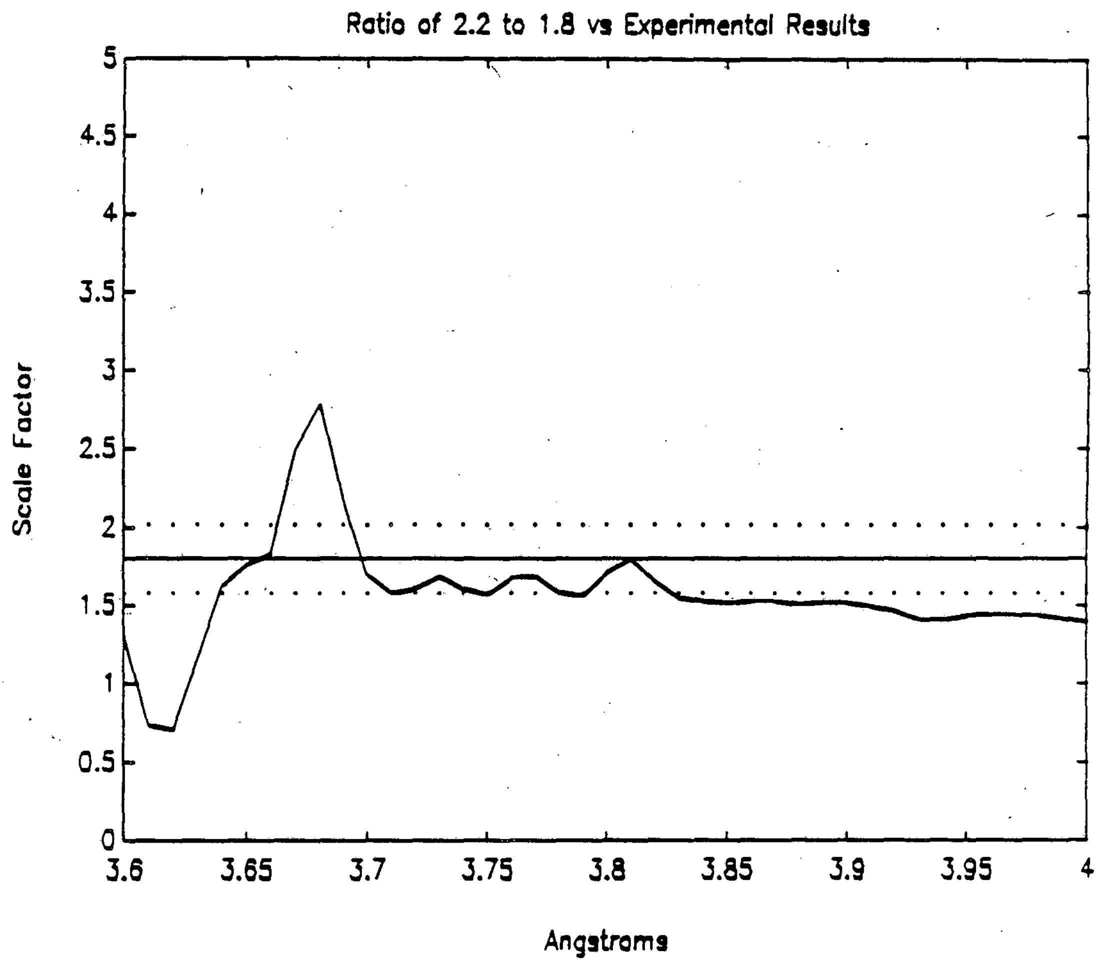


Figure 19 Scaling Predicted by Equation 19

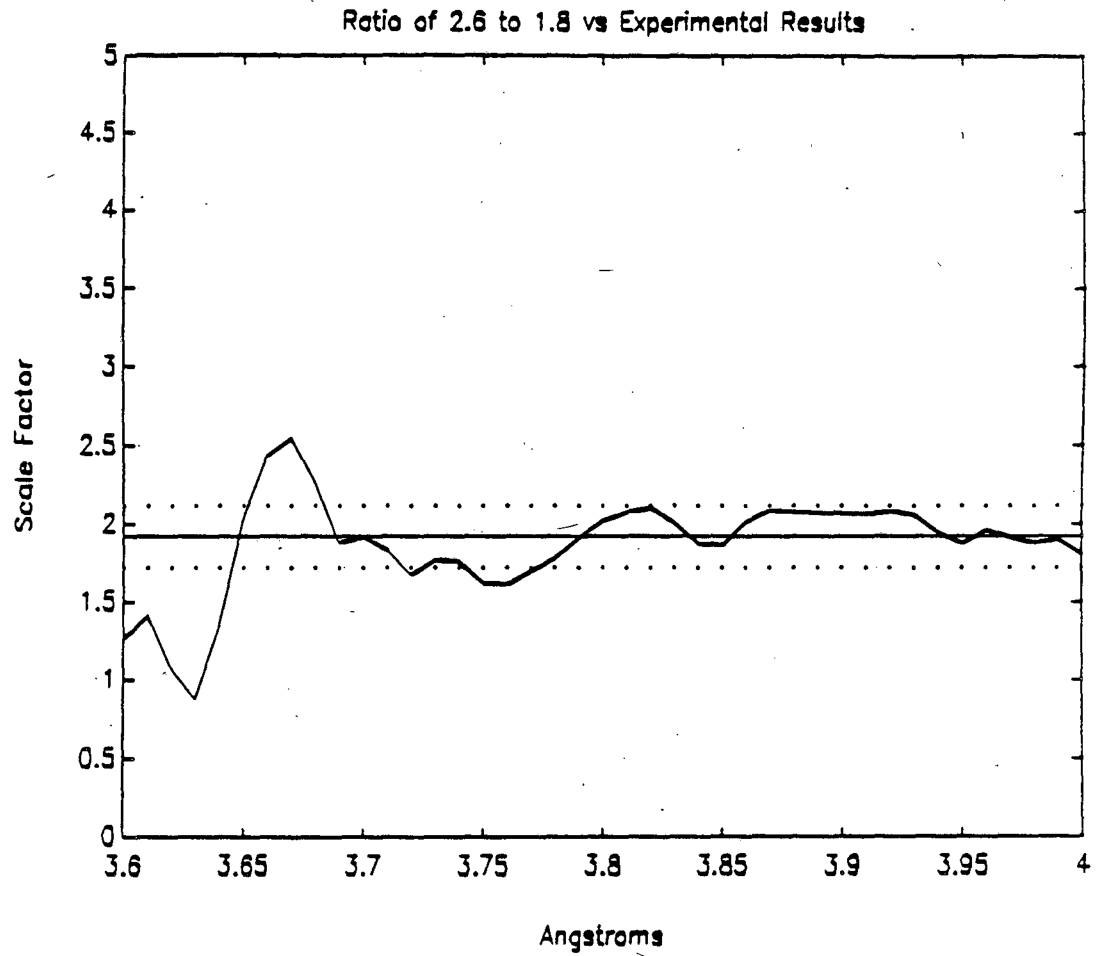


Figure 20 Scaling Predicted by Equation 19

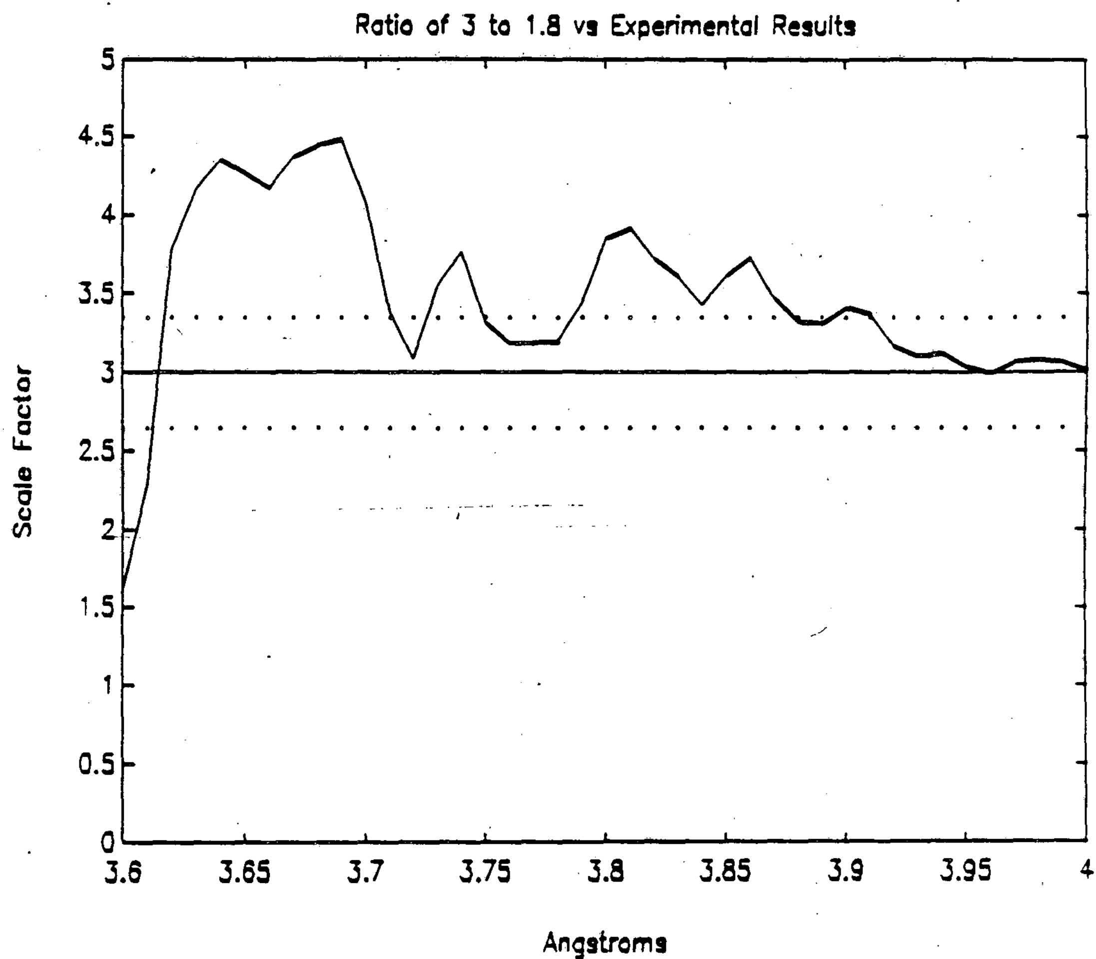
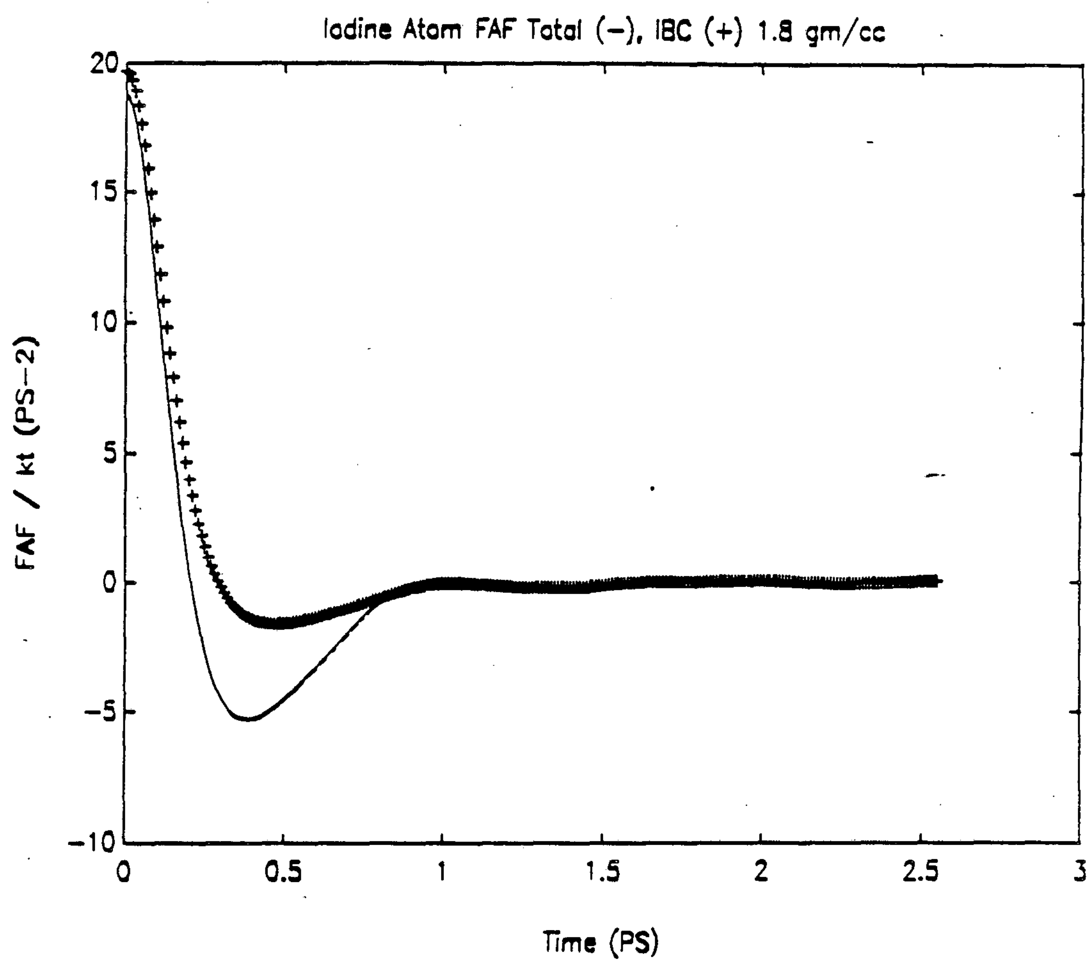


Figure 21 MD Prediction is the Solid Straight Line


Iodine Atom FAF Total (−), IBC (+) 1.8 gm/cc
FAF / kt (PS−2)
Time (PS)
20
15
10
5
0
−5
−10
0
0.5
1
1.5
2
2.5
3

early time components of the two autocorrelations are very similar. In figure 23 the power spectrum of the total force autocorrelation and the binary autocorrelation function are plotted. As Chesnoy had found, the binary force autocorrelation function frequency spectrum was very similar to the total force autocorrelation function frequency spectrum, in this case down to frequencies of ≈ 50 cm^{-1}.[31] Chesnoy found that the spectrums were the same down to ≈ 10 cm^{-1}. The results are slightly different in part probably due to the use of different potentials. This is evidence for the appropriateness for using IBC theory to model the vibrational relaxation even though I_2 has such a low vibrational frequency. The implication being that the many body forces are unimportant, if the magnitudes of the frequency spectrum for the total and isolated force spectrum are the same.

In figures 24-29 the force autocorrelation functions and power spectrums for densities 2.2, 2.6 and 3.0 g/cc are shown. Unfortunately equation 22 does not do a good job predicting relaxation rates. In figures 30-35 predictions for scale factors for equation 22 vs both experimental and MD results are plotted. The straight line across is the experimental or MD result for the scale factor with the dotted line showing a standard deviation on the prediction. Unlike the scaling results from the radial distribution function, both the experimental and MD results are not predicted well by the spectrum of the force autocorrelation function. The

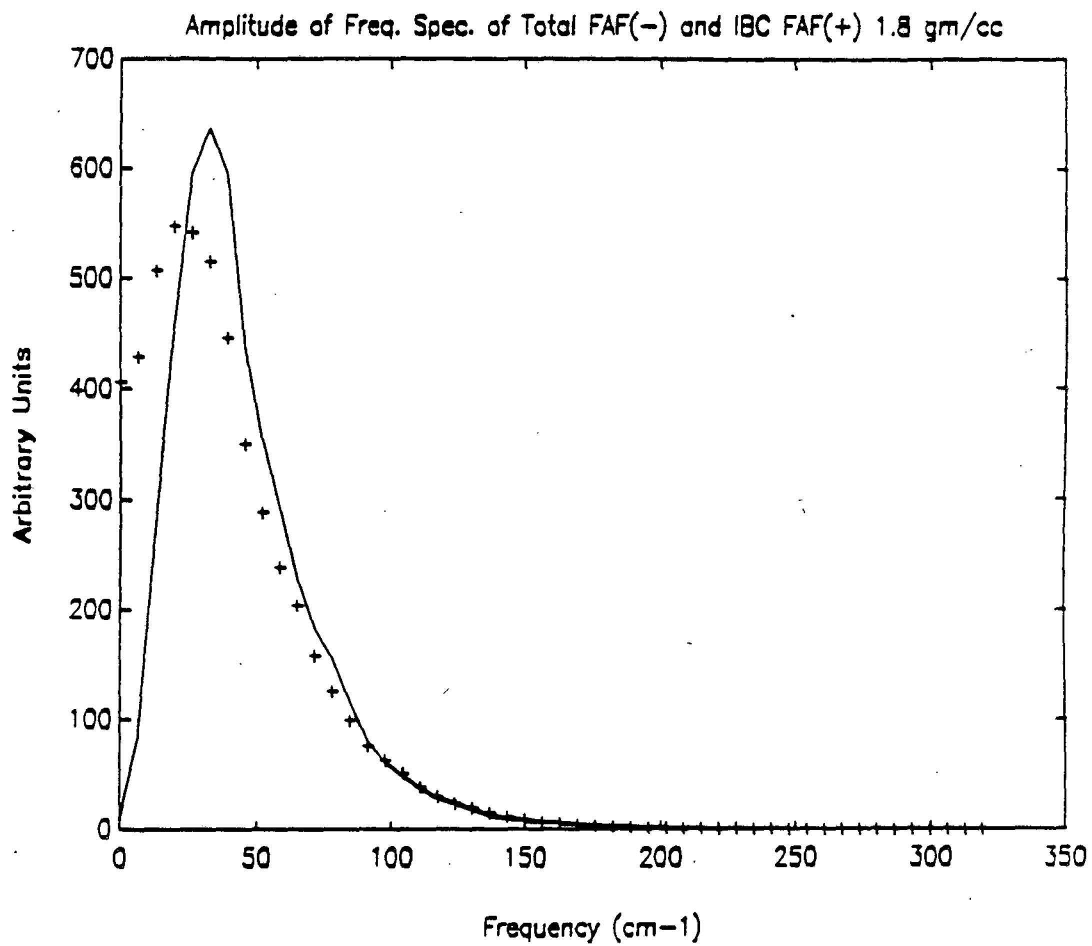


Figure 23 An Iodine Atom in Xenon

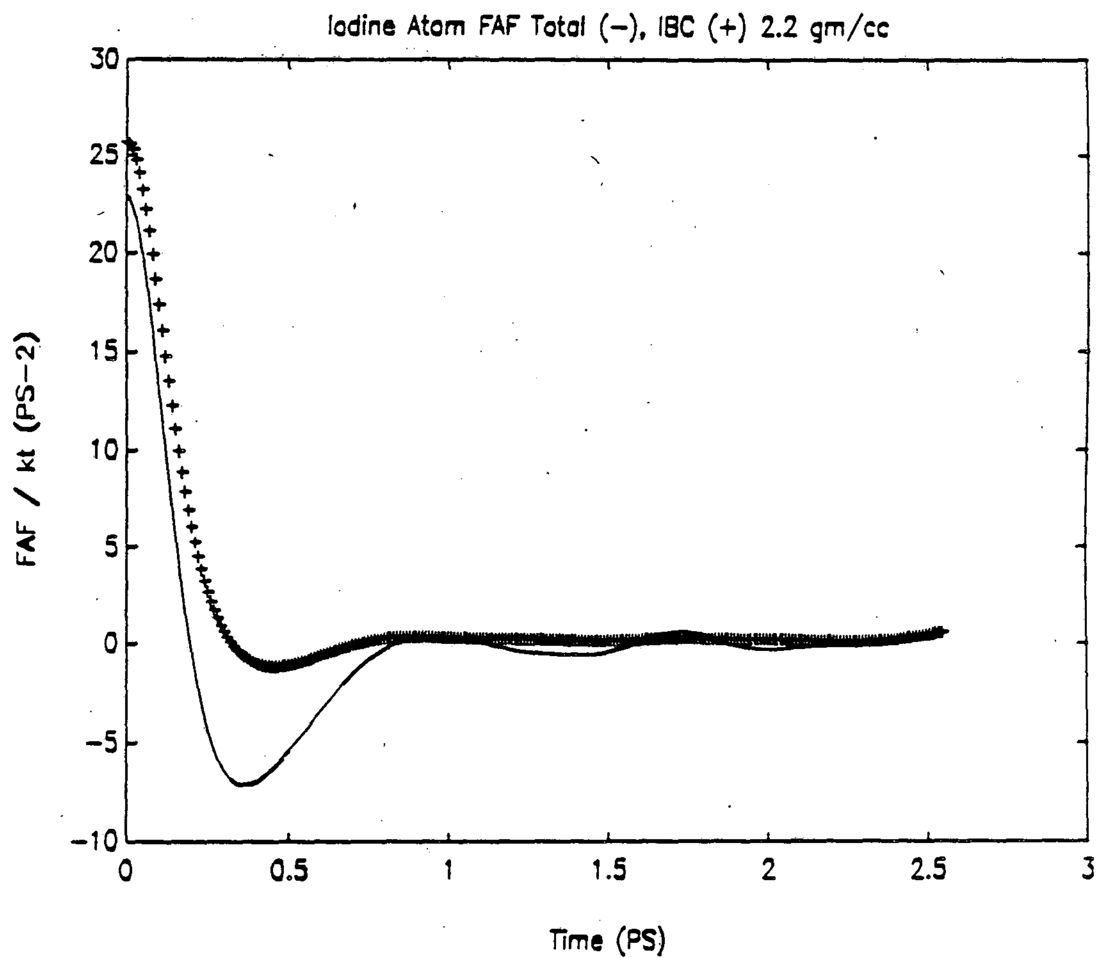


Figure 24 Force Autocorrelations for an Iodine Atom in Xenon

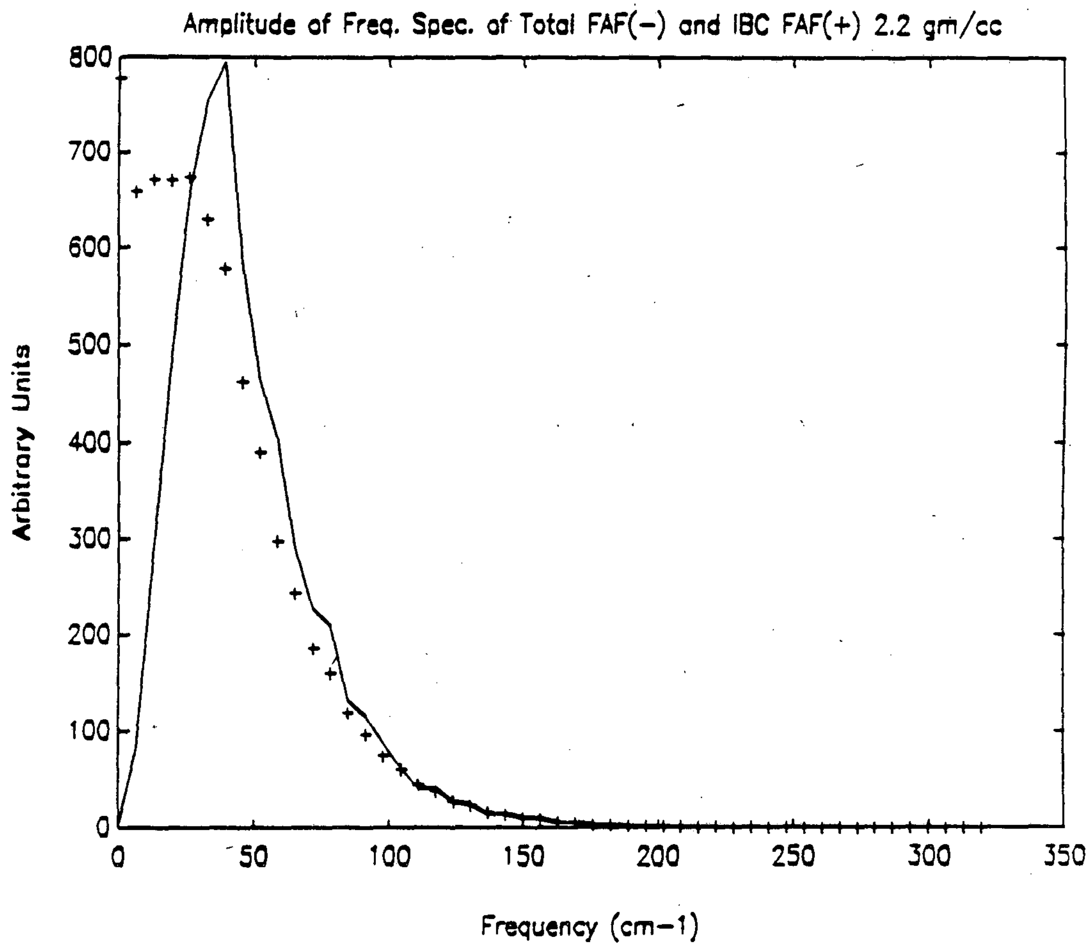


Figure 25 An Iodine Atom in Xenon

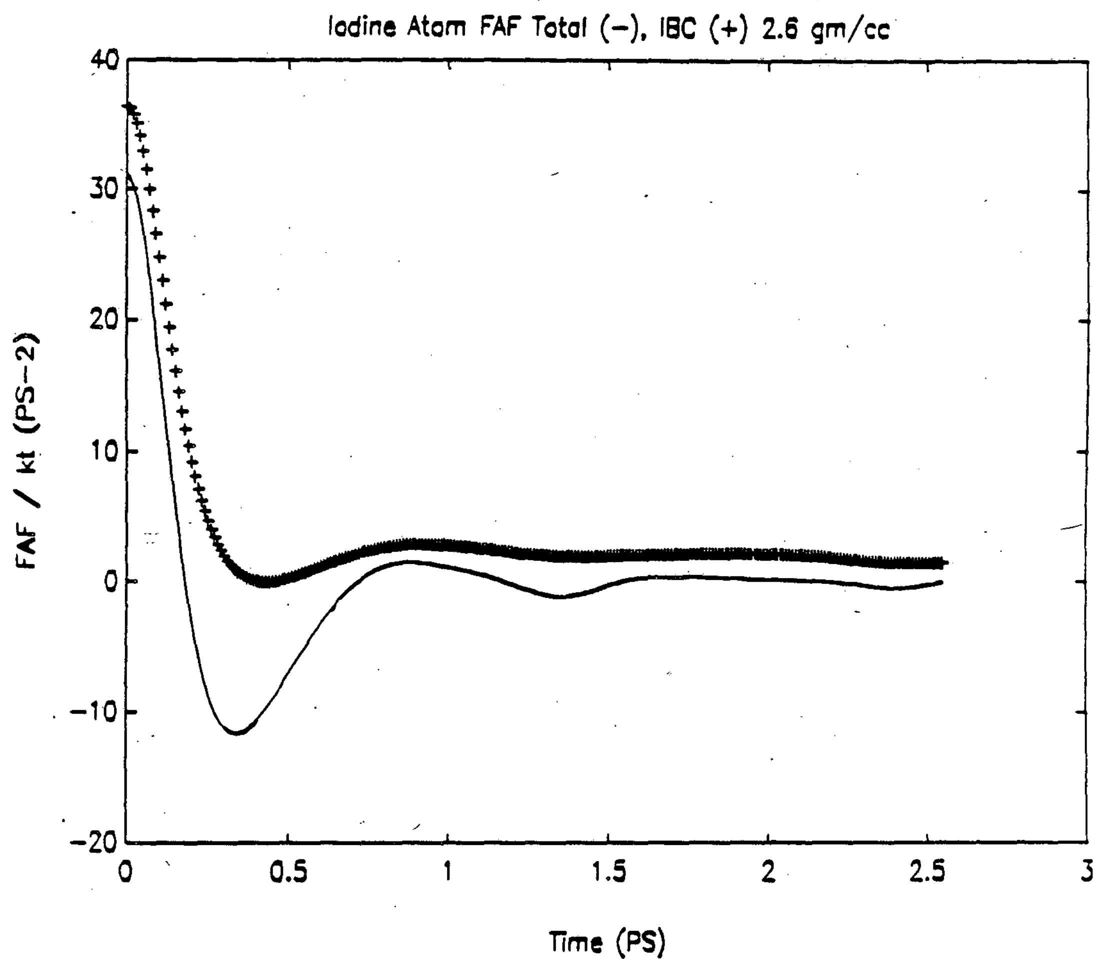


Figure 26 Force Autocorrelations for an Iodine Atom in Xenon

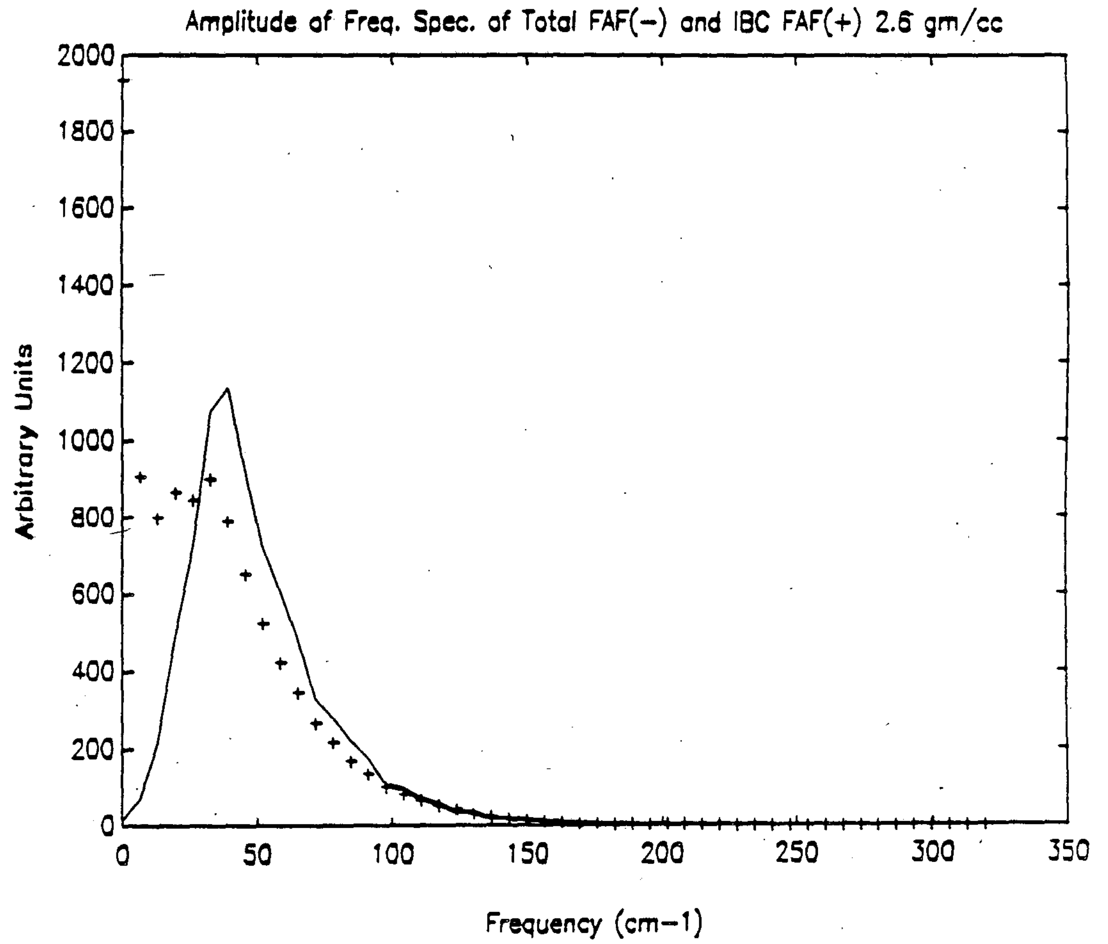


Figure 27 An Iodine Atom in Xenon

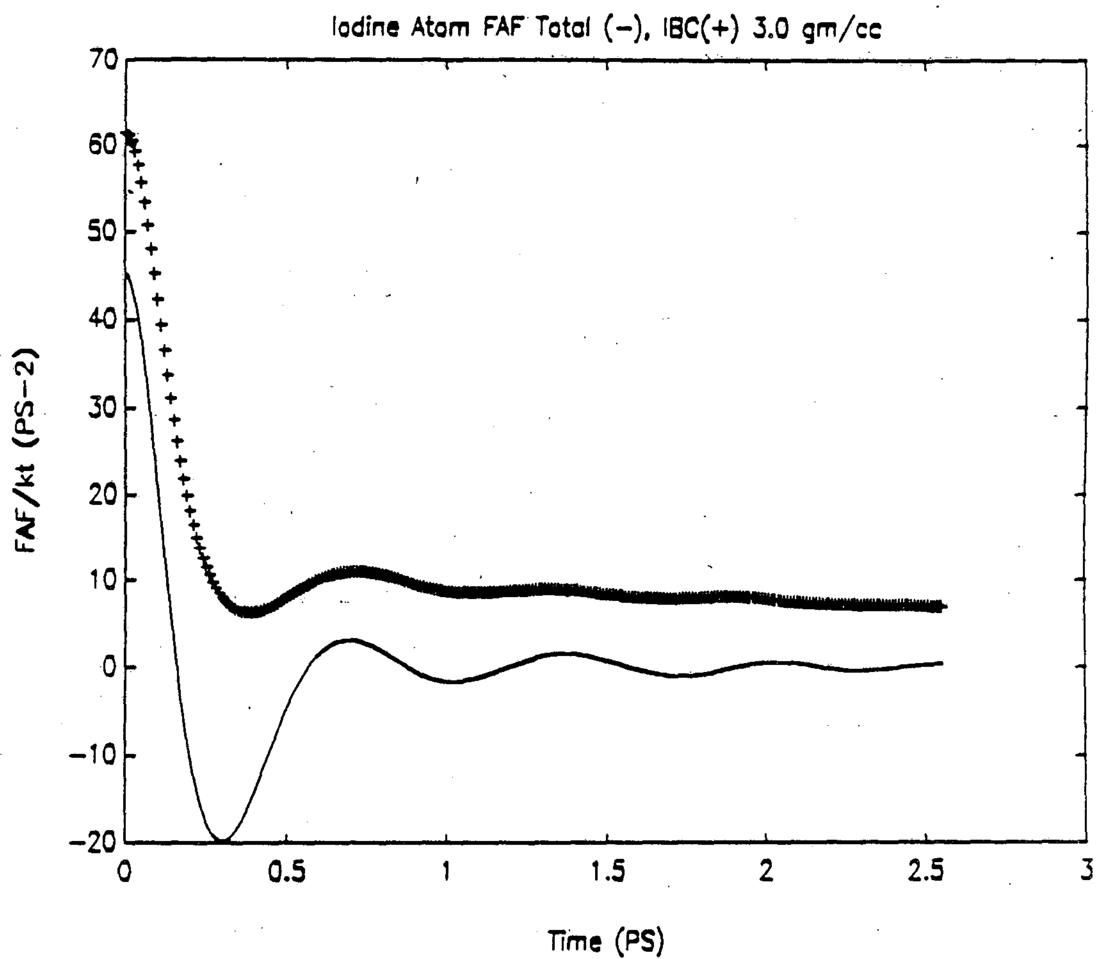


Figure 28 Force Autocorrelations for an Iodine Atom in Xenon

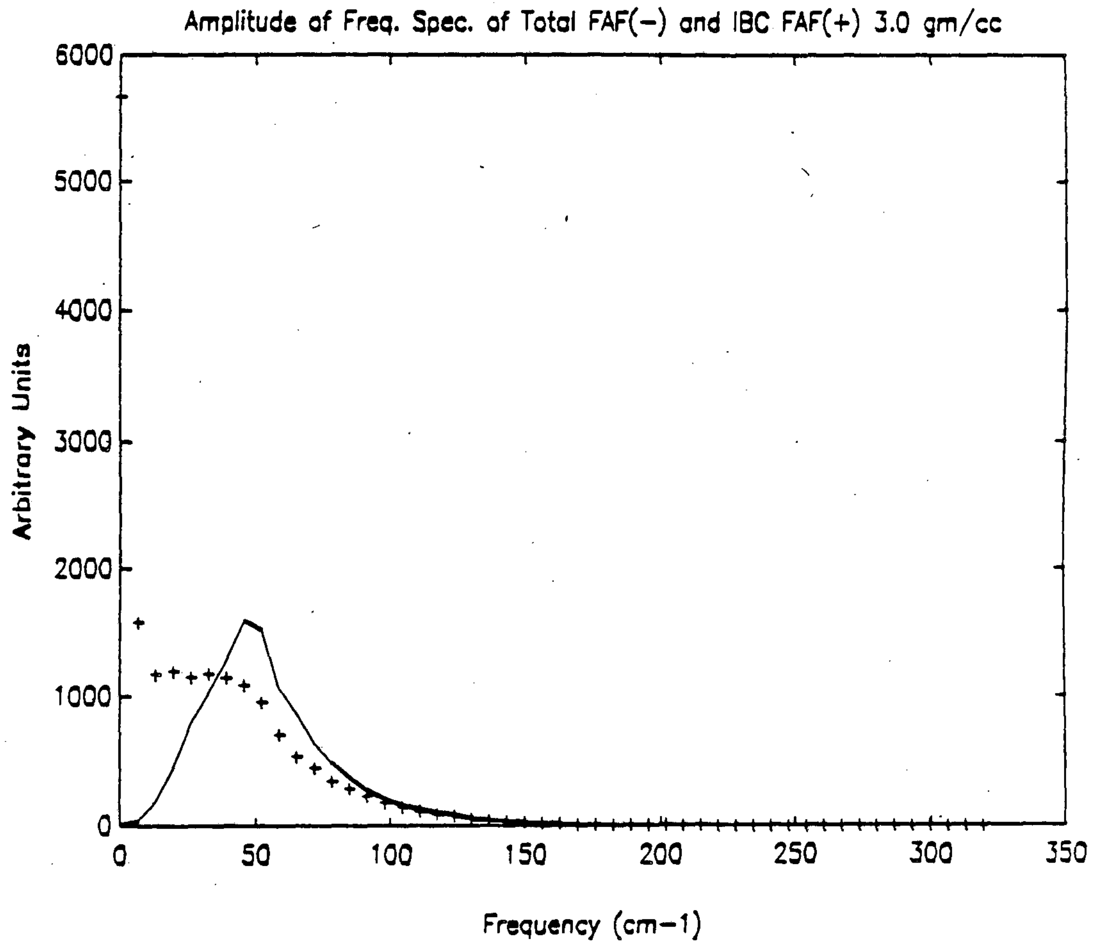


Figure 29 An Iodine Atom in Xenon

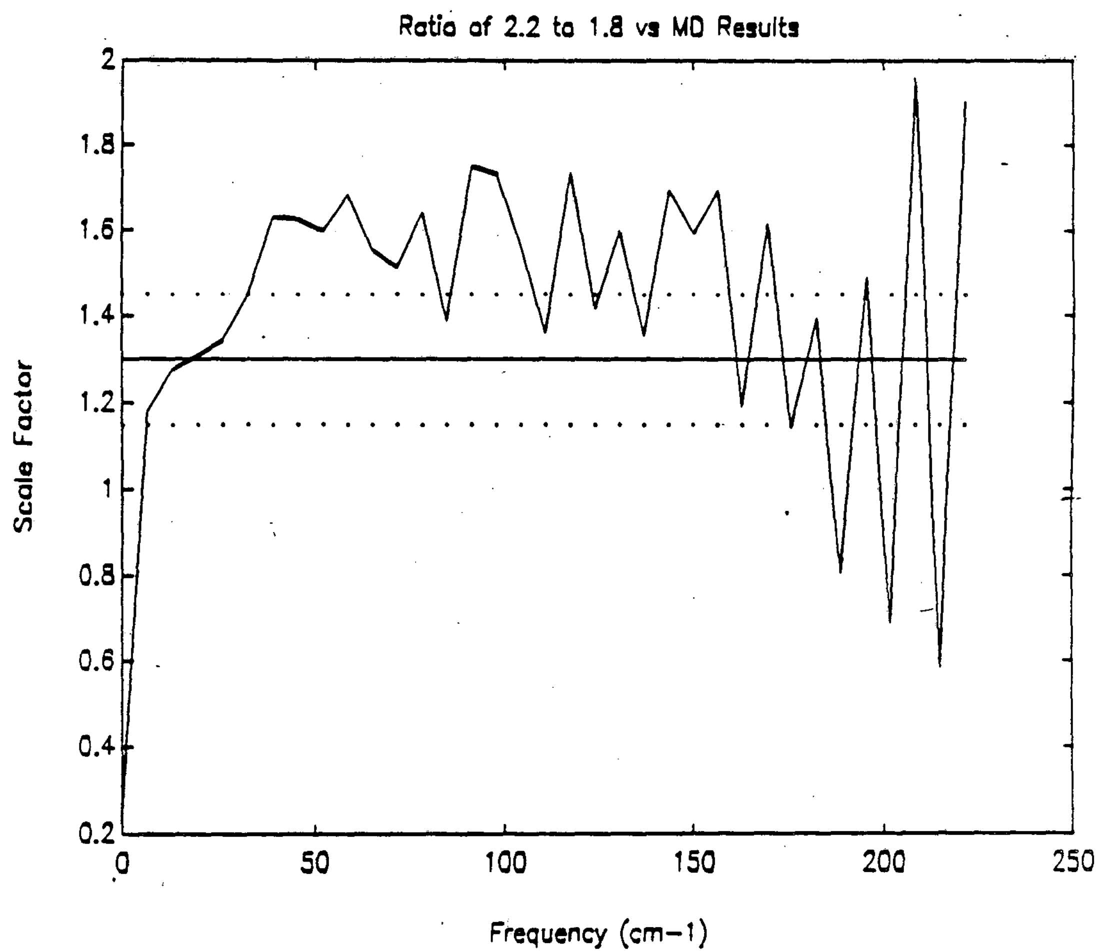


Figure 30 Scale Factor Using Equation 22

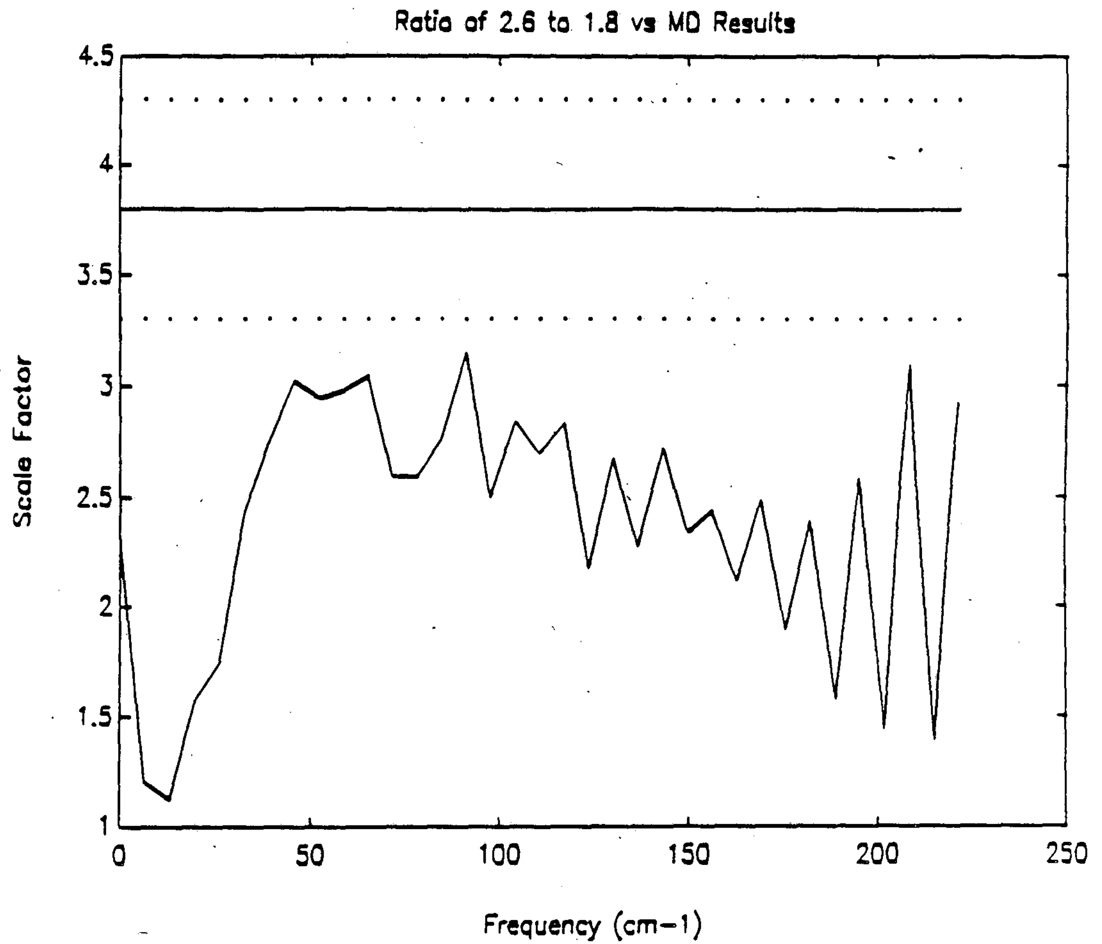


Figure 31 Scale Factor Using Equation 22

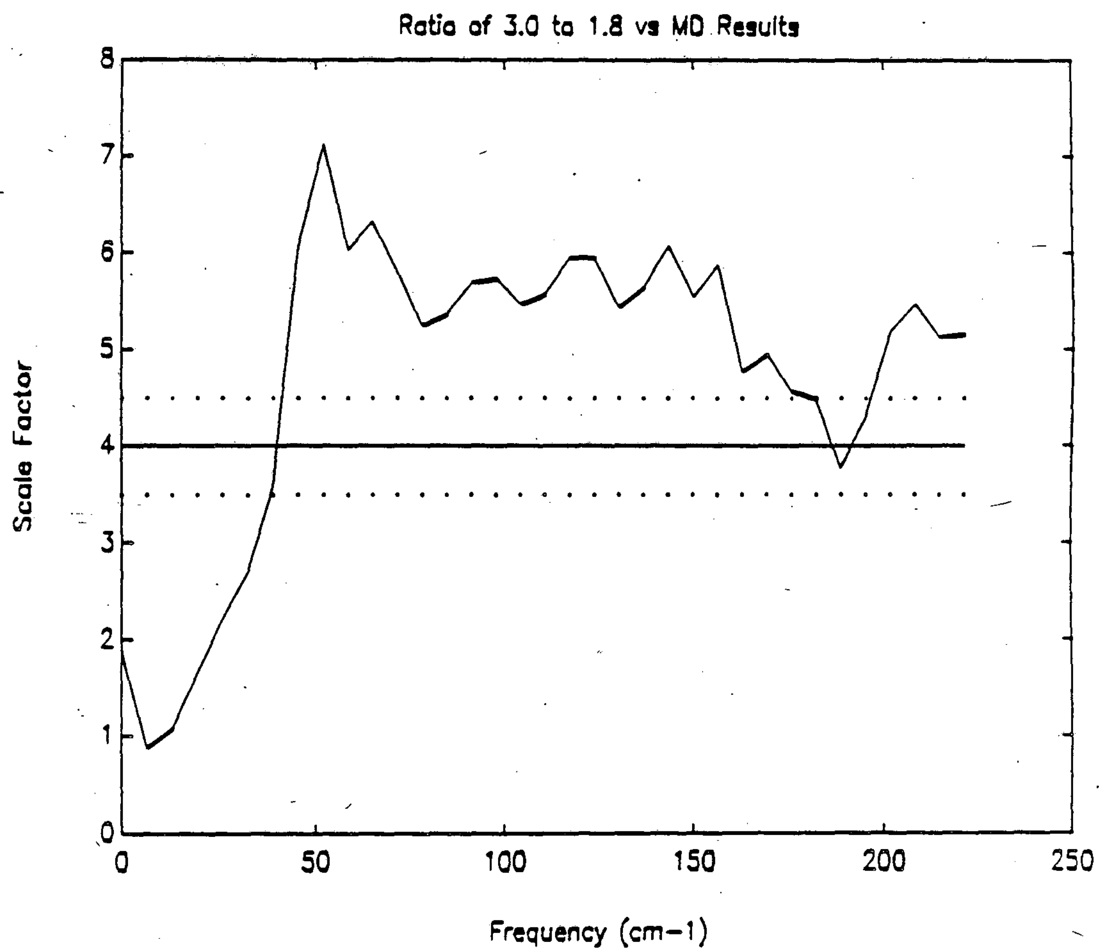


Figure 32 Scale Factor Using Equation 22

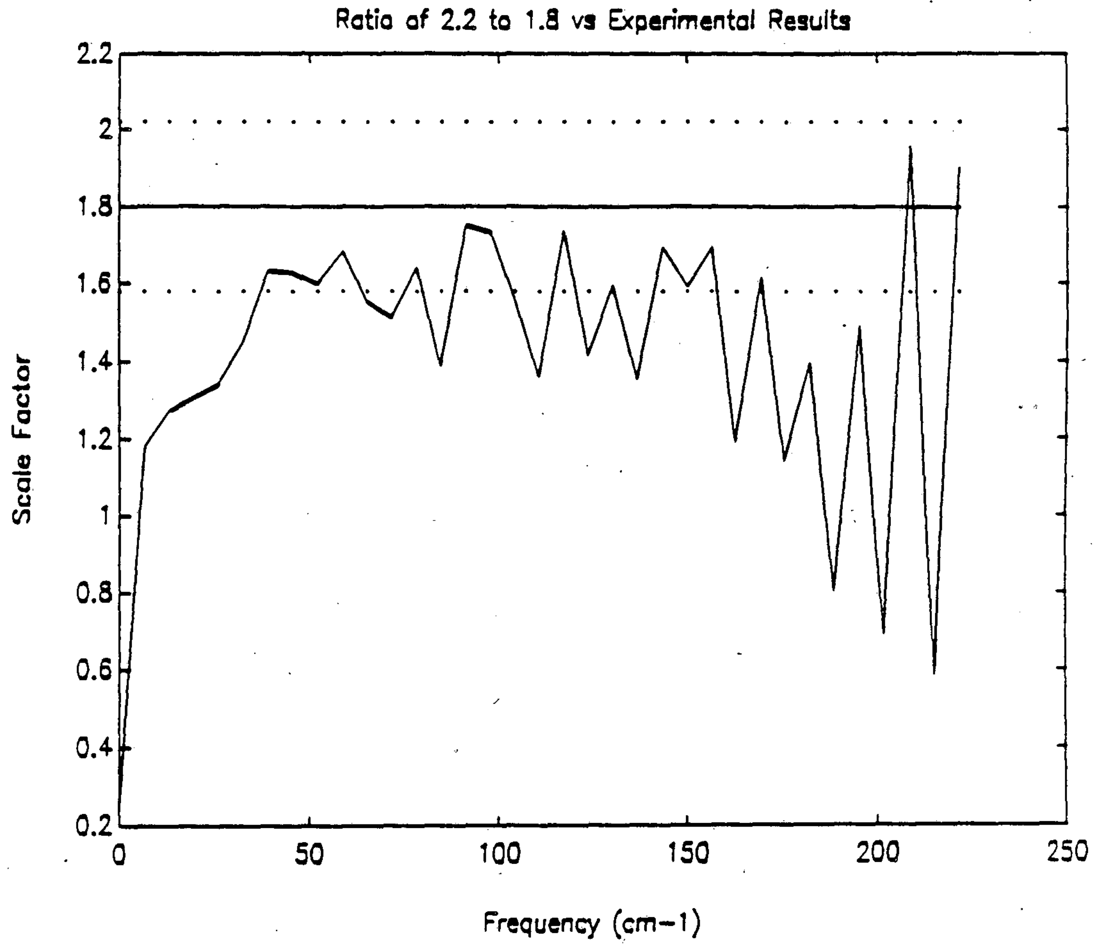


Figure 33 Scale Factor Using Equation 22

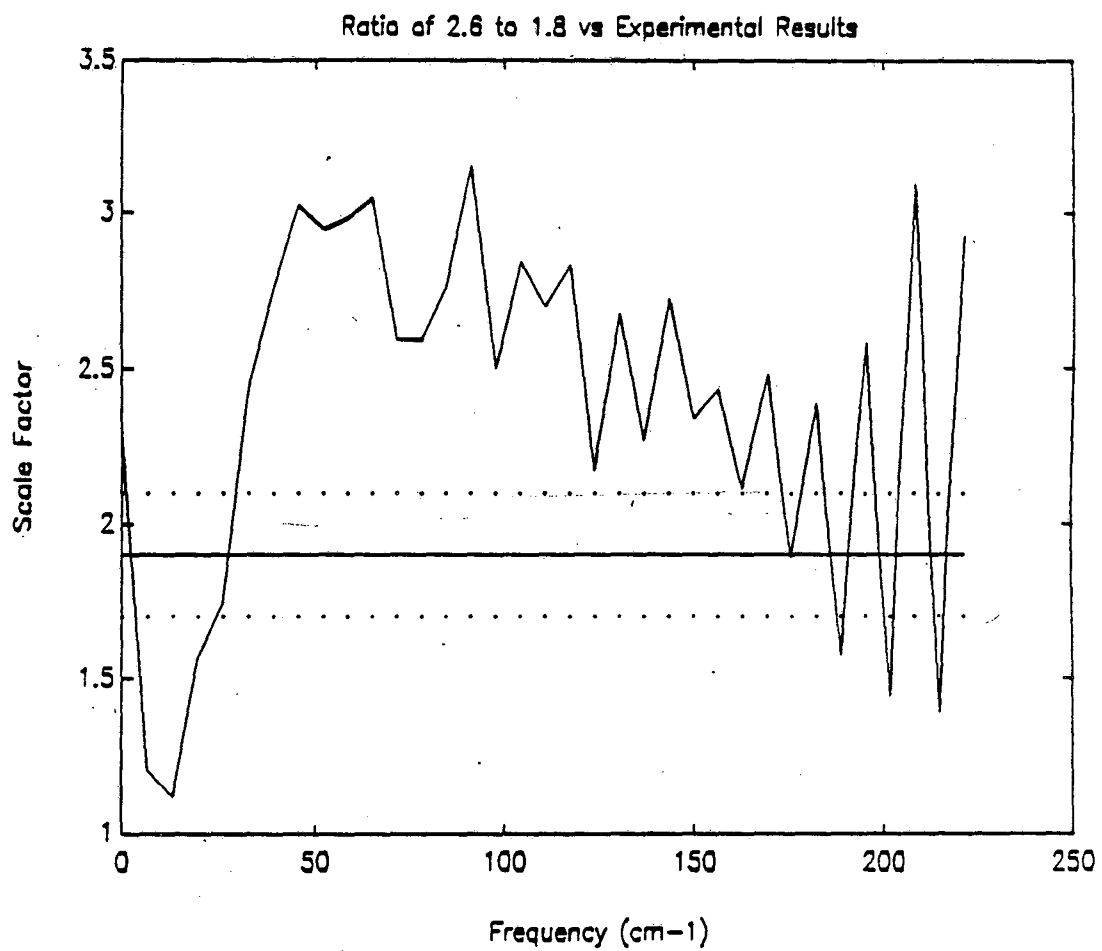


Figure 34 Scale Factor Using Equation 22

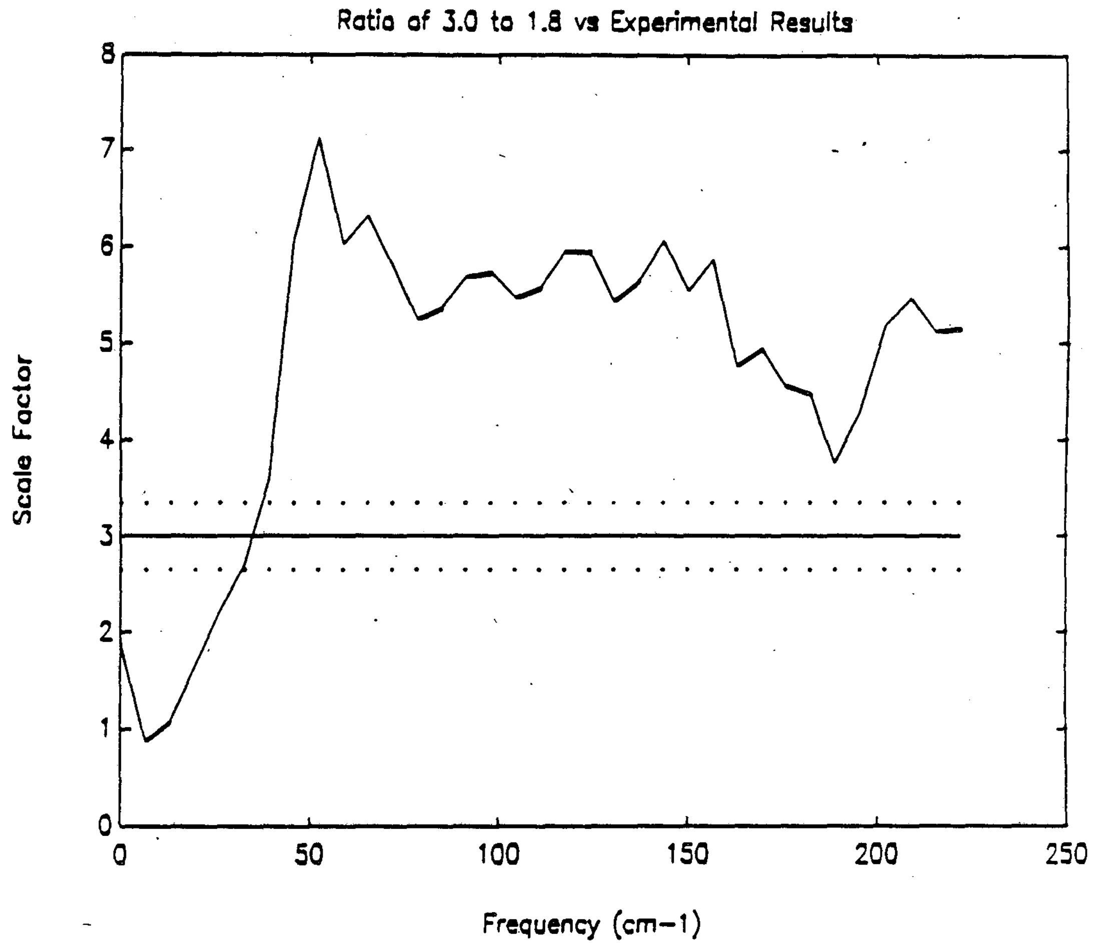


Figure 35 Scale Factor Using Equation 22

experimental values are comparable but the agreement is not good at 3.0 g/cc.

There is one factor that is not taken into account in equation 22. There is a phase factor that will influence the energy dissipation. For example, if the iodine molecule is vibrating while the xenon atoms are at rest there will be a component of force at the vibrational frequency. However, there will be no dissipation of energy due to this elastic interaction. Brown et al calculated the phase factors for the four densities studied and there was a trend for the phase factor to be more dissipative for the lower density systems. If the phase factors are applied to equation 22, the MD and experimental results compare much better (see table 2).[14]This seems to imply that the phase factors must be included as a function of density for the real experiment and the MD. This leaves the results in the ambiguous state of being partially correct, neither proven nor disproved. If phase factors must be include to describe the relaxation the first assumption of IBC theory is disproved. A density dependent phase relationship is equivalent to a density dependent P_{ij}.

Density	MD Scale Factors	Exp. Scale Factors	Scale Factor Eq. 22	Phase14 1.8 g/cc phase= 26° ± 3°	Scale Factor with Phase
2.2 g/cc	1.3 ± .15	1.8 ± .22	1.4 ± .3	22° ± 3°	1.2
2.6 g/cc	3.8 ± .5	1.9 ± .2	2.5 ± .5	19° ± 3°	1.9
3.0 g/cc	4.0 ± .5	3.0 ± .35	5.2 ± 1	18° ± 4°	3.7

Table 2

IV. Summary and Conclusions

The state of the theory of vibrational relaxation at this point must be driven by experiments. Due to the observation that relaxation is very sensitive to the potentials, all the applications of IBC examined in this paper are useful only in predicting the density and temperature dependence of relaxation. Unfortunately, there have been very few experiments that provide this information in a system that can be easily modeled. It seems that the probability of relaxation, P_{ij}, is fairly independent of potential over a realistic range. In comparisons to liquid relaxation the steric factor seems to have a quantitative effect on the relaxation rate. However, the steric factor seems independent of density and not a problem with respect to predicting density dependencies. In the classical regime I_2 in liquid Xe has been studied extensively in theory and experiment. This system has been modeled by most of the theories examined. The most successful model of the relaxation was based on the generalized Langevin equation due to the inclusion of many body effects. However the generalized Langevin model has only been applied to the densities of 1.8 g/cc and 3.0 g/cc. This is unfortunate, because IBC does a good job on the MD results for 1.8 g/cc, 2.2 g/cc, and 3.0 g/cc but not at 2.6 g/cc using both the radial distribution function as a scale factor and using the frequency spectrum of the force autocorrelation

function. The failure at 2.6 g/cc may be due to a failure of IBC or a problem with bad statistics on the MD calculations due to the small number of runs completed. The interpretation may have been clearer if the generalized Langevin model had been tried at these densities to see if it would work for all densities. This would not have proved IBC's correctness for this system, but if the generalized Langevin equations' results disagreed with MD then perhaps there is something wrong with either the MD or there is something peculiar about 2.6g/cc Xe that does not allow either the generalized Langevin equation or IBC to model the relaxation. IBC has reproduced the density trends seen in the relaxation of I_2 in Xe experiments, using the radial distribution function as a scaling factor, but not using the power spectrum of the force autocorrelation function. This may not be too disconcerting considering the possibility of the I-Xe Lennard-Jones may not be a good representation of the real potential, and would have more effect on the power spectrum than the radial distribution function.

Both the IBC and Langevin approaches will fail if the coupling between the bath and oscillator is strong. IBC theory will also fail if the binary force autocorrelation function power spectrum at the appropriate frequency is not the same as the total force autocorrelation. The most probable reason for the two force autocorrelation functions not being the same is if many body effects become more

important and provide damping at the oscillators frequency. The final reason for failure for either of the above models is the lack of phase information. If the iodine molecule is driving the collisions, there may be a different average phase relationship for the real system and the two model systems. The IBC calculation assumes a random phase approximation, and the average phase relationship is determined by the xenon and iodine velocities. This is incorporated in the P_{ij} and is density independent. The generalized Langevin equations phase vs force or collision may also be different than the MD or real system, and may be density dependent. Brown et al have seen an average phase shift over the various densities studied that may affect the vibrational relaxation, and these can be incorporated, but IBC is a failure if in reality they are density dependent.

IBC theories have been somewhat successful in modeling quantum systems. The ease in applying IBC has made it most prevalent, however it has been applied in detail differently in many experiments. This difference in application from experiment to experiment is partially due to the lack of a firm theoretical foundation for IBC. This weakness causes the theory to still be attacked theoretically. This work does not prove IBC, on the other hand considering the approximations made in implementation the results are not discouraging. It has done a surprisingly good job describing the density dependence. Other theories to model quantum systems have been

confined to semiclassical calculations, however few of these calculations have been made because of their relative difficulty. Until the relaxation of simple oscillators in simple solvents are understood it seems that relaxation mechanisms and dynamics of chemical relaxation in larger molecules in molecular solvents will still be a major challenge to experimentalist and theorist alike.

References

1. J. Frank and E. Rabinowitch, Trans. Faraday Soc. **30**, 120, (1934)

2. E. Rabinowitch and W. C. Wood, Trans. Faraday Soc., **30**, 547, (1936).

3. E. Rabinowitch and W. C. Wood, Trans. Faraday Soc. **32**, 1381 (1936).

4. See for example R. M. Noyes, Prog. React. Kin. **1**, 128, (1961).

5. T. J. Chuang, G. W. Hoffman, and K. B. Eisenthal, Chem. Phys. Lett. **25**, 201 (1974).

6. D. L. Bunker and B. S. Jacobsen, J. Am. Chem. Soc. **94**, 1843, (1972).

7. J. N. Murrell, A. J. Stace, and R. Dammel, J. Chem. Soc. Faraday Trans., **74**, 1532, (1978)

8. D. J. Nesbitt and J. T. Hynes, J. Chem. Phys. **77**, 2130 (1982).

9. D. E. Smith and C. B. Harris, J. Chem. Phys, **87**, 2709, (1987).

10. M. Berg, A. L. Harris, J. K. Brown and C. B. Harris Ultrafast Phenomena IV. Eds. D. H. Auston and K. B. Eisenthal, Springer Series in Chemical Physics 38, p. 300, (Springer-Verlag Publishers) 1984.

11. M. Berg, A. L. Harris and C. B. Harris, Phys. Rev. Lett., **54**, 951, (1985).

12. A. L. Harris, M. Berg and C. B. Harris, J. Chem. Phys. **84**, 788, 1986.

13. M. E. Paige, D. J. Russell, and C. B. Harris, J. Chem. Phys. **85**, 3699, (1986)

14. J. K. Brown, C. B. Harris, and J. C. Tully, J. Chem. Phys., **89**, 6687, (1988).

15. D. F. Kelly and P. M. Rentzepis, Chem. Phys. Lett., **85**, 85 (1982).

16. D. E. Smith and C. B. Harris, J. Chem. Phys., **92**, 1304, (1990).

17. D. E. Smith and C. B. Harris, J. Chem. Phys., **92**, 1312, (1990).

18. K. F. Herzfeld and T. A. Litovitz, Absorption and Dispersion of Ultrasonic Waves, Academic, New York, 1959

19. W. M. Madigosky and T. A. Litovitz, J. Chem. Phys., **34**, 489, (1961).

20. M. Fixman, J. Chem. Phys., **34**, 369,(1961).

21. R. Zwanzig, J. Chem. Phys., **34**, 1931, (1961).

22. K. F. Herzfeld, J. Chem. Phys., **36**, 3305, (1962).

23. K. F. Herzfeld, J. Chem. Phys., **36**, 3305, (1961).

24. R. Zwanzig, J. Chem. Phys., **36**, 2227, (1962).

25. P. K. Davis and I. Oppenheim, J. Chem. Phys., **57**, 505, (1972).

26. P. K. Davis and I. Oppenheim, J. Chem. Phys., **56**, 86, (1972).

27. T. Einwohner and B. Alder, J. Chem. Phys., **49**, 1458, (1968).

28. C. Delalande and G. M. Gale, J. Chem. Phys., **71**, 4804, (1979)

29. W. F. Calaway and G. E. Ewing, Chem. Phys. Lett., **30**, 485, (1975).

30. W. F. Calaway and G. E. Ewing, J. Chem. Phys., **63**, 2842, (1975).

31. J. Chesnoy and J. J. Weis, J. Chem. Phys., **84**, 5378, (1986).

32. P. S. Dardi and R. I. Cukier, J. Chem. Phys., **86**, 2264, (1987).

33. P. S. Dardi and R. I. Cukier, J. Chem. Phys., **86**, 6893, (1987).

34. P. S. Dardi and R. I. Cukier, J. Chem. Phys., **89**, 4145, (1988).

35. F. H. Mies, J. Chem. Phys., **40**, 523, (1964).

36. R. E. Roberts, J. Chem. Phys., **55**, 100, (1971).

37. J. T. Knudtson and E. Weitz, Chem. Phys. Lett., **104**, 71, (1984).

38. J. Chesnoy, J. Chem. Phys., **83**, 2214, (1985).

39. L. Landau and E. Teller, Physik Z. Sowjetunion, **10**, 34, (1936).

40. R. N. Schwartz, Z. I. Slawsky, and K. F. Herzfeld, J. Chem. Phys., **20**, 1591 (1952).

41. J. M. Jackson and N. F. Mott, Proc. R. Soc. A137, 703, (1932).

42. K. Takayanagi, Progr. theor. Phys., **8**, 497, (1952).

43. R. N. Schwartz, and K. F. Herzfeld, J. Chem. Phys., **22**, 767, (1954).

44. J. D. Lambert, Vibrational and Rotational Relaxation In Gases, Oxford, 1977.

45. D. Rapp and T. Kassal, Chem. Rev., **69**, 61, (1969).

46. N. A. Abul-Haj and D. F. Kelly, J. Chem. Phys. **84**, 1335, (1986).

47. R. L. McKenzie, J. Chem. Phys., **66**, 1457, (1975).

48. D. Chandler, J. D. Weeks, and H. C. Anderson, Science **220**, 778

49. The exact equation used for the ground state of iodine and the xenon-xenon interaction in the Molecular Dynamics by Brown and Harris is

$$V(R) = 4\epsilon[(\frac{\sigma}{r})^{12}-(\frac{\sigma}{r})^{6}]+c, \qquad r\leq 2.5\sigma$$

$$=0, \qquad r>2.5\sigma.$$

50. D. Beeman, J. Comp. Phys. **20**, 130, (1976).

51. J. T. Yardley, Introduction to Molecular Energy Transfer, (Academic, New York, 1980).

52. H. K. Shin, Dynamics of Molecular Collisions Part A, Ed. by W. H. Miller (Plenum, New York, 1967).

53. C. Delalande and G. M. Gale, J. Chem. Phys., **71**, 4531, (1979).

54. N. F. Carnahan and K. E. Starling, J. Chem. Phys., **51**, 635, (1969).

55. P. A. Madden and F. van Swol, Chem. Phys., **112**, 43, (1987).

56. D. W. Oxtoby, Mol. Phys., **34**, 987, (1977).

V. APPENDIX

A. PROGRAM LISTING I2IBC

```
c          Program "I2IBC"
c          Daniel Russell
c          August 2 1990
c
c          This program calculates a one dimensional trajectory
c          for one Iodine molecule and one Xenon atom.  The
c          energy before and after a collision occurred were
c          calculated giving the relative gain or loss of
c          energy after a collision.  This was done for 15
c          maxwell boltzman distributed relative velocities of
c          the Iodine molecule and xenon atom and 100 phases of
c          the iodine atom.  This gives 1500 trajectories for
c          every Iodine vibrational level that was run.


        integer ntm3, i, stop
        parameter(ntm3=45)

c          ntm3 is 3 * the number of particles.  In this
c          calculation 15 trajectories are run simultaneously.

        double precision vxyz(ntm3), rxyz(ntm3), axyz(ntm3)

c          vxyz, rxyz, and axyz are the velocity, position, and
c          the acceleration respectively.  vxyz(1) is the
c          velocity of the first iodine atom, vxyz(2) is the
c          velocity of the second iodine atom, and vxyz(3) is
c          the velocity of the xenon atom that will collide
c          with that particular iodine molecule.  vxyz(4) is
c          the velocity of the first iodine atom that will be
c          running a parallel trajectory although it will have
c          a different phase and xenon velocity than the first
c          trajectory.

        double precision axyz2(ntm3)

c          axyz2 is a space for storing the accelerations at
c          the time step before for Beeman's integrator.

        double precision rxyz2(ntm3)

c          rxyz2 is a space for storing the positions at the
c          time step before for Beeman's integrator.

        double precision sig, eps
```

```
c         sig is the Iodine - Xenon sigma for a Lennard-Jones
c         potential, and eps is epsilon.  Note a Lennard-Jones
c         interaction is not used.  A Weeks Chandler Anderson
c         decomposition of a Lennard-Jones is used.
c         eps=225.0000*1.196265744 in the units of atomic mass
c         units * angstroms^2 per picosecond^2
c         sig=3.94

        double precision masi, masx, masi2, masx2

c         masx=131.3 Is the mass of Xenon
c         masi=126.90 Is the mass of Iodine
c         masi2=masi/2.
c         masx2=masx/2.

c         These are various constants that are calculated
c         once and used during the rest of the program.

        double precision cst5
        double precision cst6, cst7, cst9, cst10
        double precision cst12
        double precision cst13, cst14, cst15, cst16

c         h is the time step, h2 is h squared, h26 is h2/6 and
c         hi is one over h

        double precision h, h2, h26, hi

c         the array ep stores the potential energy's of the
c         15 separate trajectories at every third index
c         starting at one.  The array ek stores kinetic
c         energy.  sum stores the total energy in both iodine
c         and xenon.  sumi2i is iodine's initial energy
c         including center of mass motion.

        double precision ep(45), ek(45), sum(45), sumi2i(45)

c         sumi2 is the iodine's total energy, including
c         center of mass motion, calculated for every call to
c         energy.

        double precision sumi2(45)

c         plnk is Plank's constant. ri is a dummy constant
c         used in reading in a dummy constant.  the array v is
c         the initial relative velocities for the iodine xenon
c         collision that are chosen from a Maxwell Boltzman
c         distribution.  sumi2f is the final iodine kinetic
c         energy including center of mass velocity.

        double precision plnk, ri, v(15), sumi2f(45)
```

```
c          The array rel is calculated in the subroutine energy
c          and contains the relative kinetic energy in the
c          iodine's vibrational motion and it's potential
c          energy.  Note this does not include center of mass
c          motion of the iodine molecule.  The array reli
c          contains the initial relative energy in the iodine
c          molecule.

        double precision rel(45), reli(45)

c          w is the iodine's initial vibrational frequency that
c          is used in generating a random phase.  rcount is a
c          flag that indicates if a collision has occurred and
c          should really be an integer.  The array relf is the
c          final relative energy.

        double precision w, rcount, relf(45)

c          The array r contains the distance between the two
c          iodine atoms, the distance between the second iodine
c          atom and the xenon atom, and the distance between
c          the second iodine atom and the xenon atom to the
c          inverse sixth power for all 15 trajectories.  rmax
c          contains the largest distance between two iodine
c          atoms for a particular vibrational energy.  This is
c          to make sure that the xenon atom starts out at a
c          distance where it will not be interacting with the
c          iodine molecule until it has had a chance to begin
c          it's trajectory.  The array viben contains the first
c          100 vibration energies for the iodine molecule.

        double precision r(45), rmax, viben(100)

        common /eng/ ek, ep, sumi2, rel, rmax, w, sum, rcount,
     $cst15, cst16
        common /blk2/ h, h2, h26, hi
        common /blk4/ vxyz
        common /blk5/ rxyz, rxyz2, axyz, axyz2
        common /blk9/ natom, natom1, natom3
        common /lang/ masi, masx, masi2, masx2
        common /stuff/  cst5, cst6, cst7, sig, cst9, eps
        common /stuff2/ cst10, cst12, cst13, cst14
        common /last/  r, xseed

c          This open statement was used on the Cray X-MP when
c          it was running CTSS.

c          call  link("unit1=(open,  xe,  text),  unit7=(da,
c          create, text), unit9=(rk, open, text), unit2=(fc,
c          open, text)//")

c          The following open statements can be used on most
```

```
c         Unix operating systems including UNICOS on the Cray
c         Y-MP at San Diego.  The file xe contains the initial
c         starting velocities of the xenon atoms, fcfkb.out
c         contains the vibrational energies of the iodine
c         molecule, rold stores the energy lost from the
c         iodine molecules for each trajectory, and rkr
c         contains the rkr potential for the iodine molecule.

        open(1, status='old', file='xe')
        open(2, status='old', file='fcfkb.out')
        open(7, status='new', file='rold')
        open(9, status='old', file='rkr')

c         The random number generator you use depends on the
c         operating system you are on and how well it works.
c         g0ccf initializes a random number sequence that
c         gives a non-repeatable sequence.  This is from the
c         NAG library.  This was used so that there would be
c         a different random number sequence for each
c         vibrational level.

        call g05ccf

c         xseed is a dummy argument used by g05caf which
c         generates the pseudo random numbers in a
c         distribution from 0 to 1.

        xseed=15344

c         In this section anything that is independent of
c         vibrational level is initialized.

        call inrkr


        do 22 i=1, 15
        read(1,3) ri,v(i)
22      continue

3       format(2e15.5)

c         The variables is and if are the vibrational levels
c         over which this calculation will be performed.

        read(1,*)is,if

c         Read in the vibrational levels for iodine.  The
c         variables rjunk etc are junk dummy variables.

        do 43 i=1,100
        read(2,*)rjunk,viben(i),rjunk1,rjunk2,rjunk3
43      continue
```

```
      b=1.0/.52290
      xe=2.66680
      sig=3.94
      masx=131.3
      masi=126.90
      masi2=masi/2.
      masx2=masx/2.
      eps=225.0000*1.196265744

c       ktm is a counter for the number of times an
c       integration step has occurred.

      ktm=0
      plnk=39.903130050

      cst5=4.*eps*(sig**6)
      cst6=sig**6
      cst7=cst5*(6.0)/masx
      cst9=masx/masi
      cst10=1./masi*1.1962657440
      cst12=2.*cst6
      cst13=2.0**(1./6.)*sig
      cst14=1.1962657440
      cst15=cst13+100.0
      rcount=0.0
      natom=15
      natom3=3*natom

c       In this section anything that is vibrational level
c       dependent but independent of phase is considered

      do 111 id=is,if

      ip=0
      stop=99
      rxyz(2)=1.332950
      rxyz(1)=-1.332950

      write(7,*)id

      w=(viben(id+1)-viben(id))/plnk*cst14
      cst16=sqrt(4.*viben(id)*cst14/masi)*.50

c       At this point the loop for taking the average over
c       phase begins. First initphl is called to find rmaxx

      call initphl
      goto 999
```

```
997     call initph
        ip=ip+1

999     do 2444 i=1,natom3,3
        r(i)=rxyz(i+1)-rxyz(i)
        r(i+1)=rxyz(i+2)-rxyz(i+1)
2444    r(i+2)=r(i+1)**(-6)

        call energy(rxyz,vxyz)
        do 28 i=1,ntm3,3
        reli(i)=rel(i)
28      sumi2i(i)=sumi2(i)

c         sumi2i is the initial i2 energy

c         Start the xenon atoms moving toward the iodine
c         molecules now that the phase has been randomized and
c         the initial energy is known.

        do 222 k= 1,15
222     vxyz(3*k)=-v(k)

c         The time step h is .0001 picoseconds

        h=.00010
        h2=h*h
        h26=h2/6.0
        hi=1.0/h

        do 100 i=1,260000

c         checks to see if the collision partner is far enough
c         away to stop the trajectory

        if(rxyz(3)-rxyz(2).gt.cst15.and.rcount.eq.1.0) goto
     $150
        if(rxyz(3)-rxyz(2).lt.cst15)rcount=1.0


c         calls the integrator remember integrator takes two
c         steps for every one call

        do 120 j=1,100
120     call integ(ktm)

100     continue

150     call energy(rxyz,vxyz)
        do 29 k=1,ntm3,3
        sum(k)=ek(k)+ep(k)
        relf(k)=rel(k)
29      sumi2f(k)=sumi2(k)
```

```
c          sumi2f is final i2 energy
c          Write the energy lost or gained after the collision.
c          The energy written out is in wavenumbers.

        do 27 k=1,ntm3,3
27      write(7,246)(relf(k)-reli(k))/cst14
246     format(3e17.7)


        if(ip.ne.stop)goto 997

111     continue

c          call exit(0)
        end

c          The  subroutine  initphl  calculates  the  maximum
c          distance between two iodine atoms for a particular
c          vibrational energy.  It then randomizes the iodine
c          vibrational phase and puts the xenon atom far enough
c          away that it is not immediately interaction with
c          the iodine molecule.

        subroutine initphl

        integer ntm3, i
        parameter(ntm3=45)

        double precision vxyz(ntm3), rxyz(ntm3), axyz(ntm3)
        double precision axyz2(ntm3)
        double precision rxyz2(ntm3)
        double precision   sig, eps
        double precision cst5
        double precision cst6, cst7, cst9, cst10
        double precision cst12
        double precision cst13, cst14, cst15, cst16
        double precision h, h2, h26, hi
        double precision masi, masx, masi2, masx2
        double precision ep(45), ek(45), sum(45), sumi2(45)
        double precision rel(45)
        double precision w, rx, rstop, rcount
        double precision r(45), rmaxx, rmax

        common /eng/ ek, ep, sumi2, rel, rmax, w, sum, rcount,
     $cst15, cst16
        common /blk2/ h, h2, h26, hi
        common /blk4/ vxyz
        common /blk5/ rxyz, rxyz2, axyz, axyz2
        common /blk9/ natom, natom1, natom3
        common /lang/ masi, masx, masi2, masx2
        common /stuff/  cst5, cst6, cst7, sig, cst9, eps
        common /stuff2/ cst10, cst12, cst13, cst14
```

```
        common /last/ r, xseed

        h=.00010
        h2=h*h
        h26=h2/6.0
        hi=1.0/h
        ktm=0

        rcount=0.
        do 10 i=1,ntm3
        axyz(i)=0.0
        axyz2(i)=0.0
        vxyz(i)=0.0
        rxyz2(i)=0.0
        rxyz(i)=0.0
        sum(i)=0.00
        ek(i)=0.00
10      ep(i)=0.00

c         initializes with random phase

        xseed=g0caf(xseed)
        rx=xseed
        rstop=1.00/w*rx+1.00/w

        nstep=int(rstop/h)



        do 21 i=1,ntm3,3
        vxyz(i+1)=cst16
        vxyz(i)=-cst16
        rxyz(i+1)=1.332950
        rxyz(i)=-1.332950
21      rxyz(i+2)=cst15

        do 140 i=1,nstep
        rmax=rxyz(2)-rxyz(1)
        if(rmax.gt.rmaxx)rmaxx=rmax
140     call integ(ktm)

        cst15=cst13+rmaxx/2.
        do 211 i=1,ntm3,3
211     rxyz(i+2)=cst15

        return
        end

c         initph does everything initph1 but calculate the
c         maximum distance between the two iodine atoms.
```

```
      subroutine initph

      integer ntm3, i
      parameter(ntm3=45)

      double precision vxyz(ntm3), rxyz(ntm3), axyz(ntm3)
      double precision axyz2(ntm3)
      double precision rxyz2(ntm3)
      double precision   sig, eps
      double precision cst5
      double precision cst6, cst7, cst9, cst10
      double precision cst12
      double precision cst13, cst14, cst15, cst16
      double precision h, h2, h26, hi
      double precision masi, masx, masi2, masx2
      double precision ep(45), ek(45), sum(45), sumi2(45)
      double precision rel(45), w, rcount
      double precision r(45),  rmax

      common /eng/ ek, ep, sumi2, rel, rmax, w, sum, rcount,
     $cst15, cst16
      common /blk2/ h, h2, h26, hi
      common /blk4/ vxyz
      common /blk5/ rxyz, rxyz2, axyz, axyz2
      common /blk9/ natom, natom1, natom3
      common /lang/ masi, masx, masi2, masx2
      common /stuff/  cst5, cst6, cst7, sig, cst9, eps
      common /stuff2/ cst10, cst12, cst13, cst14
      common /last/   r, xseed

      h=.00010
      h2=h*h
      h26=h2/6.0
      hi=1.0/h
      ktm=0
      rcount=0.

      do 10 i=1,ntm3
      axyz(i)=0.0
      axyz2(i)=0.0
      vxyz(i)=0.0
      rxyz2(i)=0.0
      rxyz(i)=0.0
      sum(i)=0.00
      ek(i)=0.00
10    ep(i)=0.00


c        initializes with random phase
```

```
      xseed=g0caf(xseed)

      rx=xseed
      rstop=1.00/w*rx+1.00/w

      nstep=int(rstop/h)


      do 21 i=1,ntm3,3
      vxyz(i+1)=cst16
      vxyz(i)=-cst16
      rxyz(i+1)=1.332950
      rxyz(i)=-1.332950
21    rxyz(i+2)=cst15

      do 140 i=1,nstep
140   call integ(ktm)

      return
      end

      subroutine integ(kstep)

c       This subroutine integrates Newton's equations for
c       the particles whose positions and velocities are
c       specified by the arrays rxyz and vxyz respectively.
c       The force/mass are in the arrays axyz and axyz2 for
c       the times i and i-1.  the integration is done by
c       Beeman's method.

      parameter (ntm=15, ntm1=ntm-1, ntm3=3*ntm)

      double precision vxyz(ntm3), rxyz(ntm3), axyz(ntm3)
      double precision axyz2(ntm3)
      double precision rxyz2(ntm3)
      double precision   eps, sig
      double precision cst5
      double precision cst6, cst7, cst9, cst10
      double precision cst12
      double precision cst13, cst14
      double precision h, h2, h26, hi
      double precision r(45)

      common /blk4/ vxyz
      common /blk5/ rxyz, rxyz2, axyz, axyz2
      common /blk2/ h, h2, h26, hi
      common /blk9/ natom, natom1, natom3
      common /stuff/  cst5, cst6, cst7, sig, cst9, eps
      common /stuff2/ cst10, cst12, cst13, cst14
      common /last/  r, xseed

      kstep=kstep+2
```

```
        do 100 i=1,natom3
        rxyz2(i)=rxyz(i)+h*vxyz(i)+h26*(4.0*axyz(i)-axyz2(i))
100     axyz2(i)=0.0

        call accel(axyz2,rxyz2,kstep-2)

        do 110 i=1,natom3
        rxyz(i) = (rxyz2(i)-rxyz(i) + h26 * (2.00 * axyz2(i)
     $+ axyz(i))) * hi
        rxyz(i) = rxyz2(i) + h * rxyz(i) + h26 * (4.00 *
     $axyz2(i) - axyz(i))
110     axyz(i)=0.00

        call accel(axyz,rxyz,kstep-1)

        do 120 i=1,natom3
120     vxyz(i) = (rxyz(i) - rxyz2(i) + h26 * (2.00 * axyz(i)
     $+ axyz2(i))) * hi

        return
        end

        subroutine accel(axyz,rxyz,ktm)

c         The subroutine accel calculates the accelerations
c         for each of the particles.  The first iodine atom,
c         the one farthest away from the xenon atom only feels
c         the force due to the iodine atom next to it.  The
c         second iodine atom feels the force due to the iodine
c         atom next to it and the xenon atom.  The xenon atom
c         only feels the force due to the iodine atom closest
c         to it.  The two iodine atoms feel a force derived
c         from the rkr potential, and the xenon feels the
c         force derived from a Weeks Chandler Anderson
c         decomposition.

        parameter(natom3=45)

        double precision rxyz(natom3), axyz(natom3)
        double precision   sig, eps
        double precision cst5
        double precision cst6, cst7, cst9, cst10
        double precision cst12
        double precision cst13, cst14
        double precision masi, masx, masi2, masx2
        double precision r(45)
        double precision  rkrf

        common /lang/ masi, masx, masi2, masx2
        common /stuff/  cst5, cst6, cst7, sig, cst9, eps
        common /stuff2/ cst10, cst12, cst13, cst14
        common /last/ r, xseed
```

```
c         Calculate the interatomic distances, and inverse
c         sixth power of the distance between the iodine and
c         xenon.  This will be used in calculations of the
c         acceleration due to the WCA potential.  This
c         probably should be done after the if statement which
c         checks to see if the xenon is close enough to feel
c         the iodine atom.

        do 244 i=1,natom3,3
        r(i)=rxyz(i+1)-rxyz(i)
        r(i+1)=rxyz(i+2)-rxyz(i+1)
244     r(i+2)=r(i+1)**(-6)

c         Calculate the acceleration the first iodine atom
c         feels due to the second iodine atom.

        do 24 i=1,natom3,3
24      axyz(i)=-rkrf(r(i))*cst10

c         Both if statements are left in for the user.  The
c         first if statement checks to see if the xenon is
c         close enough to feel any acceleration due to the
c         iodine atom.  The second one is a vectorizable if
c         statement for the Cray X-MP.  At the time compiler
c         development had not reached the stage such that an
c         if statement would vectorize.  The call to cvmgt is
c         basically a vectorizable if statement.

        do 19 i=1,natom3,3
19      if(r(i+1).gt.cst13)r(i+1)=cst13
c19      r(i+1)=cvmgt(cst13,r(i+1),r(i+1).gt.cst13)

        do 20 i=1,natom3,3
        axyz(i+2)=cst7*(cst12*r(i+1)**(-13)-r(i+1)**(-7))
20      axyz(i+1)=-axyz(i)-axyz(i+2)*cst9

        return
        end


        subroutine energy(rxyz,vxyz)

c         This subroutine calculates the energy of the total
c         system, for checking energy conservation, and then
c         calculates the energy in the iodine molecule.  Note
c         rel, the energy in the iodine molecule does not the
c         contain center of mass energy for the iodine
c         molecule.  sumi2 does contain center of mass motion.

        parameter(ntm3=45)

        double precision vxyz(ntm3), rxyz(ntm3)
```

```
      double precision   sig, eps
      double precision cst5
      double precision cst6, cst7, cst9, cst10
      double precision cst12
      double precision cst13, cst14, cst15, cst16
      double precision masi, masx, masi2, masx2
      double precision ep(45), ek(45), sumi22(45)
      double precision sumi2(45), rel(45)
      double precision r(45), rkrv, w, rcount
      double precision sum(45), rmax

      common /lang/ masi, masx, masi2, masx2
      common /eng/ ek, ep, sumi2, rel, rmax, w, sum, rcount,
     $cst15, cst16
      common /stuff/  cst5, cst6, cst7, sig, cst9, eps
      common /stuff2/ cst10, cst12, cst13, cst14
      common /last/ r, xseed

c       ek = kinetic energy ep = potential energy

      do 25 i=1,ntm3,3
      r(i+1)=rxyz(i+2)-rxyz(i+1)
      r(i)=rxyz(i+1)-rxyz(i)
      r(i+2)=r(i+1)**(-6)
      ek(i)=0.00
      rel(i)=0.00
      ek(i)=ek(i)+vxyz(i)**2*masi2
      ek(i)=ek(i)+vxyz(i+1)**2*masi2
      sumi2(i)=ek(i)
      rel(i)=(vxyz(i+1)-vxyz(i))**2*masi2/2.00
25    ek(i)=ek(i)+vxyz(i+2)**2*masx2

      do 26 i=1,ntm3,3
      ep(i)=0.00
      if(r(i+1).gt.cst13)goto 26
      ep(i)=ep(i)+cst5*((r(i+2)**2)*cst6-r(i+2))+eps
26    continue

      do 27 i=1,ntm3,3

c       Note rkrv returns energy in wavenumbers, and must be
c       converted to the units used in this program.

      sumi22(i)=rkrv(r(i))*1.1962657440
      rel(i)=rel(i)+sumi22(i)
      ep(i)=ep(i)+sumi22(i)
27    sumi2(i)=sumi2(i)+sumi22(i)

      return
      end

      subroutine inrkr
```

```
c         This subroutine reads in the RKR data and puts it in
c         form that can be used by the spline subroutine.

        double precision e(200), ri(200), ro(200)
        double precision b(200), c1(200), d1(200)

        common /kb1/ e, ri, ro, b, c1, d1, n

        n=0
5       n=n+1
        read(9,*) e(n),ri(n),ro(n)
        if (e(n).lt.12000.) goto 5
        m=n
        n=n+1
        e(n)=0.0
        ri(n)=2.66570
        do 6 i=1,m
        n=n+1
        e(n)=e(i)
6       ri(n)=ro(i)

        do 7 i=1,m/2
        t1=ri(i)
        t2=e(i)
        j=m+1-i
        ri(i)=ri(j)
        e(i)=e(j)
        ri(j)=t1
7       e(j)=t2

        call spline(n,ri,e,b,c1,d1)

        return
        end


        double precision function rkrf(r)

c         This calculates the force on an iodine atom due to
c         the other iodine atom.  The force is either a one
c         over radius to the thirteenth if r less than or
c         equal to 2.3138, one over radius to the 9.4 if r is
c         greater or equal to 4.4060 and finally if the radius
c         is between these two values a spline of the force
c         table generated from the rkr table is used.  Note
c         that in the molecular dynamics calculations a look
c         up table was used for the force and potential for
c         the iodine molecule.  A spline had to be used here
c         because energy conservation constraints are tighter.


        double precision e(200), ri(200), ro(200), b(200)
```

```
      double precision c1(200), d1(200)
      double precision r, f, dseval

      common /kb1/ e, ri, ro, b, c1, d1, n

      f=0.0
      if (r.le.2.31380) f=-12.*2.9255766d8/r**13
      if (r.ge.4.4060) f=8.4*1.28477d8/r**9.4
      if (f.eq.0.) f=dseval(n,r,ri,e,b,c1,d1)
      rkrf=-f

      return
      end


      double precision function rkrv(r)

c       This calculates the potential energy of the iodine
c       atoms due to the other iodine atom.  Again a spline
c       and not a look up table were used due to the fact
c       that energy conservation had to be much better for
c       this calculation than the molecular dynamics in a
c       liquid calculation.

      double precision e(200), ri(200), ro(200), b(200)
      double precision c1(200), d1(200)
      double precision r, v, seval

      common /kb1/ e, ri, ro, b, c1, d1, n

      v=0.
      if (r.le.2.31380) v = 2.9255766d8 / r**12 -3456.4670
     $-12540.260
      if (r.ge.4.4060) v=-1.28477d8/r**8.4
      if (v.eq.0.) v=seval(n,r,ri,e,b,c1,d1)-12540.260
      rkrv=v

      return

      end

      subroutine spline(n,x,y,b,c,d)

c       This subroutine is a cubic interpolating spline
c       taken from Computer Methods for Mathematical
c       Computations, by Forsythe, et al., p. 76.
c
c
c       s(x) = y(I) + b(I) * (x - x(I)) + c(I) * (x -
c       x(I)**2 + d(I) * (x - x(I))**3
c
c       for x(I) less than or equal to x and x is less than
```

```
c         or equal to x(I+1)
c
c         n = the number of data points (n.ge.2)
c         x= the abscissa in strictly increasing order
c         y = the ordinate

        integer n
        integer nm1, ib, i

        double precision x(n),y(n),b(n),c(n),d(n)
        double precision t

        nm1=n-1
        if (n.lt.2) return
        if (n.lt.3) goto 50

        d(1)=x(2)-x(1)
        c(2)=(y(2)-y(1))/d(1)

        do 10 i=2,nm1
        d(i)=x(i+1)-x(i)
        b(i)=2.*(d(i-1)+d(i))
        c(i+1)=(y(i+1)-y(i))/d(i)
        c(i)=c(i+1)-c(i)
10      continue

        b(1)=-d(1)
        b(n)=-d(n-1)
        c(1)=0.
        c(n)=0.
        if (n.eq.3) goto 15
        c(1)=c(3)/(x(4)-x(2))-c(2)/(x(3)-x(1))
        c(n)=c(n-1)/(x(n)-x(n-2))-c(n-2)/(x(n-1)-x(n-3))
        c(1)=c(1)*d(1)**2/(x(4)-x(1))
        c(n)=-c(n)*d(n-1)**2/(x(n)-x(n-3))

15      do 20 i=2,n
        t=d(i-1)/b(i-1)
        b(i)=b(i)-t*d(i-1)
        c(i)=c(i)-t*c(i-1)
20      continue

        c(n)=c(n)/b(n)

        do 30 ib=1,nm1
        i=n-ib
        c(i)=(c(i)-d(i)*c(i+1))/b(i)
30      continue

        b(n)=(y(n)-y(nm1))/d(nm1)+d(nm1)*(c(nm1)+2.*c(n))

        do 40 i=1,nm1
```

```
      b(i)=(y(i+1)-y(i))/d(i)-d(i)*(c(i+1)+2.*c(i))
      d(i)=(c(i+1)-c(i))/d(i)
      c(i)=3.*c(i)
40    continue

      c(n)=3.*c(n)
      d(n)=d(n-1)
      return

50    b(1)=(y(2)-y(1))/(x(2)-x(1))
      c(1)=0.
      d(1)=0.
      b(2)=b(1)
      c(2)=0.
      d(2)=0.
      return

      end


      double precision function seval(n,u,x,y,b,c,d)

c       This subroutine evaluates the spline function once
c       the coefficients have been calculated by spline.
c       cubic interpolating spline taken from Computer
c       Methods for Mathematical Computations, by Forsythe,
c       et al., p. 76.  This subroutine calculates the
c       spline interpolation for the potential.

      integer n
      integer i, j, k

      double precision u, x(n), y(n), b(n), c(n), d(n)
      double precision dx

      data i/1/

      if (i.ge.n) i=1
      if (u.lt.x(i)) goto 10
      if (u.le.x(i+1)) goto 30

10    i=1
      j=n+1

20    k=(i+j)/2
      if (u.lt.x(k)) j=k
      if (u.ge.x(k)) i=k
      if (j.gt.i+1) goto 20

30    dx=u-x(i)
      seval=y(i)+dx*(b(i)+dx*(c(i)+dx*d(i)))
      return
```

```
      end

      double precision function dseval(n,u,x,y,b,c,d)

c       This subroutine evaluates the spline function once
c       the coefficients have been calculated by spline.
c       cubic interpolating spline taken from Computer
c       Methods for Mathematical Computations, by Forsythe,
c       et al., p. 76.  This subroutine calculates the
c       spline interpolation for the force.  Note that the
c       return is just the derivative of the cubic
c       interpolation.

      integer n
      integer i, j, k

      double precision u, x(n), y(n), b(n), c(n), d(n)
      double precision dx

      data i/1/

      if (i.ge.n) i=1
      if (u.lt.x(i)) goto 10
      if (u.le.x(i+1)) goto 30

10     i=1

      j=n+1
20    k=(i+j)/2

      if (u.lt.x(k)) j=k
      if (u.ge.x(k)) i=k
      if (j.gt.i+1) goto 20

30    dx=u-x(i)
      dseval=b(i)+dx*(2.*c(i)+dx*3.*d(i))
      return

      end
```

B. PROGRAM LISTING IBCENE

```
c         Program "IBCENE"
c         Daniel Russell
c         Aug. 6, 1990

c         This program reads in the energy losses for the 1500
c         trajectories run the program "I2IBC" and calculates
c         the average energy loss as a function of number of
c         collisions.
c
c         The matrix dis contains the 1500 energy losses for
c         the 82 vibrational levels.  vel is vector that will
c         store a velocity distribution for the energy loss.
c         newe keeps track of the energy. The array tot keeps
c         track of the energy as a function of collisions.

      real dis(1500,82),vel(15),viben(82),newe,tot(10000)
      common /data/ dis

c         Because the data files are quite large they are
c         stored on a TK-50 tape.  The tar command retrieves
c         the data file from tape.  The file x1-20 contains
c         the data for the vibrational level 1 through 20.

      if(system('tar -x x1-20').ne.0)write(6,*)"tar err"

c         The subroutine dread() reads in the data.

      call dread(1,1,20)

c         The data file is removed so space is available for
c         the next file.

      if(system('rm x1-20').ne.0)write(6,*)"rm err"


      if(system('tar -x x21-40').ne.0)write(6,*)"tar err"

      call dread(2,21,40)

      if(system('rm x21-40').ne.0)write(6,*)"rm err"


      if(system('tar -x x41-60').ne.0)write(6,*)"tar err"

      call dread(3,41,60)

      if(system('rm x41-60').ne.0)write(6,*)"rm err"


      if(system('tar -x x61-82 ').ne.0)write(6,*)"tar err"
```

```
      call dread(4,61,82)

      if(system('rm x61-82').ne.0)write(6,*)"rm err"



      open(4,status='new',file='ene.dep')
      open(5,status='new',file='vel.dep')
      open(3,status='old',file='fcfkb.out')

c          Read in the vibrational energy levels.

      do 1 i=1,100
      read(3,*)rjunk,viben(i),rjunk1,rjunk2,rjunk3
1     continue

      iy=165335
      times=100

      do 12 i=1,10000
12    tot(i)=0.

      do 2 i=1,times

5     rnd=urand(iy)

c          Get a random number uniformly distributed between
c          0 and 1.

      k=int(1500.*rnd)+1

c          Change it to a random number between 1 and 1500
c          which then gives an energy loss for this step

      newe=viben(82)+dis(k,82)
      jj=0
      kk=0

6     do 3 j=82,2,-1

3     if(newe.gt.viben(j))goto 4
      goto 2

4     if(j.eq.82)newe=newe+dis(int(1500.*urand(iy))+1,82)
      jj=jj+1

c          Store the energy every 10 collisions

      if(jj.eq.10)kk=kk+1
      if(jj.eq.10)tot(kk)=newe+tot(kk)
```

```
      if(jj.eq.10)jj=0
      if(j.eq.82)goto 6

c          For energies in between vibrational levels choose an
c          energy loss which is a function of the two energy
c          levels.

      ri=(viben(j+1)-newe)/(viben(j+1)-viben(j))
      rf=1.-ri
      ry1=urand(iy)
      ry2=urand(iy)
      newe = newe + ri * dis( int(1500. * ry1 ) + 1, j ) + rf
     $* dis( int( 1500. * ry2 ) + 1, j + 1)
      goto 6
2     continue

      do 8 i=1,82
      jj=0

      do 9 k=1,15
9     vel(k)=0.

c          Calculate the energy loss as a function of the
c          velocities.

      do 7 j=1,1500
      jj=jj+1
      vel(jj)=vel(jj)+dis(j,i)
7     if(jj.eq.15)jj=0

      do 8 k=1,15
8     write(5,*)vel(k)

      do 10 i=1,10000
      if(tot(i).eq.0.)goto 11
10    write(4,*)tot(i)
11    continue
      end

      subroutine dread(i,j,k)
      real dis(1500,82)
      character*20 a
      common /data/ dis
      open(1,status="old",file="list")
      do 1 ii=1,i
1     read(1,*)a
      open(2,status="old",file=a)
      do 2 ii=1,k-j+1
      read(2,*)rj
      do 2 jj=1,1500
2     read(2,*)dis(jj,j+ii-1)
      close(1)
```

```
close(2)
return
end
```

C. PROGRAM LISTING IIBC

```
c         Program "IIBC"
c         Daniel Russell
c         August 6 1990
c
c         This program calculates the average properties of an
c         iodine atom in a bath of 107 xenon atoms.  The
c         system is first carefully thermalized.  Then the
c         calculation is then run long enough to get good
c         statistics on the radial distribution function and
c         the various force autocorrelation functions.  Some
c         of the subroutines are taken from Keenan Brown's
c         molecular dynamics program "I2XENON" and slightly
c         modified for this system.

      program xenon

      parameter (natom=108, natom1=natom-1, natom3=3*natom)
      parameter (ignum=1000, ivacf=256)
      parameter(nb=natom, nc=2*natom)

c         natom is the number of atoms in the system.  ignum
c         times three is the length of the array that will
c         store the radial distribution function.  ivacf is
c         the length of the array that will store the force
c         autocorrelation functions.  Note that there are
c         three dimensions,  so that the size of the array
c         will actually be three times ivacf.

c         If run on the Cray the following line should making
c         the real variables double precision should be
c         commented out. The Cray has enough precision to run
c         accurately enough with single precision. If this is
c         being run on the Digital Microvax or the Silicon
c         Graphics 3130 the reals should be double precision

c     implicit double precision (a-h, o-z)

c         The matrix acceli temporarily stores the individual
c         forces between the iodine atom and each xenon atom.
c         This is stored for all three dimensions even though
c         the iodine atom is spherically symmetric.  The
c         matrix dfibc actually stores the Isolated Binary
c         force autocorrelation that is described earlier in
c         the text.  The array g contains the radial
c         distribution function. The matrix atot stores the
c         accelerations used in calculating the total force
c         autocorrelation.  The matrix dftot contains the
c         total force autocorrelation.

      dimension acceli(natom3, ivacf), dfibc(ivacf, 3)
```

```
      dimension g(3*ignum), atot(ivacf, 3),  dftot(ivacf, 3)

c          alxyz is the length of the cube used in periodic
c          boundary conditions.  smass is the mass of the
c          solvent, xenon, in atomic mass units.  r2max is the
c          square of the maximum radius where the potential is
c          felt between xenon atoms.  vzero is the constant
c          that is added to the potential in order to have the
c          potential between xenon atoms go to zero at the
c          radius squared of r2max.  This make sure that the
c          potential and derivative of the potential are
c          continuous.  r2maxi and vzeroi are the corresponding
c          values for iodine.

      common /blk1/ alxyz, c1, c2, smass, c1d, c2d, r2max

      common /blk11/ vzero, vzeroi, r2maxi

c          h is the time step, h2 is h squared, h26 is h2/6 and
c          hi is one over h

      common /blk2/ h, h2, h26, hi

c          simass is the iodine mass (amu).  eps is the
c          Lennard-Jones well depth for iodine-xenon.  The
c          actual potential is a Weeks Chandler Anderson
c          decomposition of the Lennard-Jones potential.

      common /blk3/ c1di, c2di, c1i, c2i, simass, eps
      common /cacl2/ acceli, dfibc, dftot, iq, atot, g, scale

c          vxyz, rxyz, and axyz are the velocity, position.
c          They are stored all the x coordinates for the 108
c          atoms first, all the y coordinates next etc.

      common /blk5/ rxyz(natom3), vxyz(natom3), axyz(natom3)

      common /blk51/ axyz2(natom3)

c          This text was saved as an example of how to open a
c          file on the Cray X-MP using the CTSS operating
c          system.
c
c       call link("unit6=(eng, create, text), unit8=(vacf,
c    $create,text), unit1=(start, open, text), unit2=(flow,
c    $CREATE, TEXT),PRINT2//")

      open(1, file='final2', status='new')
      open(2, file='axyz', form='unformatted', status='new')
      open(1, file='infinal', status='old')
      open(8, file='fforce', status='new')
      open(2, file='jforce', status='new')
```

```
c         The  subroutine init initializes all the variables
c         and reads in a FCC lattice which is the starting
c         point for the iodine atom and the xenon solvent.

      call init

      scale=float(4*ignum)/alxyz/alxyz

c         zero arrays for start-up

      do 40 i=1,natom3
      axyz2(i)=0.0
40    axyz(i)=0.0

      do 43 i=1,3
      do 43 i1=1,ivacf
      dftot(i1,i)=0.
43    dfibc(i1,i)=0.

      do 45 i=1,3*ignum
45    g(i)=0.

c         tke is the kinetic energy of the system given the
c         temperature of 280 Kelvin.

      tke=1.50*natom*.8310*280.0

c         ktm is incremented by two on every integration step.
c         the following integration loop is for equilibration,
c         only energy data is stored.
c
      ktm=0
      dtemp=0.

c         The subroutine tempe calculates the kinetic energy
c         of the system and also removes any center of mass
c         motion.

         call tempe(ek,ep,etot,cek)

         do 999 i=1,25

c         The subroutine tempa() is very similar to tempe()
c         except it does not remove center of mass motion.  By
c         this time all the center of mass motion should be
c         removed from the system.

         call tempa(ek,ep,etot,cek)
         ff2=sqrt(tke/ek)

c         ff2 is the scale factor to cool the system to the
```

```
c         appropriate temperature.

         do 762 iw=1,natom3
762      vxyz(iw)=vxyz(iw)*ff2

999       call integ(ktm)

      dtemp=0.

c         In this do loop the system is run somewhat hot to
c         randomize it.  Therefore cooling is also done
c         slower.

      do 482 i1=1,256

         call integ(ktm)

         call tempa(ek,ep,etot,cek)
         dtemp=dtemp+ek

482      continue
         ff2=sqrt(tke/dtemp*256.)

         do 763 iw=1,natom3
763      vxyz(iw)=vxyz(iw)*ff2

        write(6,*)dtemp/1.5/.831/natom/256.

      dtemp=0.

      do 764 i2=1,3

      do 483 i1=1,2500
         call integ(ktm)
         call tempa(ek,ep,etot,cek)
         dtemp=dtemp+ek
483     continue

        ff2=sqrt(tke/dtemp*2500.)
        write(6,*)dtemp/1.5/.831/natom/2500.
        dtemp=0.

         do 764 iw=1,natom3
764      vxyz(iw)=vxyz(iw)*ff2

c         The following loop is the final cooling loop to get
c         the system as close as possible to the appropriate
c         temperature.

      dtemp=0.
      do 484 i1=1,5000
```

```
          call integ(ktm)


          call tempa(ek,ep,etot,cek)
          dtemp=dtemp+ek

484      continue

          ff2=sqrt(tke/dtemp*5000.)
         write(6,*)dtemp/1.5/.831/natom/5000.

          do 765 iw=1,natom3
765       vxyz(iw)=vxyz(iw)*ff2

       dtemp=0.
         iq=1

c           Run a do loop of 256 iterations in order to
c           initialize the matrix dfibc and acceli for
c           calculation of force autocorrelations.

       do 485 i1=1,256

c           The subroutine dinteg performs two integration loops
c           per call.  On the second loop it stores information
c           for calculating the force autocorrelations.

          call dinteg(ktm)

c           The variable iq keeps track of time 0 in the force
c           autocorrelation matrices.

          iq=iq+1
          call tempa(ek,ep,etot,cek)
485       dtemp=dtemp+ek

           iq=1

c           The subroutine dfrc actually calculates the force
c           autocorrelations.

         call dfrc
         call energy(ek,ep,etot)
         write(6,*)ek,ep,etot

         do 21 i2=1,50000

c           The subroutine tempa() is called at this point not
c           to equilibrate the system but in order to find out
c           what the average temperature of the system was
c           during the calculation.
```

```
         call tempa(ek,ep,etot,cek)
         dtemp=dtemp+ek
         call dinteg(ktm)

c         Note that iq is updated here.

         iq=1+mod(iq,ivacf)
21       call dfrc

        write(6,*)dtemp/1.5/.831/natom/50256.
        call energy(ek,ep,etot)

c         Write out the final temperature.

        write(6,*)ek,ep,etot

10    format(e10.4,6f10.4,i5)

c         Write out the Isolated Binary Collision Force
c         autocorrelation function.

      do 64 i=1,3
      do 64 i1=1,ivacf
64    write(8,*)dfibc(i1,i)


c         Write out the total force autocorrelation function.

      do 65 i=1,3
      do 65 i1=1,ivacf
65    write(8,*)dftot(i1,i)

c         Write out the radial distribution function

      do 55 i=1,3*ignum
55    write(8,*)g(i)

      stop
      call exit
      end

      subroutine tempe(ek,ep,etot,cek)

      parameter(natom=108, natom1=natom-1, natom3=3*natom)
      parameter(nb1=1, nb2=natom+1, nb3=2*natom+1)
      parameter(nb=natom, nc=2*natom)

c     implicit double precision (a-h, o-z)

      common /blk1/ alxyz, c1, c2, smass, c1d, c2d, r2max

      common /blk11/ vzero, vzeroi, r2maxi
```

```
      common /blk3/ c1di, c2di, c1i, c2i, simass, eps
      common /blk5/ rxyz(natom3), vxyz(natom3), axyz(natom3)

      common /blk51/ axyz2(natom3)

      dimension rx(natom), ry(natom), rz(natom)

      equivalence (rxyz(nb1), rx), (rxyz(nb2), ry),
     $(rxyz(nb3), rz)

      ek = 0.0
      cmassx=0.0
      cmassy=0.0
      cmassz=0.0

c         Calculate the center of mass velocity for the xenon
c         atoms.

      do 11 i=2,natom
      cmassy=cmassy+vxyz(i+nb)
      cmassz=cmassz+vxyz(i+nc)
11    cmassx=cmassx+vxyz(i)

c         Calculate the center of mass momentum for the xenon
c         atoms.

      cmassx=cmassx*smass
      cmassy=cmassy*smass
      cmassz=cmassz*smass

c         Add center of mass momentum for the iodine atom.

      cmassx=cmassx+vxyz(1)*simass
      cmassy=cmassy+vxyz(1+nb)*simass
      cmassz=cmassz+vxyz(1+nc)*simass

      tempx=cmassx/(natom1*smass+simass)
      tempy=cmassy/(natom1*smass+simass)
      tempz=cmassz/(natom1*smass+simass)

      cek=.50*(cmassx**2 + cmassy**2 + cmassz**2) / (natom1 *

     $smass + simass)

c         Remove Center of mass motion.

      do 13 i=1,natom
        vxyz(i)=vxyz(i)-tempx
        vxyz(i+nb)=vxyz(i+nb)-tempy
13      vxyz(i+nc)=vxyz(i+nc)-tempz

c         Calculate   kinetic   energy,   used   in   assigning
```

```
c          temperature.

      do 10 i=2,natom
      ek=ek+vxyz(i+nb)*vxyz(i+nb)
      ek=ek+vxyz(i+nc)*vxyz(i+nc)
10    ek=ek+vxyz(i)*vxyz(i)

      ek = .50 * smass * ek
      ek=ek+vxyz(1)*vxyz(1)*.50*simass
      ek=ek+vxyz(1+nb)*vxyz(1+nb)*.50*simass
      ek=ek+vxyz(1+nc)*vxyz(1+nc)*.50*simass

        return
        end


      subroutine tempa(ek,ep,etot,cek)

c          This subroutine is the same as tempe(), except there
c          is no calculation or subtraction of center of mass
c          motion.

      parameter(natom=108, natom1=natom-1, natom3=3*natom)
      parameter(nb1=1, nb2=natom+1, nb3=2*natom+1)
      parameter(nb=natom, nc=2*natom)

c     implicit double precision (a-h, o-z)

      common /blk1/ alxyz, c1, c2, smass, c1d, c2d, r2max

      common /blk11/ vzero, vzeroi, r2maxi
      common /blk3/ c1di, c2di, c1i, c2i, simass, eps
      common /blk5/ rxyz(natom3), vxyz(natom3), axyz(natom3)

      common /blk51/ axyz2(natom3)

      dimension rx(natom), ry(natom), rz(natom)

      equivalence (rxyz(nb1), rx), (rxyz(nb2), ry),
     $(rxyz(nb3), rz)

       ek = 0.0

      do 10 i=2,natom
      ek=ek+vxyz(i+nb)*vxyz(i+nb)
      ek=ek+vxyz(i+nc)*vxyz(i+nc)
10    ek=ek+vxyz(i)*vxyz(i)

      ek = .50 * smass * ek
      ek=ek+vxyz(1)*vxyz(1)*.50*simass
      ek=ek+vxyz(1+nb)*vxyz(1+nb)*.50*simass
```

```
      ek=ek+vxyz(1+nc)*vxyz(1+nc)*.50*simass

        return
        end



      subroutine integ(kstep)

c         This subroutine integrates Newton's equations for
c         the particles whose positions and velocities are
c         specified by the arrays rxyz and vxyz respectively.
c         The forces/mass are in the arrays axyz and axyz2 for
c         the times i and i-1.  The integration is done by
c         Beeman's  method.    This  is  identical  to  the
c         subroutine   used   in   the   molecular   dynamics
c         simulations of J. K. Brown.

      parameter(natom=108, natom1=natom-1, natom3=3*natom)

c     implicit double precision (a-h, o-z)

      common /blk2/ h, h2, h26, hi
      common /blk5/ rxyz(natom3), vxyz(natom3), axyz(natom3)

      common /blk51/ axyz2(natom3)

      dimension rxyz2(natom3)

      kstep=kstep+2

      do 100 i=1,natom3
      rxyz2(i)=rxyz(i)+h*vxyz(i)+h26*(4.*axyz(i)-axyz2(i))
100   axyz2(i)=0.0

      call accel(axyz2,rxyz2)

      do 110 i=1,natom3
      rxyz(i)=(rxyz2(i)-rxyz(i)+h26*(2.*axyz2(i)+axyz(i)))*hi
      rxyz(i)=rxyz2(i)+h*rxyz(i)+h26*(4.*axyz2(i)-axyz(i))
110   axyz(i)=0.0

      call accel(axyz,rxyz)

      do 120 i=1,natom3
120  vxyz(i)=(rxyz(i)-rxyz2(i)+h26*(2.*axyz(i)+axyz2(i)))*hi

      return
```

```
      end

      subroutine dinteg(kstep)

c          This subroutine is identical to the subroutine
c          integ(), except that in stead of calling the
c          subroutine accel() twice accel() is called once and
c          daccel() is called once.


      parameter(natom=108, natom1=natom-1, natom3=3*natom)

c     implicit double precision (a-h, o-z)

      common /blk2/ h, h2, h26, hi
      common /blk5/ rxyz(natom3), vxyz(natom3), axyz(natom3)

      common /blk51/ axyz2(natom3)

      dimension rxyz2(natom3)

      kstep=kstep+2

      do 100 i=1,natom3
      rxyz2(i)=rxyz(i)+h*vxyz(i)+h26*(4.*axyz(i)-axyz2(i))
100   axyz2(i)=0.0

      call accel(axyz2,rxyz2)

      do 110 i=1,natom3
      rxyz(i)=(rxyz2(i)-rxyz(i)+h26*(2.*axyz2(i)+axyz(i)))*hi
      rxyz(i)=rxyz2(i)+h*rxyz(i)+h26*(4.*axyz2(i)-axyz(i))
110   axyz(i)=0.0

      call daccel(axyz,rxyz)

      do 120 i=1,natom3
120  vxyz(i)=(rxyz(i)-rxyz2(i)+h26*(2.*axyz(i)+axyz2(i)))*hi

      return
      end


      subroutine accel(a,r)


      parameter(natom=108, natom1=natom-1, natom3=3*natom)
      parameter(nb=natom, nc=2*natom)
      parameter (ignum=1000, ivacf=256)

c     implicit double precision (a-h, o-z)
```

```
c
c         The subroutine accel() calculates the x, y, and z
c         components  of  the  acceleration  between  atoms.
c         Currently a Weeks Chandler Anderson force function
c         is assumed.  Although this code has been used for a
c         Lennard-Jones fluid and where changes are needed to
c         do this will be indicated in the comments.


      common /blk1/ alxyz, c1, c2, smass, c1d, c2d, r2max
      common /blk11/ vzero, vzeroi, r2maxi
      common /blk3/ c1di, c2di, c1i, c2i, simass, eps

      dimension dvr(natom), dx(natom), dy(natom), dz(natom)
      dimension a(natom3), r(natom3), r2(natom)
      dimension num(natom)

      i=1
      i1 = i+1

c         In this loop the relative distances are calculated
c         and scaled in order to take account of periodic
c         boundary conditions.

      do 220 j=i1,natom
      dx(j) = r(i) - r(j)
      dy(j) = r(i+nb) - r(j+nb)
      dz(j) = r(i+nc) - r(j+nc)
      dx(j) = dx(j) - alxyz*anint(dx(j)/alxyz)
      dy(j) = dy(j) - alxyz*anint(dy(j)/alxyz)
      dz(j) = dz(j) - alxyz*anint(dz(j)/alxyz)
220   r2(j) = dx(j)**2 + dy(j)**2 + dz(j)**2

      do 440 j=i1,natom
      if (r2(j) .gt. r2maxi) goto 441
      dvr(j) = (1.0 / r2(j))**4
      dvr(j) = dvr(j) * (c1di*dvr(j)*r2(j) - c2di)
      dx(j) = dx(j) * dvr(j)
      dy(j) = dy(j) * dvr(j)
      dz(j) = dz(j) * dvr(j)
      a(i) = a(i) + dx(j)
      a(i+nb) = a(i+nb) + dy(j)
      a(i+nc) = a(i+nc) + dz(j)
      goto 440

441     dx(j)=0.0
        dy(j)=0.0
        dz(j)=0.0
440     continue

      n = 0
```

```
      do 330 j=i1,natom
      if (r2(j) .gt. r2maxi) goto 330
      n = n + 1
      num(n) = j
      dx(n) = dx(j)
      dy(n) = dy(j)
      dz(n) = dz(j)
      r2(n) = r2(j)
330     continue

c          This next loop makes sure to put the opposite force
c          on the appropriate xenon atom.

      do 550 j=1,n
      i1=num(j)
      a(i1) = a(i1) - dx(j)
      a(i1+nb) = a(i1+nb) - dy(j)
550   a(i1+nc) = a(i1+nc) - dz(j)

c          If the Calculation is to be a pseudo gas phase
c          calculation,  uncomment the goto 11 line.
c         goto 11

c          The above loops calculated the accelerations for all
c          iodine xenon pairs.  The following loops do the same
c          for all xenon xenon pairs.

      do 10 i=2,natom1
      i1 = i+1

      do 20 j=i1,natom
      dx(j) = r(i) - r(j)
      dy(j) = r(i+nb) - r(j+nb)
      dz(j) = r(i+nc) - r(j+nc)
      dx(j) = dx(j) - alxyz*anint(dx(j)/alxyz)
      dy(j) = dy(j) - alxyz*anint(dy(j)/alxyz)
      dz(j) = dz(j) - alxyz*anint(dz(j)/alxyz)
20    r2(j) = dx(j)**2 + dy(j)**2 + dz(j)**2

      n = 0

      do 30 j=i1,natom

      if (r2(j) .gt. r2max) goto 30
      n = n + 1
      num(n) = j
      dx(n) = dx(j)
      dy(n) = dy(j)
      dz(n) = dz(j)
      r2(n) = r2(j)

30      continue
```

```
      do 40 j=1,n

      dvr(j) = (1.0 / r2(j))**4
      dvr(j) = dvr(j) * (c1d*dvr(j)*r2(j) - c2d)
      dx(j) = dx(j) * dvr(j)
      dy(j) = dy(j) * dvr(j)
      dz(j) = dz(j) * dvr(j)
      a(i) = a(i) + dx(j)
      a(i+nb) = a(i+nb) + dy(j)
40    a(i+nc) = a(i+nc) + dz(j)

      do 50 j=1,n
      i1 = num(j)
      a(i1) = a(i1) - dx(j)
      a(i1+nb) = a(i1+nb) - dy(j)
50    a(i1+nc) = a(i1+nc) - dz(j)

10      continue

c         The following loops turn the forces into
c         accelerations by dividing by the mass.

11    do 60 i=2,natom
      a(i+nb) = a(i+nb) / smass
      a(i+nc) = a(i+nc) / smass
60    a(i) = a(i) / smass

      a(1) = a(1) / simass
      a(1+nb) = a(1+nb) / simass
      a(1+nc) = a(1+nc) / simass

      return

      end

      subroutine daccel(a,r)


      parameter(natom=108, natom1=natom-1, natom3=3*natom)
      parameter(nb=natom, nc=2*natom)
      parameter (ignum=1000, ivacf=256)

c         implicit double precision (a-h, o-z)
c         The subroutine daccel() is the same as the
c         subroutine accel(), except that the data needed for
c         force autocorrelations are calculated here.  It is
c         done here because this subroutine must do some of
c         the calculations needed for the force
c         autocorrelations as it calculates accelerations.
c         This subroutine also does the radial distribution
c         calculation.
```

```
      dimension acceli(natom3, ivacf), dfibc(ivacf, 3)
      dimension dftot(ivacf, 3), g(3*ignum), atot(ivacf, 3)

      common /blk1/ alxyz, c1, c2, smass, c1d, c2d, r2max
      common /blk11/ vzero, vzeroi, r2maxi
      common /blk3/ c1di, c2di, c1i, c2i, simass, eps
      common /cacl2/ acceli, dfibc, dftot, iq, atot, g, scale

      dimension dvr(natom), dx(natom), dy(natom), dz(natom)
      dimension a(natom3), r(natom3), r2(natom)
      dimension num(natom), inc(natom)

      i=1
      i1 = i+1

      do 220 j=i1,natom
      dx(j) = r(i) - r(j)
      dy(j) = r(i+nb) - r(j+nb)
      dz(j) = r(i+nc) - r(j+nc)
      dx(j) = dx(j) - alxyz*anint(dx(j)/alxyz)
      dy(j) = dy(j) - alxyz*anint(dy(j)/alxyz)
      dz(j) = dz(j) - alxyz*anint(dz(j)/alxyz)
220   r2(j) = dx(j)**2 + dy(j)**2 + dz(j)**2

        do 1 i3=2,108
1       inc(i3)=anint(scale*r2(i3)+.5)

        do 2 i3=2,108
2       g(inc(i3))=g(inc(i3))+1.


      do 440 j=i1,natom
      if (r2(j) .gt. r2maxi) goto 441
      dvr(j) = (1.0 / r2(j))**4
      dvr(j) = dvr(j) * (c1di*dvr(j)*r2(j) - c2di)
      dx(j) = dx(j) * dvr(j)
      dy(j) = dy(j) * dvr(j)
      dz(j) = dz(j) * dvr(j)

c         Store the forces on the iodine atom due to each
c         xenon atom in the following steps.  This is used for
c         the Isolated Binary Collision force autocorrelation.

      acceli(j-1,iq)=dx(j)
      acceli(j-1+nb,iq)=dy(j)
      acceli(j-1+nc,iq)=dz(j)

c         For the total force autocorrelation only the total
c         acceleration on the iodine atom is needed.

      a(i) = a(i) + dx(j)
      a(i+nb) = a(i+nb) + dy(j)
```

```
      a(i+nc) = a(i+nc) + dz(j)

      goto 440

441   dx(j)=0.0
      dy(j)=0.0
      dz(j)=0.0
      acceli(j-1,iq)=dx(j)
      acceli(j-1+nb,iq)=dy(j)
      acceli(j-1+nc,iq)=dz(j)
440   continue

      atot(iq,1)=a(i)
      atot(iq,2)=a(i+nb)
      atot(iq,3)=a(i+nc)

      n = 0
      do 330 j=i1,natom

      if (r2(j) .gt. r2maxi) goto 330
      n = n + 1
      num(n) = j
      dx(n) = dx(j)
      dy(n) = dy(j)
      dz(n) = dz(j)
      r2(n) = r2(j)
330   continue

      do 550 j=1,n

      i1=num(j)
      a(i1) = a(i1) - dx(j)
      a(i1+nb) = a(i1+nb) - dy(j)
550   a(i1+nc) = a(i1+nc) - dz(j)


c         If a pseudo gas phase calculation is needed
c         uncomment the following line.
c         goto 11

      do 10 i=2,natom1

      i1 = i+1

      do 20 j=i1,natom

      dx(j) = r(i) - r(j)
      dy(j) = r(i+nb) - r(j+nb)
      dz(j) = r(i+nc) - r(j+nc)
      dx(j) = dx(j) - alxyz*anint(dx(j)/alxyz)
      dy(j) = dy(j) - alxyz*anint(dy(j)/alxyz)
```

```
      dz(j) = dz(j) - alxyz*anint(dz(j)/alxyz)
20    r2(j) = dx(j)**2 + dy(j)**2 + dz(j)**2

      n = 0

      do 30 j=i1,natom

      if (r2(j) .gt. r2max) goto 30
      n = n + 1
      num(n) = j
      dx(n) = dx(j)
      dy(n) = dy(j)
      dz(n) = dz(j)
      r2(n) = r2(j)
30      continue

      do 40 j=1,n
      dvr(j) = (1.0 / r2(j))**4
      dvr(j) = dvr(j) * (c1d*dvr(j)*r2(j) - c2d)
      dx(j) = dx(j) * dvr(j)
      dy(j) = dy(j) * dvr(j)
      dz(j) = dz(j) * dvr(j)
      a(i) = a(i) + dx(j)
      a(i+nb) = a(i+nb) + dy(j)
40    a(i+nc) = a(i+nc) + dz(j)

      do 50 j=1,n
      i1 = num(j)
      a(i1) = a(i1) - dx(j)
      a(i1+nb) = a(i1+nb) - dy(j)
50    a(i1+nc) = a(i1+nc) - dz(j)

10      continue

11    do 60 i=2,natom
      a(i+nb) = a(i+nb) / smass
      a(i+nc) = a(i+nc) / smass
60    a(i) = a(i) / smass

      a(1) = a(1) / simass
      a(1+nb) = a(1+nb) / simass
      a(1+nc) = a(1+nc) / simass
      return
        end


      subroutine energy(ek,ep,etot)


c         The  subroutine  calculates  kinetic  energy  (ek),
c         potential  energy  (ep)  and  the  total  energy  (etot)
c         for  the  system  of  particles  whose  velocities  and
```

```
c          positions are given by the arrays vxyz and rxyz
c          respectively.  A  Weeks  Chandler  Anderson
c          decomposition of a Lennard-Jones 6-12 potential
c          shifted to zero at rzero is assumed.

      parameter(natom=108, natom1=natom-1, natom3=3*natom)
      parameter(nb1=1, nb2=natom+1, nb3=2*natom+1)
      parameter(nb=natom, nc=2*natom)
c     implicit double precision (a-h, o-z)

      common /blk1/ alxyz, c1, c2, smass, c1d, c2d, r2max

      common /blk11/ vzero, vzeroi, r2maxi
      common /blk3/ c1di, c2di, c1i, c2i, simass, eps
      common /blk5/ rxyz(natom3), vxyz(natom3), axyz(natom3)

      common /blk51/ axyz2(natom3)

      dimension r2(natom), rx(natom), ry(natom), rz(natom)

      integer num(natom)

      equivalence (rxyz(nb1), rx), (rxyz(nb2), ry),
     $(rxyz(nb3), rz)

c
c           Calculate  the kinetic energy.
c
      ek = 0.0

      do 10 i=2,natom
      ek=ek+vxyz(i+nb)*vxyz(i+nb)
      ek=ek+vxyz(i+nc)*vxyz(i+nc)
10    ek=ek+vxyz(i)*vxyz(i)

      ek = .50 * smass * ek
      ek=ek+vxyz(1)*vxyz(1)*.50*simass
      ek=ek+vxyz(1+nb)*vxyz(1+nb)*.50*simass
      ek=ek+vxyz(1+nc)*vxyz(1+nc)*.50*simass

      ep=0.
        i=1
      i1 = i + 1

c          Calculate the potential energy of all the xenon
c          atoms interacting with the iodine atom.

      do 220 j=i1,natom
      dx = rx(i) - rx(j)
      dy = ry(i) - ry(j)
      dz = rz(i) - rz(j)
      dx = dx - alxyz*anint(dx/alxyz)
```

```
      dy = dy - alxyz*anint(dy/alxyz)
      dz = dz - alxyz*anint(dz/alxyz)
220   r2(j) = dx**2 + dy**2 + dz**2

      n = 0
      do 330 j=i1,natom
      if (r2(j) .gt. r2maxi) goto 330
      n = n + 1
      num(n) = j
      r2(n) = r2(j)
330   continue

      do 440 j=1,n
      vr = (1.0 / r2(j))**3

c         The following line should add vzeroi instead of eps
c         if a Lennard-Jones potential is to be used.

440   ep = ep + vr * (c1i*vr - c2i)+eps



c         For a pseudo gas uncomment the following line.

c        goto 11

c         Calculate the potential energy of all the xenon
c         atoms interacting with the other xenon atoms.

      do 15 i=2,natom1

      i1 = i + 1

      do 20 j=i1,natom
      dx = rx(i) - rx(j)
      dy = ry(i) - ry(j)
      dz = rz(i) - rz(j)
      dx = dx - alxyz*anint(dx/alxyz)
      dy = dy - alxyz*anint(dy/alxyz)
      dz = dz - alxyz*anint(dz/alxyz)
20    r2(j) = dx**2 + dy**2 + dz**2

      n = 0
      do 30 j=i1,natom
      if (r2(j) .gt. r2max) goto 30
      n = n + 1
      num(n) = j
      r2(n) = r2(j)
30    continue

      do 40 j=1,n
      vr = (1.0 / r2(j))**3
```

```
40    ep = ep + vr * (c1*vr - c2)

c         This line adds the amount of energy that the Weeks
c         Chandler Anderson potential was shifted up relative
c         to the Lennard-Jones potential.

      ep = ep + n*vzero

15    continue

11    etot = ek + ep

      return
      end


      subroutine init

      parameter(natom=108, natom1=natom-1, natom3=3*natom)
      parameter(nb=natom, nc=2*natom)

c     implicit double precision (a-h, o-z)

      common /blk1/ alxyz, c1, c2, smass, c1d, c2d, r2max

      common /blk11/ vzero, vzeroi, r2maxi
      common /blk2/ h, h2, h26, hi
      common /blk3/ c1di, c2di, c1i, c2i, simass, eps
      common /blk5/ rxyz(natom3), vxyz(natom3), axyz(natom3)

      common /blk51/ axyz2(natom3)



      rzero = 10.0
      elj = 4.0 * 154.0
      sigma = 4.10
      smass = 131.30

c         The following line is for the density 1.8 gm/cc.

      alxyz = (108.0 * smass / .60230 / 1.80)**(1.0/3.0)

c
c    calculate various other quantities needed for execution
c
      read(1,10) h,rzero,rmax,elj,sigma,smass,alxyz,ndatom

      h = .0050

c   Iodine mass (amu)
```

```
      simass=126.9

c   Iodine - Xenon well depth

      elji=225.*4.*1.196

c   Iodine - Xenon sigma

      sigi=3.94

      h2=h*h
      h26=h2/6.0
      hi=1.0/h

c         The following line should be used for a Lennard-
c         Jones potential, and the two lines after should be
c         commented out.
c      r2max = rzero**2

      r2maxi= (2.0 **(1./6.)*sigi)**2
      r2max= (2.0 **(1./6.)*sigma)**2
      eps= elji/4.

      elj=1.19610*elj

      c2 = sigma**6
      c1 = elj * c2 * c2
      c1d = 12.0 * c1
      c2 = c1 / c2
      c2d = 6.0 * c2

      c2i=sigi**6
      c1i=  elji *c2i *c2i
      c1di=12.0 * c1i
      c2i=c1i / c2i
      c2di = 6.0 * c2i

c         The following two lines should be used for a
c         Lennard-Jones potential, and the line after should
c         be commented out.

c     vzero=1.0/rzero**6
c     vzero = -vzero*(c1*vzero-c2)
      vzero = elj/4.

c         The following two lines should be used for a
c         Lennard-Jones potential, and the line after should
c         be commented out.
c      vzeroi=1.0/rzero**6
c      vzeroi = -vzeroi*(c1i*vzeroi-c2i)

c
```

```
c          Read in initial positions and velocities

      do 20 i=1,natom
      read(1,*) rxyz(i),rxyz(i+nb),rxyz(i+nc)
      read(1,*) vxyz(i),vxyz(i+nb),vxyz(i+nc)
20    continue
      close(1)

      return
      end


      subroutine dfrc

c          The subroutine dfrc calculates the Isolated force
c          autocorrelation  function  and  the  total  force
c          autocorrelation function.

      parameter (natom=108, natom1=natom-1, natom3=3*natom)
      parameter (ignum=1000, ivacf=256)
      parameter(nb=natom, nc=2*natom)

c     implicit double precision (a-h, o-z)

      dimension acceli(natom3, ivacf), dfibc(ivacf, 3)
      dimension dftot(ivacf, 3), g(3*ignum), atot(ivacf, 3)

      common /blk1/ alxyz, c1, c2, smass, c1d, c2d, r2max

      common /blk11/ vzero, vzeroi, r2maxi
      common /cacl2/  acceli, dfibc, dftot, iq, atot, g, scale


      do 11 i1=iq-1,1,-1

      it=ivacf-iq+i1+1
      dftot(it,1)=atot(iq,1)*atot(i1,1)+dftot(it,1)
      dftot(it,2)=atot(iq,2)*atot(i1,2)+dftot(it,2)
      dftot(it,3)=atot(iq,3)*atot(i1,3)+dftot(it,3)

      do 1 i=1,natom1

      dfibc(it,1)=acceli(i,iq)*acceli(i,i1)+dfibc(it,1)
      dfibc(it,2)=acceli(i+nb,iq)*acceli(i+nb,i1)+dfibc(it,2)
1     dfibc(it,3)=acceli(i+nc,iq)*acceli(i+nc,i1)+dfibc(it,3)

11       continue

      do 12 i1=iq,ivacf

      it=1- iq + i1
      dftot(it,1)=atot(iq,1)*atot(i1,1)+dftot(it,1)
```

```
      dftot(it,2)=atot(iq,2)*atot(i1,2)+dftot(it,2)
      dftot(it,3)=atot(iq,3)*atot(i1,3)+dftot(it,3)

      do 2 i=1,natom1

      dfibc(it,1)=acceli(i,iq)*acceli(i,i1)+dfibc(it,1)
      dfibc(it,2)=acceli(i+nb,iq)*acceli(i+nb,i1)+dfibc(it,2)
2     dfibc(it,3)=acceli(i+nc,iq)*acceli(i+nc,i1)+dfibc(it,3)

12    continue

      return
      end
```

D. PROGRAM LISTING MAIN

```
/*      Program "MAIN"
 *      Daniel Russell
 *      Aug. 6, 1990
 *
 *           The following program is a data acquisition program
 *      written specifically for a 80386 computer with 80387
 *      coprocessor.  The program takes data on a CAMAC based
 *      system using the following.  A LeCroy model 2323A Dual
 *      Gate and Delay Generator, A LeCroy model 4300 Fast
 *      Encoding and Readout Gated ADC, a LeCroy model 4301
 *      Driver Module, and finally a DSP Technologies Model 6001
 *      CAMAC crate controller and PC004 IBM-PC interface.  The
 *      DSP equipment also came with sample code that was
 *      modified to take at advantage of the 80836 and the 80837
 *      for speed.  Speed was needed because the data is
 *      collected and normalized at 8 kHz, this is at the limits
 *      of the 80386's ability.  That assembly code will not be
 *      presented here because of DSP's copywrite.  The software
 *      also communicates with a Klinger Scientific MC-4 Stepping
 *      Motor Controller Driver which controls a stepping stage.
 *      The last piece of hardware that the software can
 *      communicate with is a Stanford Research Systems Model
 *      DG535 Digital Delay / Pulse Generator that is no longer
 *      needs to communicate to the software.
 *
 *           The file Main.c contains the following subroutines.
 *
 *           main()
 *           void readfile()
 *           void setupfile()
 *           void p_scan()
 *           void s_step()
 *           void p_step()
 *           void stage_start()
 *           void gate()
 *           void read_ped()
 *           void read_comment()
 *           void display()
 *           void channel_display()
 *           void storefile()
 *           void ddg_check()
 */

#include <process.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <graph.h>
#include <ctype.h>
```

```
#include <malloc.h>
#include <dos.h>
#include <decl.h>
#include <process.h>
#include <conio.h>
#include "dan.h"
#include <setjmp.h>

/*      These variables are system dependent and are based
*    the graphics coordinate system defined in the Microsoft
*    C 5.1 graphics library.  The variables rect_xmin,
*    rect_ymin rect_xmax, and rect_ymax are dependent on what
*    type of video board, although the values in this program
*    will work on a Hercules video card, EGA, and VGA although
*    the VGA will be running in EGA mode.
*/

struct rccoord rcoord;
struct videoconfig vc;

/*   These variables are dependent on where the CAMAC A to D
*    and gate and delay generator are physically located in
*    the CAMAC crate.  AD=8 signifies that the A to D is in
*    slot 8.  GATE=16 signifies that the gate and delay
*    generator is in slot 16.
*/

int AD=8;
int GATE=16;


/*      The variables channel_dis1 and channel_dis2
*    determine which of the three data variables are
*    displayed.
*/

int channel_dis1, channel_dis2;

/*      The variables channel_color1, channel_color2, and
*    box_color define the color of the two channels displayed
*    and the color of the box.
*/

int channel_color1, channel_color2,box_color;

/*      The variables gate_chan_a and gate_chan_b are the
*    variables that store the data that is actually sent to
*    the gate and delay generator (LeCroy model 2323A).  The
*    format of the data is described on page 11 of the manual
*    for the 2323A.  Note also that the value of 0 for these
*    variables is invalid.
*/
```

```
int     gate_chan_a = 0, gate_chan_b = 0;

/*        The variables points_scan and stage_step describe
 *    the number of points the stage is going to scan(
 *    basically the time base of the experiment) and the number
 *    of stage steps between these points.  0 is an invalid
 *    value for points_scan and for stage_step since the user
 *    may wish to move 0 stage steps per data point the author
 *    of the program used his birthday to be an invalid value
 *    assuming it would be an unusual value to chose.
 */

int  points_scan = 0, stage_step = -1003;

/*        The array pedestal is an array of the three values
 *    that will be subtracted off data channel 0, and 15.  This
 *    also leaves room for one more pedestal.  This is to allow
 *    the subtraction of background current in the A to D.  See
 *    LeCroy's CAMAC model 4300B manual pg 1-5.   If
 *    pedestal[0]=-1 the data is declared invalid.  -1 was
 *    chosen to allow the user to input
 *    0 for a pedestal so that the user can find out what the
 *    pedestal value is.
 */

int     pedestal[3] ;

/*        The variables low and high determine what is the
 *    highest and lowest acceptable data. 0 is an invalid value
 *    for both low and high
 */

int     low=0,high=0;

int     multi_count = 0, reverse_flag,scan;

/*         multi_count is the number of times that the stage
 *    should be scanned, reverse_flag is 1 if data is taken in
 *    only one direction of stage movement and 2 if data is to
 *    be taken in both directions.  Taking data in both
 *    directions is more efficient and will also help cancel
 *    any long term drift in time of the concentration of the
 *    molecule that you are studying if the drift is small.
 *    scan is the actual number of scans taken.
 */

long int  shots_step = 0, stage_beg = -100362;

/*        shots_step is the number of laser shots taken for
 *    each data point. Any value less than or equal to two is
 *    invalid at this time due to need for speed in the
 *    assembly code.  stage_beg is the variable that stores
```

```
*     where the stage should begin taking data.  Note that both
*     of these variables are long integers to allow for values
*     that can be greater that 65000.  The invalid data values
*     are 0 and -100362 respectively.
*/

char file_name[7], file_num[3], data_file[12];

/*         These three character arrays store the file name,
*     for example "dan", the file number, for example "1", and
*     data_file stores the total file name, using the above
*     data as an example data_file="dan.1" .  The data is
*     stored this way to allow multiple scans to be store
*     easily in succession.
*/

char comment[1000];

/*         The array comment stores the users comment for a
*     particular data file.  It can store up to 998 characters
*     and ends with the character "~" to indicate the user has
*     finished the comment.  The "~" is not displayed by the
*     program except when the user enters it.
*/

int  delay_a, delay_b,  gwidth_a, gwidth_b;

/*         delay_a and delay_b are the delays for the gate
*     pulses that come out of the LeCroy 2323A gate and delay
*     generator for channel A and B respectively.  They are
*     stored in nanoseconds and must be less than 1000
*     nanoseconds and greater that 0 nanoseconds.  gwidth_a and
*     gwidth_b are the widths for channel A and B respectively.
*     The value of gwidth_a and gwidth_b must be 0, 1, 2, or 3.
*     See the LeCroy 2323A manual pg 11 for more information.
*/

double  *d_norm[4];

/*         This an array of pointers to arrays where the data
*     is stored.  The arrays are dynamically allocated in the
*     program.  The number of arrays is 4 due to the fact that
*     at a later point in time someone may want to also collect
*     the actual laser power as a function of time.
*/

int tcolor=7, ecolor=4;

/*         tcolor and ecolor are the color of normal text
*     strings and error text used in the program.
*/
```

```
int ddg,trig_check=1;

/*      ddg is the variable associated by the GPIB-PC
*    software with the digital delay generator.  trig_check is
*    a flag to see if user wants to check that the digital
*    delay is being triggered by the CPM.   The default
*    trig_check=0 is no checking, trig_check=1 checking is
*    enabled while the software is waiting for keystrokes.
*/

jmp_buf mark;

/*      mark is used by setjmp(mark) and longjmp(mark,-1) to
*    set up where the program should jump to if a triggering
*    error is detected.  First check() informs the user that
*    the DDG is on internal and then jumps back to the main
*    menu.
*/

int main()
{

    char    inputs[3];
    int     n = 1, is_set = 0,trig,sresult;

    char     *input;
    channel_dis1=2;
    channel_dis2=3;
    channel_color1=10;
    channel_color2=13;
    box_color=11;
    ddg=ibfind("ddg");
    comment[0]='~';
    pedestal[0] = -1;

    setjmp(mark);

/*      Program will jump here if the DDG does not get
*    triggered properly
*/
    _setvideomode(_DEFAULTMODE);
    while (n != 6 ) {
         _settextcolor(tcolor);
         _settextposition(10, 15);
         rcoord = _gettextposition();

/*      This next loop writes out the main menu.   The
*    strings output are found in the file const.c.  Although
*    reading the program is a little harder due to the fact
*    the strings are not in this file, it saves space this
*    way.
*/
```

```
        for (n = 1; n < 7; n++) {
             _outtext(main_string(n));
             rcoord.row++;
             _settextposition(rcoord.row, rcoord.col);
        }
        if (trig_check==0)
        {
             trig=0;
             sresult=system("mode spe com");
             while(kbhit()==0&&trig==0)
                  trig=check();
             ibloc(ddg);
             sresult=system("mode spe auto");
             if (trig==-1)
                  longjmp(mark,-1);
        }

        input = gets(inputs);

        n = atoi(inputs);

        if (n == 1)

/*           Read in a setup file that contains all the
 *    parameters needed for data collection
 */
             readfile();
        else if (n == 2)
             setupfile();
        else if (n == 4){

/*      The space allocated for d_norm is freed here so that
 *    other data files can be looked at that may or may not
 *    have different data lengths.
 */


             if(d_norm[0]!= NULL)
                  free (d_norm[0]);
             if(d_norm[1]!= NULL)
                  free (d_norm[1]);
             if(d_norm[2]!= NULL)
                  free (d_norm[2]);
             if(d_norm[3]!= NULL)
                  free (d_norm[3]);
             look_data();
             if (points_scan != 0 ) {

/*      If points_scan is defined the space is reallocated
 */
                  d_norm[0] = (double *)calloc(points_scan,
```

```
                                    sizeof(double));
                        d_norm[1] = (double *)calloc(points_scan,
                                    sizeof(double));
                        d_norm[2] = (double *)calloc(points_scan,
                                    sizeof(double));
                        d_norm[3] = (double *)calloc(points_scan,
                                    sizeof(double));
                    }
                }

        else if (n == 3) {
             is_set = 0;

        /*        Check for valid data parameters before allowing data
         *   to be taken
         */

             _setvideomode(_DEFAULTMODE);
             _settextposition(2, 15);
             rcoord = _gettextposition();
             _settextcolor(ecolor);
             if (gate_chan_a == 0) {
                  is_set = 1;
                  _outtext(error_string(1));
             }
             if (stage_step == -1003 && stage_beg == -1003621 &&
                 points_scan == 0) {
                  rcoord.row++;
                  _settextposition(rcoord.row, rcoord.col);
                  is_set = 1;
                  _outtext(error_string(2));
             }
             if (shots_step == 2) {
                  rcoord.row++;
                  _settextposition(rcoord.row, rcoord.col);
                  is_set = 1;
                  _outtext(error_string(3));
             }
             if (pedestal[0] == -1) {
                  rcoord.row++;
                  _settextposition(rcoord.row, rcoord.col);
                  is_set = 1;
                  _outtext(error_string(4));
             }
             if (high == 0 || low == 0 ) {
                  rcoord.row++;
                  _settextposition(rcoord.row, rcoord.col);
                  is_set = 1;
                  _outtext(error_string(6));
             }

             if ( is_set == 0) {
```

```
/*          At this point all data points are found to be valid
 *     and data collection can begin.
 */

            setupcamac();

            take_data_menu();
      }




}
else if(n==5)
ddg_check();
else if (n==6)
exit(0);
else if (n < 1 || n > 6) {
      _setvideomode(_DEFAULTMODE);
      _settextposition(9, 15);
      _settextcolor(ecolor);
      _outtext( main_string(0));
      _settextcolor(tcolor);
}

      }

      exit(-1);
}


/*               Read in a setup file that contains all the
 *     parameters needed for data collection
 */

void readfile(void)

{

      FILE * stream;
      char inputs[10], inputs2[2], test[4];
      char    *input, *input2;

      int      i, ch,trig,sresult;
      inputs2[0] = 'y';
      stream = NULL;

/*          On basically all user questions the user is asked
 *     for information.  If the information is not usable or a
 *     file is not found this section loops around until the
 *     user gives up.  That is why inputs2 is defined Y
```

```
*/

     while ((inputs2[0] == 'Y' || inputs2[0] == 'y') && stream
             == NULL) {
          _setvideomode(_DEFAULTMODE);
          _settextposition(10, 15);
          rcoord = _gettextposition();
          _settextcolor(tcolor);
          _outtext( prompt_string(1));
          if (trig_check==0)
          {
               trig=0;
               sresult=system("mode spe com");
               while(kbhit()==0&&trig==0)
                    trig=check();
               ibloc(ddg);
               sresult=system("mode spe auto");
               if (trig==-1)
                    longjmp(mark,-1);
          }


          input = gets(inputs);

/*        Attempt to open the file.
*/
          if ((stream = fopen(input, "rb")) == NULL) {
               rcoord.row++;
               _settextcolor(ecolor);
               _settextposition(rcoord.row, rcoord.col);
               _outtext(prompt_string(4));
               rcoord.row++;
               _settextposition(rcoord.row, rcoord.col);
               _outtext(prompt_string(5));
               rcoord.row++;
               _settextposition(rcoord.row, rcoord.col);
               _outtext( prompt_string(7));
               if (trig_check==0)
               {
                    trig=0;
                    sresult=system("mode spe com");
                    while(kbhit()==0&&trig==0)
                         trig=check();
                    ibloc(ddg);
                    sresult=system("mode spe auto");
                    if (trig==-1)
                         longjmp(mark,-1);
               }


               input2 = gets(inputs2);
               if (inputs2[0] != 'y' && inputs2[0] == 'Y') {
```

```
                    _setvideomode(_DEFAULTMODE);
                    return;
               }
          }
     }

/*        If the file exists read the data in
*/

     if (stream != NULL) {
          for (i = 0; i != 3; i++) {
               ch = fgetc(stream);
               test[i] =(char) ch;
          }

          test[i] = '\0';

/*        All setup files begin with djr
*/
          if (strcmp(test, "djr") != 0)
               printf("not setup file");

/*        Probably should exit this gracefully if it really is
*    not a setup file as opposed to reading blindly
*/

          fscanf(stream, "%i", &gate_chan_a);
          fscanf(stream, "%i", &gate_chan_b);
          fscanf(stream, "%i", &points_scan);
          fscanf(stream, "%i", &stage_step);
          fscanf(stream, "%i", &high);
          fscanf(stream, "%i", &low);

          fscanf(stream, "%li", &shots_step);
          fscanf(stream, "%li", &stage_beg);
          fscanf(stream, "%i", &pedestal[0]);
          fscanf(stream, "%i", &pedestal[1]);
          fscanf(stream, "%i", &pedestal[2]);
          for (i = 0; (i < 999) && (ch = fgetc(stream)) !=
               '~'; i++)
               comment[i] =(char) ch;
          comment[i] =(char) ch;
          i++;
          ch = fgetc(stream);
          comment[i] =(char) ch;
          fclose(stream);
          _setvideomode(_DEFAULTMODE);

          return;
     }
     _setvideomode(_DEFAULTMODE);
}
```

```
void setupfile(void)


{
     char     *input;
     char    inputs[3];
     char    buffer[10];
     int trig,sresult;
     int      is_set,n,i;

     _setvideomode(_DEFAULTMODE);
     n=0;

/*        This section of code gives prompts for the setup
 *   data values and loops around until 14 or quit is chosen.
 */
     while (n != 14) {
          _settextcolor(tcolor);
          _settextposition(10, 15);
          rcoord = _gettextposition();
          _settextposition(rcoord.row, rcoord.col);
          _outtext( setup_string(1));
          if (points_scan != 0) {
               rcoord.col = rcoord.col + 40;
               _settextposition(rcoord.row, rcoord.col);
               sprintf(buffer, " %i", points_scan);
               _settextcolor(ecolor);
               _outtext(buffer);
               _settextcolor(tcolor);
               rcoord.col = rcoord.col - 40;
          }
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          _outtext( setup_string(2));
          if (stage_step != -1003) {
               rcoord.col = rcoord.col + 40;
               _settextposition(rcoord.row, rcoord.col);
               sprintf(buffer, " %i", stage_step);
               _settextcolor(ecolor);
               _outtext(buffer);
               _settextcolor(tcolor);

               rcoord.col = rcoord.col - 40;
          }
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          _outtext( setup_string(3));
          if (shots_step != 0) {
               rcoord.col = rcoord.col + 40;
               _settextposition(rcoord.row, rcoord.col);
               sprintf(buffer, " %li", shots_step);
               _settextcolor(ecolor);
```

```
        _outtext(buffer);
        _settextcolor(tcolor);
        rcoord.col = rcoord.col - 40;
    }
    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext( setup_string(4));
    if (stage_beg != -100362) {
        rcoord.col = rcoord.col + 40;
        _settextposition(rcoord.row, rcoord.col);
        sprintf(buffer, " %li", stage_beg);
        _settextcolor(ecolor);
        _outtext(buffer);
        _settextcolor(tcolor);
        rcoord.col = rcoord.col - 40;
    }
    for (i=5;i<14;i++){
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext( setup_string(i));
    }
    if (high != 0 && low != 0) {
        rcoord.col = rcoord.col + 40;
        _settextposition(rcoord.row, rcoord.col);
        sprintf(buffer, "%i, %i", low,high);
        _settextcolor(ecolor);
        _outtext(buffer);
        _settextcolor(tcolor);
        rcoord.col = rcoord.col - 40;
    }

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext( setup_string(14));
    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    if (trig_check==0)
    {
        trig=0;
        sresult=system("mode spe com");
        while(kbhit()==0&&trig==0)
            trig=check();
        ibloc(ddg);
        sresult=system("mode spe auto");
        if (trig==-1)
            longjmp(mark,-1);
    }


    input = gets(inputs);
    n = atoi(inputs);
    if (n < 1 || n > 14 ) {
```

```
        _setvideomode(_DEFAULTMODE);
        _settextcolor(ecolor);
        _settextposition(9, 15);
        rcoord = _gettextposition();
        _settextposition(rcoord.row, rcoord.col);

        _outtext( main_string(0));
    } else if (n == 1) {
        p_scan(1);
        _setvideomode(_DEFAULTMODE);
    } else if (n == 2) {
        s_step(1);
        _setvideomode(_DEFAULTMODE);
    } else if (n == 3) {
        p_step(1);
        _setvideomode(_DEFAULTMODE);

    } else if (n == 4) {
        stage_start(1);
        _setvideomode(_DEFAULTMODE);
    } else if (n == 5) {
        gate(1);
        _setvideomode(_DEFAULTMODE);
    } else if (n == 6) {
        read_ped(1);
        _setvideomode(_DEFAULTMODE);
    } else if (n == 7) {
        read_comment();
        _setvideomode(_DEFAULTMODE);
    } else if (n == 8 ) {
        display();
        _setvideomode(_DEFAULTMODE);
    } else if (n == 9) {
        storefile();
        _setvideomode(_DEFAULTMODE);
    } else if (n == 10) {
        channel_display();
        _setvideomode(_DEFAULTMODE);
    } else if (n == 11) {
        is_set = 0;
        _setvideomode(_DEFAULTMODE);
        _settextposition(2, 15);
        rcoord = _gettextposition();
        _settextcolor(ecolor);
        if (gate_chan_a == 0) {
             is_set = 1;
             _outtext(error_string(1));
        }
        if  (stage_step  ==  -1003  &&  stage_beg  ==
            -1003621 && points_scan == 0) {
             rcoord.row++;
             _settextposition(rcoord.row,rcoord.col);
```

```
                    is_set = 1;
                    _outtext(error_string(2));
                }
                if (shots_step == 2) {
                    rcoord.row++;
                    _settextposition(rcoord.row,rcoord.col);
                    is_set = 1;
                    _outtext(error_string(3));
                }
                if (pedestal[0] == -1) {
                    rcoord.row++;
                    _settextposition(rcoord.row,rcoord.col);
                    is_set = 1;
                    _outtext(error_string(4));
                }
                if (is_set == 0) {
                    setupcamac();
                    take_data();
                    _setvideomode(_DEFAULTMODE);
                }
            } else if (n == 12) {
                is_set = 0;
                _setvideomode(_DEFAULTMODE);
                _settextposition(2, 15);
                rcoord = _gettextposition();
                _settextcolor(ecolor);
                if (gate_chan_a == 0) {
                    is_set = 1;
                    _outtext(error_string(1));
                }
                if  (stage_step  ==  -1003  &&  stage_beg  ==
                    -1003621 && points_scan == 0) {
                    rcoord.row++;
                    _settextposition(rcoord.row,rcoord.col);
                    is_set = 1;
                    _outtext(error_string(2));
                }
                if (shots_step == 2) {
                    rcoord.row++;
                    _settextposition(rcoord.row,rcoord.col);
                    is_set = 1;
                    _outtext(error_string(3));
                }
                if (pedestal[0] == -1) {
                    rcoord.row++;
                    _settextposition(rcoord.row,rcoord.col);
                    is_set = 1;
                    _outtext(error_string(4));
                }
                if (is_set == 0) {
                    setupcamac();
                    norm();
```

```
                    _setvideomode(_DEFAULTMODE);
                }
            } else if (n == 13) {
                bound(1);
                _setvideomode(_DEFAULTMODE);

            }
        }
        _setvideomode(_DEFAULTMODE);
        return;
}

void p_scan(n)
int     n;
{
        int trig,sresult;
        int     i;
        char    numbers[10];
        char    buffer[30];
        char    *result, *stage_string();

        if(d_norm[0]!= NULL)
            free (d_norm[0]);
        if(d_norm[1]!= NULL)
            free (d_norm[1]);
        if(d_norm[2]!= NULL)
            free (d_norm[2]);
        if(d_norm[3]!= NULL)
            free (d_norm[3]);

/*          If n== 0 and points_scan is defined allocate space
*       for the data
*/

        if (points_scan != 0 && n == 0) {
            d_norm[0]    =    (double    *)calloc(points_scan,
                              sizeof(double));
            d_norm[1]    =    (double    *)calloc(points_scan,
                              sizeof(double));
            d_norm[2]    =    (double    *)calloc(points_scan,
                              sizeof(double));
            d_norm[3]    =    (double    *)calloc(points_scan,
                              sizeof(double));
            for (i = 0; i < 4; i++) {
                if (d_norm[i] == NULL){
                    points_scan=0;

                    _setvideomode(_DEFAULTMODE);

                    _settextposition(10, 15);
                    rcoord = _gettextposition();
```

```
                    sprintf(buffer,"calloc failed on %i",i);
                    _outtext(buffer);
                    rcoord.row++;
                    _settextposition(rcoord.row,rcoord.col);

                    _outtext("hit a key to continue");
                    while (i == 0)
                         i = kbhit();
                    i = getche();


               }
          }
          return;

     }

     _setvideomode(_DEFAULTMODE);
     _settextcolor(tcolor);
     _settextposition(10, 15);
     rcoord = _gettextposition();
     _outtext(stage_string(1));
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     if (trig_check==0)
     {
          trig=0;
          sresult=system("mode spe com");
          while(kbhit()==0&&trig==0)
               trig=check();
          ibloc(ddg);
          sresult=system("mode spe auto");
          if (trig==-1)
               longjmp(mark,-1);
     }


     result = gets(numbers);
     points_scan = atoi(numbers);

/*        Prompt for the # of points per scan and only accept
 *   positive values
 */

     while (points_scan == 0 || points_scan < 0) {
          _setvideomode(_DEFAULTMODE);
          _settextcolor(ecolor);
          _settextposition(10, 15);
          rcoord = _gettextposition();
          _outtext(error_string(5));
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
```

```
                _settextcolor(tcolor);
                _outtext(stage_string(1));
                rcoord.row++;
                _settextposition(rcoord.row, rcoord.col);
                if (trig_check==0)
                {
                     trig=0;
                     sresult=system("mode spe com");
                     while(kbhit()==0&&trig==0)
                          trig=check();
                     ibloc(ddg);
                     sresult=system("mode spe auto");
                     if (trig==-1)
                          longjmp(mark,-1);
                }


                result = gets(numbers);
                points_scan = atoi(numbers);

           }
           d_norm[0] = (double *)calloc(points_scan,
                          sizeof(double));
           d_norm[1] = (double *)calloc(points_scan,
                          sizeof(double));
           d_norm[2] = (double *)calloc(points_scan,
                          sizeof(double));
           d_norm[3] = (double *)calloc(points_scan,
                          sizeof(double));
           for (i = 0; i < 4; i++) {
                if (d_norm[i] == NULL){
                     points_scan=0;
                     _settextposition(10, 15);
                     rcoord = _gettextposition();

                     sprintf(buffer,"calloc failed on %i",i);
                     _outtext(buffer);
                     rcoord.row++;
                     _settextposition(rcoord.row, rcoord.col);

                     _outtext("hit a key to continue");
                     while (i == 0)
                          i = kbhit();
                     i = getche();

                }

           }
      }

      void s_step(n)
      int     n;
```

```
{
     int trig,sresult;
     char    numbers[10];
     char    *result, *stage_string();
     if (stage_step != -1003 && n == 0)

/*        This was left here is case there was stage setup
*    required at the current time there is not
*/
/*        put in stage setup stuff
*/

          return;

     _setvideomode(_DEFAULTMODE);
     _settextcolor(tcolor);
     _settextposition(10, 15);
     rcoord = _gettextposition();
     _outtext(stage_string(2));
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     if (trig_check==0)
     {
          trig=0;
          sresult=system("mode spe com");
          while(kbhit()==0&&trig==0)
               trig=check();
          ibloc(ddg);
          sresult=system("mode spe auto");
          if (trig==-1)
               longjmp(mark,-1);
     }


     result = gets(numbers);
     stage_step = atoi(numbers);
     if (test_num(numbers) == -1)
          stage_step = -1003;
     while (stage_step == -1003) {
          _setvideomode(_DEFAULTMODE);
          _settextcolor(ecolor);
          _settextposition(10, 15);
          rcoord = _gettextposition();
          _outtext(error_string(5));
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          _settextcolor(tcolor);
          _outtext(stage_string(2));
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          if (trig_check==0)
          {
```

```
                trig=0;
                sresult=system("mode spe com");
                while(kbhit()==0&&trig==0)
                     trig=check();
                ibloc(ddg);
                sresult=system("mode spe auto");
                if (trig==-1)
                     longjmp(mark,-1);
           }


           result = gets(numbers);
           stage_step = atoi(numbers);
           if (test_num(numbers) == -1)
                stage_step = -1003;

      }
/*         put in stage setup stuff
*/
}

void p_step(n)
int     n;
{
     int trig,sresult;
     char    numbers[10];
     char    *result, *stage_string();
     if (shots_step != 0 && n == 0)

/*         This was left here is case there was # of shots
*    setup required; at the current time there is not
*/
          return;

     _setvideomode(_DEFAULTMODE);
     _settextcolor(tcolor);
     _settextposition(10, 15);
     rcoord = _gettextposition();
     _outtext(stage_string(3));
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     if (trig_check==0)
     {
          trig=0;
          sresult=system("mode spe com");
          while(kbhit()==0&&trig==0)
               trig=check();
          ibloc(ddg);
          sresult=system("mode spe auto");
          if (trig==-1)
               longjmp(mark,-1);
     }
```

```
     result = gets(numbers);
     shots_step = atol(numbers);
     while (shots_step <= 1 || shots_step == 0) {
          _setvideomode(_DEFAULTMODE);
          _settextcolor(ecolor);
          _settextposition(10, 15);
          rcoord = _gettextposition();
          _outtext(error_string(5));
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          _settextcolor(tcolor);
          _outtext(stage_string(3));
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          if (trig_check==0)
          {
               trig=0;
               sresult=system("mode spe com");
               while(kbhit()==0&&trig==0)
                    trig=check();
               ibloc(ddg);
               sresult=system("mode spe auto");
               if (trig==-1)
                    longjmp(mark,-1);
          }


          result = gets(numbers);
          shots_step = atoi(numbers);
     }

}

void stage_start(n)
int     n;
{
     int trig,sresult;
     char    numbers[10];
     char    *result, *stage_string();
     if (stage_beg != 0 && n == 0)
          return;


     _setvideomode(_DEFAULTMODE);
     _settextcolor(tcolor);
     _settextposition(10, 15);
     rcoord = _gettextposition();
     _outtext(stage_string(4));
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     if (trig_check==0)
```

```
        {
            trig=0;
            sresult=system("mode spe com");
            while(kbhit()==0&&trig==0)
                trig=check();
            ibloc(ddg);
            sresult=system("mode spe auto");
            if (trig==-1)
                longjmp(mark,-1);
        }


        result = gets(numbers);
        stage_beg = atol(numbers);
        if (test_num(numbers) == -1)
            stage_beg = -100362;
        while (stage_beg == -100362) {
            _setvideomode(_DEFAULTMODE);
            _settextcolor(ecolor);
            _settextposition(10, 15);
            rcoord = _gettextposition();
            _outtext(error_string(5));
            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            _settextcolor(tcolor);
            _outtext(stage_string(4));
            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            if (trig_check==0)
            {
                trig=0;
                sresult=system("mode spe com");
                while(kbhit()==0&&trig==0)
                    trig=check();
                ibloc(ddg);
                sresult=system("mode spe auto");
                if (trig==-1)
                    longjmp(mark,-1);
            }


            result = gets(numbers);
            stage_beg = atoi(numbers);
            if (test_num(numbers) == -1)
                stage_beg = -100362;
        }

    /*          put in stage setup stuff
    */

    }
```

```
void gate(n)
int     n;
{
     int trig,sresult;
     unsigned int    DATA, gwidth, delay, gate_temp;
     unsigned int   c,   fun, A, X, Q,i;
     char    delays[20], gwidths[3];
     char    *result;

     if (gate_chan_a != 0 && gate_chan_b != 0 && n == 0) {

          A = 0;
          fun = 17;
          c = camo(&GATE, &fun, &A, &gate_chan_a, &Q, &X);
          if(Q!=1 || X!=1){
               camerr(GATE,fun,A,gate_chan_a,Q,X);
               return;
          }

          A = 1;
          c = camo(&GATE, &fun, &A, &gate_chan_b, &Q, &X);
          if(Q!=1 || X!=1){
               camerr(GATE,fun,A,gate_chan_b,Q,X);
               return;
          }

          return;
     }

     _setvideomode(_DEFAULTMODE);
     _settextcolor(tcolor);
     _settextposition(10, 15);
     rcoord = _gettextposition();
     for(i=1;i<7;i++){
          _outtext(gate_string(i));
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
     }
     if (trig_check==0)
     {
          trig=0;
          sresult=system("mode spe com");
          while(kbhit()==0&&trig==0)
               trig=check();
          ibloc(ddg);
          sresult=system("mode spe auto");
          if (trig==-1)
               longjmp(mark,-1);
     }
```

```
result = gets(gwidths);
gwidth = atoi(gwidths);
if (test_num(gwidths) == -1) {
     gwidth = 5;

}

while (gwidth < 0 || gwidth > 3) {
     _setvideomode(_DEFAULTMODE);
     _settextposition(10, 15);
     rcoord = _gettextposition();
     _outtext(gate_string(1));
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     _settextcolor(6);
     _outtext(gate_string(10));
     _settextcolor(tcolor);
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     for (i=3;i<7;i++){
          _outtext(gate_string(i));
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
     }
     if (trig_check==0)
     {
          trig=0;
          sresult=system("mode spe com");
          while(kbhit()==0&&trig==0)
               trig=check();
          ibloc(ddg);
          sresult=system("mode spe auto");
          if (trig==-1)
               longjmp(mark,-1);
     }


     result = gets(gwidths);
     gwidth = atoi(gwidths);
     if (test_num(gwidths) == -1) {
          gwidth = 5;

     }
}
rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext(gate_string(7));
if (trig_check==0)
{
     trig=0;
     sresult=system("mode spe com");
     while(kbhit()==0&&trig==0)
```

```
            trig=check();
        ibloc(ddg);
        sresult=system("mode spe auto");
        if (trig==-1)
            longjmp(mark,-1);
    }


    result = gets(delays);
    delay = atoi(delays);
    while (delay >= 1000 || delay <= 0) {
        _setvideomode(_DEFAULTMODE);
        _settextposition(10, 15);
        rcoord = _gettextposition();

        _settextposition(rcoord.row, rcoord.col);
        _outtext(gate_string(9));
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(gate_string(7));
        if (trig_check==0)
        {
            trig=0;
            sresult=system("mode spe com");
            while(kbhit()==0&&trig==0)
                trig=check();
            ibloc(ddg);
            sresult=system("mode spe auto");
            if (trig==-1)
                longjmp(mark,-1);
        }


        result = gets(delays);
        delay = atoi(delays);
    }
    DATA = gwidth << 4;
    gate_temp = DATA;
    DATA = gate_temp << 10;
    gate_temp = DATA + delay;
    DATA = gate_temp;
    gate_chan_a = DATA;

    _setvideomode(_DEFAULTMODE);

    _settextposition(10, 15);
    rcoord = _gettextposition();
    DATA = 0;
    _outtext(gate_string(8));
    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    for (i=2;i<7;i++){
```

```
        _outtext(gate_string(i));
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
    }
    if (trig_check==0)
    {
        trig=0;
        sresult=system("mode spe com");
        while(kbhit()==0&&trig==0)
            trig=check();
        ibloc(ddg);
        sresult=system("mode spe auto");
        if (trig==-1)
            longjmp(mark,-1);
    }


    result = gets(gwidths);
    gwidth = atoi(gwidths);
    if (test_num(gwidths) == -1) {
        gwidth = 5;
    }
    while (gwidth < 0 || gwidth > 3) {
        _setvideomode(_DEFAULTMODE);
        _settextposition(10, 15);
        rcoord = _gettextposition();
        _outtext(gate_string(8));
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _settextcolor(6);
        _outtext(gate_string(10));
        _settextcolor(tcolor);
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(gate_string(3));
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(gate_string(4));
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(gate_string(5));
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(gate_string(6));
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        if (trig_check==0)
        {
            trig=0;
            sresult=system("mode spe com");
            while(kbhit()==0&&trig==0)
                trig=check();
```

```
                ibloc(ddg);
                sresult=system("mode spe auto");
                if (trig==-1)
                     longjmp(mark,-1);
          }


          result = gets(gwidths);
          gwidth = atoi(gwidths);
          if (test_num(gwidths) == -1) {
               gwidth = 5;
          }
     }
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     _outtext(gate_string(7));
     if (trig_check==0)
     {
          trig=0;
          sresult=system("mode spe com");
          while(kbhit()==0&&trig==0)
               trig=check();
          ibloc(ddg);
          sresult=system("mode spe auto");
          if (trig==-1)
               longjmp(mark,-1);
     }


     result = gets(delays);
     delay = atoi(delays);
     while (delay >= 1000 || delay <= 0) {
          _setvideomode(_DEFAULTMODE);
          _settextposition(10, 15);
          rcoord = _gettextposition();
          _settextposition(rcoord.row, rcoord.col);
          _outtext(gate_string(9));
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          _outtext(gate_string(7));
          if (trig_check==0)
          {
               trig=0;
               sresult=system("mode spe com");
               while(kbhit()==0&&trig==0)
                    trig=check();
               ibloc(ddg);
               sresult=system("mode spe auto");
               if (trig==-1)
                    longjmp(mark,-1);
          }
```

```
          result = gets(delays);
          delay = atoi(delays);
     }
     DATA = gwidth << 4;
     gate_temp = DATA;
     DATA = gate_temp << 10;
     gate_temp = DATA + delay;
     DATA = gate_temp;
     gate_chan_b = DATA;

}


void read_ped(n)
int     n;
{
     int trig,sresult;
     int      i,c,fun2;
     int      fun,X,Q,junk=100;
     char    numbers[10];
     char    *result, *stage_string();

     fun = 17;
     fun2=1;

     if (pedestal[0] != -1 && n == 0) {

/*        The pedestal values are sent to the gate and delay
*    generator here.  See the DSP manual and the LeCroy gate
*    and delay manual
*/

          for (i = 0; i != 3; i++) {
               c = camo(&AD, &fun, &i,&pedestal[i] , &Q, &X);
          }
          for (i = 1; i != 15; i++) {
               c = camo(&AD, &fun, &i,&junk , &Q, &X);
          }
          i=15;
          c = camo(&AD, &fun, &i,&pedestal[1] , &Q, &X);


          return;

     }

     _settextcolor(tcolor);
     for (i = 0; i != 3; i++) {
          _setvideomode(_DEFAULTMODE);
          _settextposition(10, 15);
          rcoord = _gettextposition();
          printf("Input pedestal %i", i);
```

```
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        if (trig_check==0)
        {
             trig=0;
             sresult=system("mode spe com");
             while(kbhit()==0&&trig==0)
                  trig=check();
             ibloc(ddg);
             sresult=system("mode spe auto");
             if (trig==-1)
                  longjmp(mark,-1);
        }


        result = gets(numbers);
        pedestal[i] = atoi(numbers);
        if (test_num(numbers) == -1)
             pedestal[i] = -1003;
        while (pedestal[i] < 0 || pedestal[i] > 100) {
             _setvideomode(_DEFAULTMODE);
             _settextposition(10, 15);
             rcoord = _gettextposition();
             _settextcolor(ecolor);
             _outtext(error_string(5));
             rcoord.row++;
             _settextposition(rcoord.row, rcoord.col);
             _settextcolor(tcolor);
             printf("Input pedestal %i", i);
             rcoord.row++;
             _settextposition(rcoord.row, rcoord.col);
             if (trig_check==0)
             {
                  trig=0;
                  sresult=system("mode spe com");
                  while(kbhit()==0&&trig==0)
                       trig=check();
                  ibloc(ddg);
                  sresult=system("mode spe auto");
                  if (trig==-1)
                       longjmp(mark,-1);
             }


             result = gets(numbers);
             pedestal[i] = atoi(numbers);
             if (test_num(numbers) == -1)
                  pedestal[i] = -1003;
        }
}
```

```
}

void read_comment()

{

     int      i, ch;

     int trig,sresult;
     _setvideomode(_DEFAULTMODE);
     _settextcolor(tcolor);
     _settextposition(10, 15);
     rcoord = _gettextposition();
     _settextposition(rcoord.row, rcoord.col);
     _outtext(stage_string(5));
     rcoord.row++;
     _settextposition(rcoord.row, 0);
     if (trig_check==0)
     {
          trig=0;
          sresult=system("mode spe com");
          while(kbhit()==0&&trig==0)
               trig=check();
          ibloc(ddg);
          sresult=system("mode spe auto");
          if (trig==-1)
               longjmp(mark,-1);
     }


     for (i = 0; (i < 999) && (ch = getchar()) != '~'; i++)
          comment[i] =(char) ch;
     comment[i] =(char) ch;
     i++;
     ch = getchar();
     comment[i] =(char) ch;
     i++;
     comment[i] = '\0';
     return;
}

void display(void)

{



     int      real_width[4];
     int      i = 0;
     unsigned int gate_temp;
     char buffer[30];
```

```
/*      This changes the binary variables as described in
 *   the LeCroy gate and delay manual into a more
 *   understandable form
*/

     if (gate_chan_a != 0) {
          delay_a = gate_chan_a & 01777;
          gate_temp = gate_chan_a >> 14;
          gwidth_a = gate_temp & 03;
          delay_b = gate_chan_b & 01777;
          gate_temp = gate_chan_b >> 14;
          gwidth_b = gate_temp & 03;
     }


     real_width[0] = 10;
     real_width[1] = 30;
     real_width[2] = 100;
     real_width[3] = 300;
     _setvideomode(_DEFAULTMODE);
     _settextcolor(tcolor);
     _settextposition(10, 15);
     rcoord = _gettextposition();
     _outtext( set_string(1));
     if (points_scan != 0) {
          rcoord.col = rcoord.col + 40;
          _settextposition(rcoord.row, rcoord.col);
          sprintf(buffer," %i", points_scan);
          _outtext(buffer);
          rcoord.col = rcoord.col - 40;
     }
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     _outtext( set_string(2));
     if (stage_step != 0) {
          rcoord.col = rcoord.col + 40;
          _settextposition(rcoord.row, rcoord.col);
          sprintf(buffer," %i", stage_step);
          _outtext(buffer);
          rcoord.col = rcoord.col - 40;
     }
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     _outtext( set_string(3));
     if (shots_step != 0) {
          rcoord.col = rcoord.col + 40;
          _settextposition(rcoord.row, rcoord.col);
          sprintf(buffer," %li", shots_step);
          _outtext(buffer);
          rcoord.col = rcoord.col - 40;
     }
     rcoord.row++;
```

```
_settextposition(rcoord.row, rcoord.col);
_outtext( set_string(4));
if (stage_beg != 0) {
     rcoord.col = rcoord.col + 40;
     _settextposition(rcoord.row, rcoord.col);
     sprintf(buffer," %li", stage_beg);
     _outtext(buffer);
     rcoord.col = rcoord.col - 40;
}
rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext( set_string(5));
if (delay_a != 0) {
     rcoord.col = rcoord.col + 40;
     _settextposition(rcoord.row, rcoord.col);
     sprintf(buffer,"%i ns, %i ns", real_width[gwidth_a],
             delay_a);
     _outtext(buffer);
     rcoord.col = rcoord.col - 40;
}
rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext( set_string(6));
if (delay_a != 0) {
     rcoord.col = rcoord.col + 40;
     _settextposition(rcoord.row, rcoord.col);
     sprintf(buffer,"%i ns, %i ns", real_width[gwidth_b],
             delay_b);
     _outtext(buffer);
     rcoord.col = rcoord.col - 40;
}
rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext( set_string(7));
if (pedestal[0] != -1) {
     rcoord.col = rcoord.col + 40;
     _settextposition(rcoord.row, rcoord.col);
     sprintf(buffer,"%i   ,  %i   ,   %i",   pedestal[0],
             pedestal[1], pedestal[2]);
     _outtext(buffer);
     rcoord.col = rcoord.col - 40;
}

rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext( set_string(8));
rcoord.row++;
_settextcolor(ecolor);
_settextposition(rcoord.row, 0);
if(comment[0]!='~')
     _outtext(comment);
rcoord = _gettextposition();
```

```
    rcoord.row++;
    _settextcolor(tcolor);
    _settextposition(rcoord.row, 10);
    _outtext( set_string(9));
    while (i == 0)
         i = kbhit();
    i = getche();
}

void channel_display(void)

{

    int trig,sresult;
    char inputs2[3];
    char *input2;
    _setvideomode(_DEFAULTMODE);
    _settextcolor(ecolor);
    _settextposition(10, 15);
    rcoord = _gettextposition();

    _outtext("Input the first channel to be displayed");
    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext("either 0, 1, 2, or 3");
    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    if (trig_check==0)
    {
         trig=0;
         sresult=system("mode spe com");
         while(kbhit()==0&&trig==0)
              trig=check();
         ibloc(ddg);
         sresult=system("mode spe auto");
         if (trig==-1)
              longjmp(mark,-1);
    }


    input2 = gets(inputs2);
    channel_dis1 = atoi(input2);
    _setvideomode(_DEFAULTMODE);
    _settextcolor(ecolor);
    _settextposition(10, 15);
    rcoord = _gettextposition();
    _outtext("Input the second channel to be displayed");
    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext("either 0, 1, 2, or 3");
    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
```

```
        if (trig_check==0)
        {
             trig=0;
             sresult=system("mode spe com");
             while(kbhit()==0&&trig==0)
                  trig=check();
             ibloc(ddg);
             sresult=system("mode spe auto");
             if (trig==-1)
                  longjmp(mark,-1);
        }


        input2 = gets(inputs2);
        channel_dis2 = atoi(input2);

   }


   void storefile(void)

   {
        FILE * stream;
        int trig,sresult;
        char inputs[10], inputs2[2];
        char *input, *input2, *prompt_string();
        inputs2[0] = 'n';
        while ((inputs2[0] == 'n' || inputs2[0] == 'N') && stream
               != NULL) {
             _setvideomode(_DEFAULTMODE);
             _settextposition(10, 15);
             rcoord = _gettextposition();

             _outtext(prompt_string(1));
             if (trig_check==0)
             {
                  trig=0;
                  sresult=system("mode spe com");
                  while(kbhit()==0&&trig==0)
                       trig=check();
                  ibloc(ddg);
                  sresult=system("mode spe auto");
                  if (trig==-1)
                       longjmp(mark,-1);
             }


             input = gets(inputs);
             if ((stream = fopen(input, "rb")) != NULL) {
                  fclose(stream);
                  rcoord.row++;
                  _settextcolor(ecolor);
```

```
                    _settextposition(rcoord.row, rcoord.col);
                    _outtext( prompt_string(2));
                    rcoord.row++;
                    _settextposition(rcoord.row, rcoord.col);
                    _outtext( prompt_string(3));
                    rcoord.row++;
                    _settextposition(rcoord.row, rcoord.col);
                    _outtext( prompt_string(6));
                    if (trig_check==0)
                    {
                         trig=0;
                         sresult=system("mode spe com");
                         while(kbhit()==0&&trig==0)
                              trig=check();
                         ibloc(ddg);
                         sresult=system("mode spe auto");
                         if (trig==-1)
                              longjmp(mark,-1);
                    }


                    input2 = gets(inputs2);
                    if (inputs2[0] == 'Q' || inputs2[0] == 'q') {
                         _setvideomode(_DEFAULTMODE);
                         return;
                    }
          }
     }
     stream = fopen(input, "wb");
     fprintf( stream,"djr");
     fprintf(stream, "%i %i ", gate_chan_a, gate_chan_b);
     fprintf(stream, "%i %i ", points_scan, stage_step);
     fprintf(stream, "%i %i ", high,low);
     fprintf(stream, "%li %li ", shots_step, stage_beg);
     fprintf(stream, "%i %i %i ", pedestal[0], pedestal[1],
             pedestal[2]);
     fprintf(stream, " %s", comment);
     fclose(stream);
     _setvideomode(_DEFAULTMODE);
     return;
}

void ddg_check(void)
{
     int trig,sresult;
     char inputs2[3];
     char *input2;
     _setvideomode(_DEFAULTMODE);
     _settextcolor(tcolor);
     _settextposition(10, 15);
     rcoord = _gettextposition();
     if (trig_check==0)
```

```
            _outtext("Trigger checking is enabled now");
        if (trig_check==1)
            _outtext("Trigger checking is not enabled now");

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext("Input a 0 for trigger checking or a 1 ,");
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(" for no trigger checking ");

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        if (trig_check==0)
        {
            sresult=system("mode spe com");
            while(kbhit()==0&&trig==0)
                trig=check();
            ibloc(ddg);
            sresult=system("mode spe auto");
            if (trig==-1)
                longjmp(mark,-1);
        }


        input2 = gets(inputs2);
        trig_check = atoi(input2);
        _setvideomode(_DEFAULTMODE);
    }
```

E. PROGRAM LISTING MULTI

```
/*      Program "MULTI"
 *      Daniel Russell
 *      Aug. 6, 1990
 *
 *              The file Multi.c contains the following subroutines.
 *
 *              void set_multi()
 *              void open_data_file()
 *              void take_data()
 *              void top()
 */

#include <string.h>
#include <conio.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <graph.h>
#include <time.h>
#include <math.h>
#include <float.h>
#include "dan.h"
#include <setjmp.h>

extern int      channel_dis1, channel_dis2;
extern int      channel_color1, channel_color2, box_color;
                time_t ltime;
extern int      multi_count, reverse_flag, scan;
extern char     file_name[7], file_num[3];
extern int      gate_chan_a, gate_chan_b;
extern int      points_scan, stage_step;
extern long int      shots_step, stage_beg;
extern int      pedestal[3], multi_flag;
extern char     file_name[7], file_num[3], data_file[12];
extern char     comment[1000];
extern long int      rect_xmin,    rect_ymin,    rect_xmax,
rect_ymax;
extern int      tcolor;
extern int      ecolor;

extern struct videoconfig vc;
extern struct rccoord rcoord;
extern int      ddg, trig_check;
extern jmp_buf mark;

/*              This sets up the variables for taking multiple scans
*/

void set_multi(void)
```

```
{
     char inputs[4];
     char *input;
     int  trig, sresult;
     multi_count = 0;
     reverse_flag = 0;
     while (multi_count == 0 || multi_count != 1) {
          _setvideomode(_DEFAULTMODE);

          _settextcolor(tcolor);
          _settextposition(10, 15);
          rcoord = _gettextposition();
          _outtext(multi_string(9));
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          _outtext(multi_string(10));
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          if (trig_check == 0) {
               trig = 0;
               sresult = system("mode spe com");
               while (kbhit() == 0 && trig == 0)

                    trig = check();
               ibloc(ddg);
               sresult = system("mode spe auto");
               if (trig == -1)
                    longjmp(mark, -1);
          }


          input = gets(inputs);
          multi_count = atoi(inputs);
          if (multi_count == 2) {
               multi_count = 0;
               return;
          }
     }
     _setvideomode(_DEFAULTMODE);
     _settextcolor(tcolor);
     _settextposition(10, 15);
     rcoord = _gettextposition();
     _outtext(multi_string(1));
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     if (trig_check == 0) {
          trig = 0;
          sresult = system("mode spe com");
          while (kbhit() == 0 && trig == 0)

               trig = check();
```

```
          ibloc(ddg);
          sresult = system("mode spe auto");
          if (trig == -1)
               longjmp(mark, -1);
     }

     input = gets(inputs);
     multi_count = atoi(inputs);

     while (multi_count % 2 != 0 || multi_count == 0) {
          _setvideomode(_DEFAULTMODE);
          _settextposition(10, 15);
          rcoord = _gettextposition();
          _settextcolor(ecolor);
          _outtext(multi_string(2));
          rcoord.row++;
          _settextcolor(tcolor);
          _settextposition(rcoord.row, rcoord.col);

          _outtext(multi_string(1));

          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          if (trig_check == 0) {
               trig = 0;
               sresult = system("mode spe com");
               while (kbhit() == 0 && trig == 0)

                    trig = check();
               ibloc(ddg);
               sresult = system("mode spe auto");
               if (trig == -1)
                    longjmp(mark, -1);
          }

          input = gets(inputs);
          multi_count = atoi(inputs);
     }
     _setvideomode(_DEFAULTMODE);
     _settextposition(10, 15);
     _settextcolor(tcolor);
     rcoord = _gettextposition();
     _outtext(multi_string(3));
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);

     _outtext(multi_string(4));
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     if (trig_check == 0) {
```

```
			trig = 0;
			sresult = system("mode spe com");
			while (kbhit() == 0 && trig == 0)

				trig = check();
			ibloc(ddg);
			sresult = system("mode spe auto");
			if (trig == -1)
				longjmp(mark, -1);
		}


		input = gets(inputs);
		reverse_flag = atoi(inputs);
		while (reverse_flag != 2 && reverse_flag != 1) {
			_setvideomode(_DEFAULTMODE);
			_settextposition(10, 15);
			rcoord = _gettextposition();
			_settextcolor(ecolor);
			_outtext(multi_string(5));
			rcoord.row++;
			_settextposition(rcoord.row, rcoord.col);
			_settextcolor(tcolor);
			_outtext(multi_string(3));
			rcoord.row++;
			_settextposition(rcoord.row, rcoord.col);

			_outtext(multi_string(4));
			rcoord.row++;
			_settextposition(rcoord.row, rcoord.col);
			if (trig_check == 0) {
				trig = 0;
				sresult = system("mode spe com");
				while (kbhit() == 0 && trig == 0)

					trig = check();
				ibloc(ddg);
				sresult = system("mode spe auto");
				if (trig == -1)
					longjmp(mark, -1);
			}


			input = gets(inputs);
			reverse_flag = atoi(inputs);
		}

}

void open_data_file(void)
```

```
/*        This subroutine makes sure that the file name you
 *        want to store the data in doesn't already exist on
 *        the disk before you take the data.
 */

{

     char inputs2[2];

     int  trig, sresult;
     char *input,  *input2;
     FILE * stream;


     inputs2[0] = 'n';
     stream == 0;
     while ((inputs2[0] == 'n' || inputs2[0] == 'N') && stream
             != NULL) {
          _setvideomode(_DEFAULTMODE);
          _settextcolor(tcolor);
          _settextposition(10, 15);
          rcoord = _gettextposition();
          _outtext(multi_string(7));
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          if (trig_check == 0) {
               trig = 0;
               sresult = system("mode spe com");
               while (kbhit() == 0 && trig == 0)

                    trig = check();
               ibloc(ddg);
               sresult = system("mode spe auto");
               if (trig == -1)
                    longjmp(mark, -1);
          }


          input = gets(file_name);
          strcpy(data_file, file_name);
          strcat(data_file, ".");
          _setvideomode(_DEFAULTMODE);
          _settextposition(10, 15);
          rcoord = _gettextposition();
          _outtext(multi_string(8));
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          if (trig_check == 0) {
               trig = 0;
               sresult = system("mode spe com");
               while (kbhit() == 0 && trig == 0)
```

```
                trig = check();
            ibloc(ddg);
            sresult = system("mode spe auto");
            if (trig == -1)
                longjmp(mark, -1);
        }

        input = gets(file_num);
        strcat(data_file, file_num);
        if ((stream = fopen(data_file, "rb")) != NULL) {
            fclose(stream);
            rcoord.row++;
            _settextcolor(ecolor);
            _settextposition(rcoord.row, rcoord.col);
            _outtext( prompt_string(2));
            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            _outtext( prompt_string(3));
            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            _outtext( prompt_string(7));
            if (trig_check == 0) {
                trig = 0;
                sresult = system("mode spe com");
                while (kbhit() == 0 && trig == 0)

                    trig = check();
                ibloc(ddg);
                sresult = system("mode spe auto");
                if (trig == -1)
                    longjmp(mark, -1);
            }

            input2 = gets(inputs2);
        }

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(data_file);
        rcoord.row++;

    }
}

void take_data(void)

{
    float     ymin = (float) -1., ymax = (float)1., xmin =
```

```
(float)0., xmax = (float)1000.;


     screen();


}

/*        This subroutine plots the information at the top of
*         the screen.  The variable n is the number of scans
*         taken in multi mode.   This allows top() to print
*         the right file name for that scan (n starts counting
*         at 0).
*/

void top(n)
int  n;
{
     char *window_string(), *p, file_numt[3];
     char buffer[50];

     float     ymin = (float) -1., ymax = (float)1., xmin =
               (float)0., xmax = (float)1000.;
     int  num1, i;
     strcpy(data_file, file_name);
     strcat(data_file, ".");
     num1 = atoi(file_num);
     num1 = num1 + n;
     p = itoa(num1, file_numt, 10);
     strcat(data_file, file_numt);
     if (_setvideomode(_ERESCOLOR))
          ;
     else if (_setvideomode(_HERCMONO))
          ;
     else {
          _outtext("Graphics  not  supported,  is  msherc.com
                  loaded if herc");
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          _outtext("Hit a key to continue");
          while (i == 0)
               i = kbhit();
          i = getche();
          return;
     }
     _getvideoconfig(&vc);
     _settextcolor(tcolor);
     _settextposition(1, 1);
     rcoord = _gettextposition();
     _outtext(window_string(1));
```

```
        _outtext(data_file);
        rcoord.col = rcoord.col + 25;
        _settextposition(rcoord.row, rcoord.col);

        time(&ltime);
        _outtext((ctime(&ltime)));
        rcoord.col = rcoord.col + 35;
        _settextposition(rcoord.row, rcoord.col);
        sprintf(buffer, "display %i  ", channel_dis1);
        _settextcolor(channel_color1);
        _outtext(buffer);
        _settextcolor(tcolor);
        _settextposition(2, 1);
        rcoord = _gettextposition();
        _outtext(window_string(3));
        sprintf(buffer, "%li", shots_step);
        _outtext(buffer);
        rcoord.col = rcoord.col + 20;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(window_string(4));
        sprintf(buffer, "%li", stage_beg);
        _outtext(buffer);
        rcoord.col = rcoord.col + 20;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(window_string(5));
        sprintf(buffer, "%li", (long int) stage_beg + (long int)
                points_scan * (long int)stage_step);
        _outtext(buffer);
        if (multi_count != 1) {
             _settextposition(3, 1);
             rcoord = _gettextposition();
             _outtext(window_string(6));
             rcoord.col = rcoord.col + 20;
             _settextposition(rcoord.row, rcoord.col);
             _outtext(window_string(7));
             sprintf(buffer, "%i", multi_count);
             _outtext(buffer);
             if (reverse_flag == 2)    {
                  rcoord.col = rcoord.col + 20;
                  _settextposition(rcoord.row, rcoord.col);
                  _outtext(window_string(8));
             }
        }
        rcoord.col = rcoord.col + 20;
        _settextposition(rcoord.row, rcoord.col);
        sprintf(buffer, "display %i  ", channel_dis2);
        _settextcolor(channel_color2);
        _outtext(buffer);
        _settextcolor(tcolor);

}
```

F. PROGRAM LISTING CAMAC

```
/*    Program "CAMAC"
*     Daniel Russell
*     Aug. 6, 1990
*
*          The file Camac.c contains the following subroutines.
*
*          void setupcamac()
*/

 #include "dan.h"

 void setupcamac(void)
 {


     int  Q, X, l, A[16], crate, c, bytes, ads, QBL, fun;
     unsigned int   DATA[16];

     extern int     low, high;
     extern long int     shots_step;
     extern int     AD;

     Q = 0;
     X = 0;

 /*        All the subroutines in this file are described in
 *         the DSP manual.  The Assembler was modified for all
 *         these subroutines to follow C language calling
 *         conventions.
 */

     fun = 0;
     l = 64;
     A[0] = 0;
     crate = 1;
     crateset(&crate);
     c = camcl(&l);
     l = 4;
     c = camcl(&l);
     c = cami(&AD, &fun, A, DATA, &Q, &X);

 /*        This next camo sets up the A to D in the
 *         appropriate way.  See pg 1-5 in the LeCroy A to D
 *         manual.  Basically, bit 12 is high meaning pedestal
 *         subtraction is done by the A to D, bit 13 is high
 *         and Camac data compression is enabled, bit 14 is
 *         high so sequential readout occurs, bit 15 is high
 *         so a LAM is set as soon as data is ready to be
 *         read, and bit 16 is set  telling the A to D to
 *         suppress 0's and overflows on CAMAC readout.
```

```
*/

     DATA[0] = 0174000;
     fun = 16;
     c = camo(&AD, &fun, A, DATA, &Q, &X);

     fun = 0;
     c = cami(&AD, &fun, A, DATA, &Q, &X);
     if (DATA[0] != 064000 && Q != 1 && X != 1) {
          camerr(AD, fun, A[0], DATA[0], Q, X);
          return;
     }

     DATA[0] = 0x000;
     bytes = 2;
     ads = 17;
     QBL = 1;

/*        This subroutine is slightly different than the one
 *        described in the DSP manual.  It also passes to the
 *        assembler code the low and high for bounds
 *        checking, and the number of shots per step.
 */

     dmaset(&crate,    &bytes,    &QBL,    &ads,    &low,    &high,
shots_step);

     p_scan(0);
     p_step(0);
     stage_start(0);
     gate(0);
     read_ped(0);

     l = 0;
     c = camcl(&l);
     return;
}
```

G. PROGRAM LISTING CONST

```
/*    Program "CONST"
*     Daniel Russell
*     Aug. 6, 1990
*           The    file    Const.c    contains    the    following
subroutines. *
*           char *multi_string()
*           char *window_string()
*           char *error_string()
*           char *gate_string()
*           char *main_string()
*           char *setup_string()
*           char *prompt_string()
*           char *look_string()
*           char *stage_string()
*           char *data_string()
*           char *set_string()
*           char *odd_string()
*/

#include "dan.h"

/*          This file contains most of the string constants
used *      in the program.  This was done for two reasons: it
saves *     data space for strings that are used more than
once, and *      also allows some menus to be output with for
loops *     instead of having one line for each output.
Basically, *     these are subroutines that return a string.
The first *      part is the definition of the static char
which is *       basically an array of character strings.  In
*     multi_string[0], string[0] is equal to "ill strng".
That *      was put in there so that in the development phase
errors *    could be caught easily.  Also for formatting for
the *       thesis some constant strings would not fit on one
line *      and a newline character was put in.  This is not
allowed * and the source file generated from the thesis must
be * changed to one line constants
*/

char *multi_string(n)
int  n;
{
     static char    *string[] = {
          "ill strng",
          " Input number of scans (must be even number)",
          " Must be an even number",
          " 1: Stage takes data in one direction",
          " 2: Stage takes data in both directions",
          " Input 2 or 1 ",
          " 5: Return to Main Menu",
          "Input Data File name",
```

```
            "Input Data File number",
            " 1: Turn multi on",
            " 2: Turn multi off",
            "Input Ascii data file name"
     };
/*          The return line should be interpreted to say if n
is * less than 1  or greater than 11 return string[0], else
*    return string[n]
*/
     return((n < 1 || n > 11) ? string[0] : string[n]);
}


char *window_string(n)
int  n;
{
     static char    *string[] = {
          "ill strng",
          "File Name ",
          " Date ",
          "Shots ",
          "Begin ",
          "End " ,
          "Multi is on",
          " Scans ",
          "Reverse",
          "Input Y if you wish to continue,",
          "Input S if you wish to save,and quit",
          "Input Q if you wish to quit without saving",
          "remember if you quit to put the stage back to
origin"
     };
     return((n < 1 || n > 12 ) ? string[0] : string[n]);
}


char *error_string(n)
int  n;
{
     static char    *string[] = {
          "ill strng",
          "gate not set",
          "stage not set",
          "shots not set",
          "pedestal not set",
          "not an acceptable number",
          "bounds not set",
          "write error occurred "
     };
     return((n < 1 || n > 7) ? string[0] : string[n]);
}
```

```
char *gate_string(n)
int  n;
{
     static char    *string[] = {
          "ill strng",
          "Set up Channel A",
          "choose a gate width  ",
          "0: 10 nsec  " ,
          "1: 30 nsec  ",
          "2: 100 nsec  ",
          "3: 300 nsec  ",
          "input delay in nsec  ",
          "Set up Channel B",
          "delay < 1000 nsec & delay > 0  ",
          "Choice must be between 0, 1, 2 or 3"
     };
     return((n < 1 || n > 10) ? string[0] : string[n]);
}


char *main_string(n)
int  n;
{
     static char    *string[] = {
          "ill strng",
          " 1: Read in Setup File",
          " 2: Create Setup File ",
          " 3: Take DATA",
          " 4: Look at DATA",
          /*    " 5: Add DATA files",   */
          " 5: To check trigger, on not to check trigger ?",
          " 6: Quit",
          "this video mode is not supported"
     };
     return((n < 1 || n > 6) ? string[0] : string[n]);
}


char *setup_string(n)
int  n;
{
     static char    *string[] = {
          "ill strng",
          " 1: # of Points in Scan ",
          " 2: # of Stage steps between points ",
          " 3: # of Shots/Step",
          " 4: Stage Starting Position",
          " 5: Set Gate & Delay",
          " 6: Set Pedestals",
          " 7: Comment of 5 sentences or less ",
          " 8: Display All ",
          " 9: Save setup",
```

```
            "10: Choose which channels to display",
            "11: Take data for setup without saving",
            "12: Run Norm",
            "13: Set Bounds",
            "14: Quit"
     };
     return((n < 1 || n > 14) ? string[0] : string[n]);
}


char *prompt_string(n)
int  n;
{
     static char    *string[] = {
          "ill strng",
          "Type File Name ",
          "File Already Exists !!!",
          " Do you Want to overwrite ?",
          "File Doesn't Exists !!!",
          " Do you Want Try Another ?",
          "Answer Y or N or Q  ",
          "Answer Y or N  "
     };
     return((n < 1 || n > 7) ? string[0] : string[n]);
}


char *look_string(n)
int  n;
{
     static char    *string[] = {
          "ill strng",
          "Input Channel 0,1,2, or 3 (norm)"
     };
     return((n < 1 || n > 2) ? string[0] : string[n]);
}


char *stage_string(n)
int  n;
{
     static char    *string[] =      {
          "ill strng",
          "Input Total points in scan",
          "Input # of Stage Steps Between Points",
          "Input Shots per Step",
          "Input Stage position where Scan should begin",
          "Input Comment of 5 sentences or less and end with
a ~"
     };
     return((n < 1 || n > 5) ? string[0] : string[n]);
}
```

```
char *data_string(n)
int  n;
{
     static char    *string[] = {
          "ill strng",
          " 1: Change Comment",
          " 2: Set up Multi",
          " 3: Open Data Files",
          " 4: Set which channel to display",
          " 5: Take Data",
          " 6: Look at Data",
          " 7: Run Norm",
          " 8: Return to Main Menu"

     };
     return((n < 1 || n > 8) ? string[0] : string[n]);
}


char *set_string(n)
int  n;
{
     static char    *string[] = {
          "ill strng",
          "Points in scan",
          "Stage Steps Between points",
          "Shots per step ",
          "Stage Start",
          "Gate A width and delay",
          "Gate B width and delay",
          "Pedestal 0, 1 , and 2",
          "Comment",
          "Hit a key to return"
     };
     return((n < 1 || n > 9) ? string[0] : string[n]);
}


char *odd_string(n)
int  n;
{
     static char    *string[] = {
          "ill strng",
          "Trigger is now on internal",
          "Hit a key to continue",
          "Camac Error",
          "Ibfind error; does device or board",
          "name given match configuration name?",
          "GPIB function call error:",
          "Device error",
```

```
        "Graphics not supported, is msherc.com loaded if
herc?",
        "Input q to quit y for another",
        "Input number of files to add",
        "Input the first channel to be displayed",
        "either 0, 1, 2, or 3",
        "Input the second channel to be displayed",
        "Trigger checking is enabled now",
        "Trigger checking is not enabled now",
        "Input a 0 for trigger checking or a 1,",
        "for no trigger checking",
        "Camac is not giving a LAM",
        "DDG is now on internal",
        "Input a y to continue or a q to quit",
        "No LAM, DDG now on internal",
        "hit q to quit",
        "Input Low",
        "input High",
        "mode spe com",
        "mode spe auto"

    };
    return((n < 1 || n > 26) ? string[0] : string[n]);
}
```

H. PROGRAM LISTING ERROR

```
/*    Program "ERROR"
*     Daniel Russell
*     Aug. 6, 1990
*
*           The   file   Error.c   contains   the   following
subroutines. *
*           void  camerr()
*           void finderr()
*           void error()
*/

#include <graph.h>
#include <conio.h>
#include <stdio.h>
#include <decl.h>
#include "dan.h"

extern struct videoconfig vc;
extern struct rccoord rcoord;

/*          If there is an error in any of the camac routines
*     this subroutine will be called.  This is really only for
*     development purposes, although if any hardware problems
*     do occur this may help in tracking down the error.  This
*     will tell you which module was being talked to and what
*     parameters were passed.
*/

void  camerr(int mod, int fun, int A, unsigned int data, int
Q, int X)
{
      char buffer[100];
      int  i;
      _setvideomode(_DEFAULTMODE);
      _settextcolor(4);
      _settextposition(10, 15);
      rcoord = _gettextposition();
      _outtext("Camac Error");
      sprintf(buffer, "Module = %i", mod);
      rcoord.row++;
      _settextposition(rcoord.row, rcoord.col);
      _outtext(buffer);
      sprintf(buffer, "Function = %i", fun);
      rcoord.row++;
      _settextposition(rcoord.row, rcoord.col);
      _outtext(buffer);
      sprintf(buffer, "A = %i", A);
      rcoord.row++;
      _settextposition(rcoord.row, rcoord.col);
      _outtext(buffer);
      sprintf(buffer, "data = %ui", data);
```

```
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(buffer);
        sprintf(buffer, "Q = %i", Q);
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(buffer);
        sprintf(buffer, "X = %i", X);
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(buffer);
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext("Hit a key to continue");
        while (i == 0)
             i = kbhit();
        i = getche();
        return;
}

/*        This routine would notify you that the ibfind call
*    failed.
*/

void finderr()
{

        int  i;

        _setvideomode(_DEFAULTMODE);
        _settextcolor(4);
        _settextposition(10, 15);
        rcoord = _gettextposition();

        _outtext("Ibfind error; does device or board");
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);

        _outtext("name given match configuration name?");
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext("Hit a key to continue");
        while (i == 0)
             i = kbhit();
        i = getche();
        return;

}
/*        The error checking routine will, among other
things, *      check iberr to determine the exact cause of
the *          error condition and then take action
appropriate to *        the application.  For errors during
```

```
data transfers, *        ibcnt may be examined to determine
the actual number *      of bytes transferred.
*/


void error()
{
     char buffer[100];
     int  i;
     _setvideomode(_DEFAULTMODE);
     _settextcolor(4);
     _settextposition(10, 15);
     rcoord = _gettextposition();

     _outtext("GPIB function call error:");
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);

     sprintf(buffer,  "ibsta=0x%x,  iberr=0x%x,",  ibsta,
iberr);
     _outtext(buffer);
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);

     sprintf(buffer, " ibcnt=0x%x\n", ibcnt);
     _outtext(buffer);
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     _outtext("Hit a key to continue");
     while (i == 0)
          i = kbhit();
     i = getche();
     return;

}
```

I. PROGRAM LISTING PLOT5

```
/*   Program "PLOT5"
 *   Daniel Russell
 *   Aug. 6, 1990
 *
 *        The   file   Plot5.c   contains   the   following
subroutines.
 *
 *        void rbound1()
 *        void rtickxy1()
 *        void rpoint1()
 *        void text_point()
 *        void rbound2()
 *        void rtickxy2()
 *        void rpoint2()
 *        void csave()
 *        void rpointc()
 *        void cend()
 */

#include <graph.h>
#include <conio.h>
#include <stdio.h>
#include <malloc.h>
#include "dan.h"

extern struct videoconfig vc;
extern struct rccoord rcoord;
extern int     channel_dis1, channel_dis2;
extern int     channel_color1, channel_color2, box_color;

struct xycoord xycoord;
float     xa, xs1, ya, ys1;
float     xs2, ys2;
float     dx, dy;
float     xdiff1, ydiff1, xmin1, ymin1;
float     xdiff2, ydiff2, xmin2, ymin2;
float     xmin_tick1, tick_temp, xmax_tick, temp_tick;
float     ymin_tick1, ymax_tick1;
float     xmin_tick2, ymin_tick2, ymax_tick2;
long int  rect_xmin, rect_ymin, rect_xmax, rect_ymax;
float     rect_xdiff, rect_ydiff;

void rbound1(float a, float b, float xmax1, float ymax1)

/*        rbound1() sets up the mapping of data into the
 *        screens coordinates for the first set of data points
 *        the user wants to plot
 */

{
```

```
	extern float   xs1, ys1;
	extern float   xdiff1, ydiff1, xmin1, ymin1;
	xmin1 = a;
	ymin1 = b;
	xdiff1 = xmax1 - xmin1;
	ydiff1 = ymax1 - ymin1;
	rect_xdiff = (float)(rect_xmax - rect_xmin);
	rect_ydiff = (float)(rect_ymax - rect_ymin);
	xs1 = (rect_xdiff) / (xdiff1);
	ys1 = (rect_ydiff) / (ydiff1);
	rtickxy1();
}


void rtickxy1(void)
/*		rtickxy1() plots the tick marks for the first set of
*	data points
*/

{
	extern float   xdiff1, ydiff1, xmin1, ymin1;
	char buffer[10];
	extern float   xs1, ys1;
	extern long int     rect_xmin,  rect_ymin,  rect_xmax,
	                    rect_ymax;
	float     temp, temp2;
	temp = (xdiff1) / 50;
	temp = temp * 10;

	tick_temp = (((int)xmin1) / temp) * temp;
	if (xmin1 >= tick_temp)
		xmin_tick1 = tick_temp;
	else
		xmin_tick1 = tick_temp - temp;
	tick_temp = (((int)(xmin1 + xdiff1)) / temp) * temp;
	if ((xmin1 + xdiff1) <= tick_temp)
		xmax_tick = tick_temp;
	else
		xmax_tick = tick_temp + temp;
	temp_tick = xmin_tick1;
	xdiff1 = xmax_tick - xmin_tick1;
	xs1 = (rect_xdiff) / (xdiff1);
	xmin1 = xmin_tick1;
	_settextcolor(box_color);
	_setcolor(box_color);
	while (temp_tick <= xmax_tick) {
		temp2 = (temp_tick - xmin_tick1) * xs1 + rect_xmin;
		_moveto((int)temp2, (int) rect_ymax);
		_lineto((int)temp2, (int) rect_ymax - 10);
		text_point((long int)temp2, rect_ymax + 10);
```

```
            if (temp > (float)1.)
                sprintf(buffer, "%.0f", temp_tick);
            else if (temp < (float) 1.)
                sprintf(buffer, "%.1f", temp_tick);
            else if (temp > (float) .1)
                sprintf(buffer, "%.2f", temp_tick);

            _outtext(buffer);
            temp_tick = temp_tick + temp;
        }
        temp = (ydiff1) / 50;
        temp = (temp * 10);
        tick_temp = ((ymin1) / temp) * temp;
        if (ymin1 >= tick_temp)
            ymin_tick1 = tick_temp;
        else
            ymin_tick1 = tick_temp - temp;
        tick_temp = (((ymin1 + ydiff1)) / temp) * temp;
        if ((ymin1 + ydiff1) <= tick_temp)
            ymax_tick1 = tick_temp;
        else
            ymax_tick1 = tick_temp + temp;
        temp_tick = ymin_tick1;
        ydiff1 = ymax_tick1 - ymin_tick1;
        ys1 = (rect_ydiff) / (ydiff1);
        ymin1 = ymin_tick1;
        _setcolor(channel_color1);
        _settextcolor(channel_color1);
        while (temp_tick <= ymax_tick1) {
            temp2 = rect_ymax - (temp_tick - ymin_tick1) * ys1;
            _moveto((int)rect_xmin, (int)temp2);
            _lineto((int)rect_xmin + 10, (int)temp2);
            text_point(rect_xmin - 30, (long int)temp2);

            if (temp >  (float)1)
                sprintf(buffer, "%.0f", temp_tick);
            else if (temp > (float) .1)
                sprintf(buffer, "%.1f", temp_tick);
            else if (temp > (float) .01)
                sprintf(buffer, "%.2f", temp_tick);
            else if (temp < (float) .01)
                sprintf(buffer, "%.3f", temp_tick);


            _outtext(buffer);
            temp_tick = temp_tick + temp;
        }
    }


    void rpoint1(float rx0, float ry0)
```

```
/*        rpoint1() plots one point per call using rbound1()'s
*         mapping to map onto screen coordinates
*/

{
     int  ix, iy;
     ix = (int)((rx0 - xmin_tick1) * xs1 + rect_xmin);
     iy = (int)(rect_ymax - (ry0 - ymin_tick1) * ys1);
     _setcolor(channel_color1);
     _setpixel(ix, iy);
}


void text_point(ix, iy)

/*        text_point() moves the text coordinate to where the
*    label for the tick marks should be
*/

long int  ix, iy;
{

     xycoord = _getphyscoord((short)ix, (short) iy);
     ix    =    (long    int)(    (float)xycoord.xcoord    /
          (float)vc.numxpixels * ((float)vc.numtextcols));
     iy   =   (long   int)(   1   +   (float)xycoord.ycoord   /
          (float)vc.numypixels * ((float)vc.numtextrows));
     _settextposition((short)iy, (short)ix);
}


void rbound2(float b, float ymax2)

/*        rbound2()  is  the  same  as  rbound1()  except  the
*    mapping is for the second set of data points.  Note that
*    the xaxis mapping is already defined from rbound1()
*/

{
     extern float   xs2, ys2;
     extern float   xdiff2, ydiff2, xmin2, ymin2;

     ymin2 = b;
     xs2 = (rect_xdiff) / (xdiff1);
     ydiff2 = ymax2 - ymin2;

     ys2 = (rect_ydiff) / (ydiff2);
     rtickxy2();
}


void rtickxy2(void)
```

```
/*      rtickxy2() is the same as rtickxy1() except the
 *   xaxis tick marks have already been drawn by rtickxy1()
 */

{
     extern float   xdiff2, ydiff2, xmin2, ymin2;
     char buffer[10];
     extern float   xs2, ys2;
     extern long int     rect_xmin,  rect_ymin,  rect_xmax,
                         rect_ymax;
     float     temp, temp2;

     temp = (ydiff2) / 50;
     temp = (temp * 10);
     tick_temp = ((ymin2) / temp) * temp;
     if (ymin2 >= tick_temp)
          ymin_tick2 = tick_temp;
     else
          ymin_tick2 = tick_temp - temp;
     tick_temp = (((ymin2 + ydiff2)) / temp) * temp;
     if ((ymin2 + ydiff2) <= tick_temp)
          ymax_tick2 = tick_temp;
     else
          ymax_tick2 = tick_temp + temp;
     temp_tick = ymin_tick2;
     ydiff2 = ymax_tick2 - ymin_tick2;
     ys2 = (rect_ydiff) / (ydiff2);
     _setcolor(channel_color2);
     _settextcolor(channel_color2);
     ymin2 = ymin_tick2;
     while (temp_tick <= ymax_tick2) {


          temp2 = rect_ymax - (temp_tick - ymin_tick2) * ys2;
          _moveto((short)rect_xmax, (short)temp2);
          _lineto((short)(rect_xmax - 10), (short)temp2);
          text_point(rect_xmax + 20, (long int)temp2);


          if (temp > (float) 1)
               sprintf(buffer, "%.0f", temp_tick);
          else if (temp > (float) .1)
               sprintf(buffer, "%.1f", temp_tick);
          else if (temp > (float) .01)
               sprintf(buffer, "%.2f", temp_tick);
          else if (temp < (float) .01)
               sprintf(buffer, "%.3f", temp_tick);


          _outtext(buffer);
          temp_tick = temp_tick + temp;
     }
```

```
}


void rpoint2(float rx0, float ry0)

/*       rpoint2() is the same as rpoint1() except it is
 *    mapped to the screen coordinates by rbound2()
 */

{
     int  ix, iy;
     ix = (int)((rx0 - xmin_tick1) * xs2 + rect_xmin);
     iy = (int)(rect_ymax - (ry0 - ymin_tick2) * ys2);
     _setcolor(channel_color2);
     _setpixel(ix, iy);
}


int  ixold, iyold;
static char    far *cimage;
void csave(float rx0, float ry0)

/*       csave() initializes cursor movements by saving space
 *    for the original image and then writes the cursor image.
 *    There is a better and quicker way to do this with xor but
 *    I did not realize it at the time
 */

{

     extern char    far *cimage;
     int  ix, iy, i;
     extern int     ixold, iyold;
     ix = (int)((rx0 - xmin_tick1) * xs1 + rect_xmin);
     iy = (int)(rect_ymax - (ry0 - ymin_tick1) * ys1);
     ixold = ix;
     iyold = iy;
     cimage = _fmalloc((unsigned int) _imagesize(ix - 5, iy -
                    5, ix + 5, iy + 5));
     if (cimage == (char far * ) NULL) {
          printf("calloc failed");

          printf("hit a key to continue");
          while (i == 0)
               i = kbhit();
          i = getche();

          return;
     }

     _getimage(ix - 5, iy - 5, ix + 5, iy + 5, cimage);
     _setlinestyle(0xaaaa);
```

```
    _rectangle(_GFILLINTERIOR, 400, 330, 600, 360);
}


void rpointc(float rx0, float ry0)

/*        This plots the cursor at the new position
*/

{
    extern char    far *cimage;
    extern int     ixold, iyold;
    int  ix, iy;
    char buffer[20];
    _putimage(ixold - 5, iyold - 5, cimage, _GPSET);
    ix = (int)( (rx0 - xmin_tick1) * xs1 + rect_xmin);
    iy = (int)(rect_ymax - (ry0 - ymin_tick1) * ys1);

    _getimage(ix - 5, iy - 5, ix + 5, iy + 5, cimage);
    _setcolor(15);
    _moveto(ix, iy + 5);

    _lineto(ix, iy - 5);
    _moveto(ix, iy + 5);
    _lineto(ix, iy - 5);
    ixold = ix;
    iyold = iy;
    _settextposition(25, 55);
    _remappalette(6, _BLACK);
    _setcolor(6);
    sprintf(buffer, "%i %f", (int)rx0, ry0);
    _rectangle(_GFILLINTERIOR, 400, 330, 600, 360);
    _settextcolor(box_color);
    _outtext(buffer);
}


void cend()
/*        cend() frees up the space used for cursor movements
*/
{
    extern char    far *cimage;
    _ffree(cimage);
}
```

J. PROGRAM LISTING COMM

```
/*    Program "COMM"
 *    Daniel Russell
 *    Aug. 6, 1990
 *
 *          The file Comm.c contains the following subroutines.
 *
 *          gp_talk()
 *          se_talk()
 *          int  check()
 */

#include <decl.h>
#include <graph.h>
#include <conio.h>
#include <stdio.h>
#include "dan.h"

extern struct videoconfig vc;
extern struct rccoord rcoord;

/*          There are two ways to communicate to the stage.
 *    Using GPIB-PC, or serial line.  gp_talk sends the string
 *    to the GPIB-PC and returns error info.
 */

gp_talk(device, data, count)
int  device;
char data[];
int  count;
{
     extern int     ibsta, iberr, ibcnt;
     ibwrt(device, data, count);
     return (ibsta & ERR);
}

/*          se_talk is used to communicate through the serial
 *    line I am not sure if this code was ever tested and
 *    therefore I have commented it out.  Right now this is not
 *    a problem since all communication occurs through the
 *    GPIB-PC.
 */

se_talk(dummy, data, dummy2)
int  dummy, dummy2;
char data[];

{
     FILE * stream;
     int  sent;
```

```
     /*stream=fopen("com1","wb");
sent=fprintf(stream,"%s",data);
if(sent!=dummy2||fclose(stream)!=0){


     return(-1);}
else


*/
     return(0);

}

/*        check is a program that is used to check if the
*    digital delay generator is being triggered.
*/

int  check()
{
     extern int     ddg;
     int  i, dummy = 0;
     for (i = 0; i != 5; i++) {

          ibwrt (ddg, "IS 1", 4);
/*        The command "IS 1" asks the digital delay generator
*    to return bit one of the status byte.  see pg 11 of the
*    manual.   If  rd[0]='0'  then  the  ddg  is  not  being
*    triggered.
*/

          ibrd (ddg, rd, 3) ;
          if ( rd[0] == '0')
               dummy++;
     }

     if ( dummy == 5) {
/*   If after 5 tries the ddg is not busy with a trigger, the
*    DDG is put on internal trigger mode
*/

          ibwrt(ddg, "rc 1", 4);
          _setvideomode(_DEFAULTMODE);
          _settextcolor(4);
          _settextposition(10, 15);
          rcoord = _gettextposition();
          _outtext("Trigger is now on internal");
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          _outtext("Hit a key to continue");
          while (i == 0)
               i = kbhit();
```

```
        i = getche();
        return(-1);
    }
    return(0);
}
```

K. PROGRAM LISTING DATA

```
/*    Program "DATA"
 *    Daniel Russell
 *    Aug. 6, 1990
 *
 *         The file Data.c contains the following subroutines.
 *
 *         void take_data_menu()
 *         int  test_num()
 */

#include <string.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <graph.h>
#include <conio.h>
#include "dan.h"
#include <setjmp.h>

extern double   *d_norm[4];
extern int      gate_chan_a, gate_chan_b;
extern char     comment[1000];
extern int      points_scan, stage_step;
extern long int     shots_step, stage_beg;
extern int      multi_count, reverse_flag, scan;
extern int      pedestal[3], multi_flag;
extern int      delay_a, delay_b,  gwidth_a, gwidth_b;
extern int      tcolor;
extern int      ecolor;
extern struct videoconfig vc;
extern struct rccoord rcoord;
extern int      ddg, trig_check;
extern jmp_buf mark;

/*         This section of code writes the take data menu and
 *    gets input.
 */

void take_data_menu(void)

{
     int  trig, sresult;
     int  count = 1;
     char *data_string(int)   ;
     char inputs[3];
     char *input;
     int  i = 0;
```

```
    _setvideomode(_DEFAULTMODE);
    _settextcolor(tcolor);
    _settextposition(10, 15);
    rcoord = _gettextposition();
    while (count != 8) {
         for (i = 1; i < 9; i++) {

/*       Write the menu on the screen
*/

              _outtext(data_string(i));
              rcoord.row++;
              _settextposition(rcoord.row, rcoord.col);
         }
         if (trig_check == 0) {
              trig = 0;
              sresult = system("mode spe com");
              while (kbhit() == 0 && trig == 0)

                   trig = check();
              ibloc(ddg);
              sresult = system("mode spe auto");
              if (trig == -1)
                   longjmp(mark, -1);
         }


         input = gets(inputs);
         count = atoi(inputs);
         if (count == 0 || count < 1 || count > 8) {

/*       If a valid key was not hit write error message
*/

              _setvideomode(_DEFAULTMODE);
              _settextposition(9, 15);
              rcoord = _gettextposition();
              _settextcolor(ecolor);
              _outtext(error_string(5));
              rcoord.row++;
              _settextposition(rcoord.row, rcoord.col);

              _settextcolor(tcolor);
         } else if (count == 1) {
              read_comment();
              _setvideomode(_DEFAULTMODE);
              _settextposition(10, 15);

              _settextcolor(tcolor);

              rcoord = _gettextposition();
         } else if (count == 2) {
```

```
                set_multi();
                _setvideomode(_DEFAULTMODE);
                _settextcolor(tcolor);
                _settextposition(10, 15);
                rcoord = _gettextposition();
            } else if (count == 3) {
                open_data_file();
                _setvideomode(_DEFAULTMODE);
                _settextcolor(tcolor);
                _settextposition(10, 15);
                rcoord = _gettextposition();
            } else if (count == 4) {
                channel_display();
                _setvideomode(_DEFAULTMODE);
                _settextposition(10, 15);
                _settextcolor(tcolor);
                rcoord = _gettextposition();
            } else if (count == 5) {
                take_data();
                _setvideomode(_DEFAULTMODE);
                _settextcolor(tcolor);
                _settextposition(10, 15);
                rcoord = _gettextposition();
            } else if (count == 6) {

/*      In order to look at another file data space must be
 *   free for the new data.  Note this new data file might not
 *   have the same number of points per scan as the current
 *   setup file has defined, therefore even though points_scan
 *   is defined in look_data() it is not the same variable as
 *   points_scan in the rest of the program.  This allows the
 *   user to look at different data files without losing their
 *   current setup.
*/

                if (d_norm[0] != NULL)
                    free (d_norm[0]);
                if (d_norm[1] != NULL)
                    free (d_norm[1]);
                if (d_norm[2] != NULL)
                    free (d_norm[2]);
                if (d_norm[3] != NULL)
                    free (d_norm[3]);
                look_data();
                if (points_scan != 0 ) {
                    d_norm[0] = (double *)calloc(points_scan,
                                 sizeof(double));
                    d_norm[1] = (double *)calloc(points_scan,
                                 sizeof(double));
                    d_norm[2] = (double *)calloc(points_scan,
                                 sizeof(double));
                    d_norm[3] = (double *)calloc(points_scan,
```

```
                                sizeof(double));
                }
                _setvideomode(_DEFAULTMODE);
                _settextposition(10, 15);
                _settextcolor(tcolor);
                rcoord = _gettextposition();
            } else if (count == 7) {
                norm();
                _setvideomode(_DEFAULTMODE);
                _settextposition(10, 15);
                _settextcolor(tcolor);
                rcoord = _gettextposition();
            }


        }
        _setvideomode(_DEFAULTMODE);
        return;
}



/*      test_num is a routine I wrote to make sure the
 *      string returned by the user was all numbers.  There are
 *      easier ways to do this but I never changed the parts of
 *      the program where this is used. One could for example do
 *      the same thing with atoi() as was done in the subroutine
 *      take_data_menu()
 */

int     test_num(nums)

char nums[];
{
        int  i = 0, ch;
        while (nums[i] != '\0') {
            ch = nums[i];
            if (isdigit(ch) == 0) {
                if (islower(ch))
                    return(-1);
                else if (isupper(ch))
                    return(-1);
                else
                    i++;
            } else
                i++;
        }
        return(0);
}
```

L. PROGRAM LISTING LOOK

```
/*    Program "LOOK"
 *    Daniel Russell
 *    Aug. 6, 1990
 *
 *          The file Look.c contains the following subroutines.
 *
 *          void look_data()
 *          void plottop()
 *          int  plot()
 *          void edit()
 *          void clesave()
 */

#include <malloc.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <graph.h>
#include <conio.h>
#include <bios.h>
#include <math.h>
#include "dan.h"
#include <string.h>
#include <setjmp.h>

/*          Note the use of static variables in this file.  This
 *    is to allow the user to look at data with different setup
 *    parameters without affecting the setup parameters used in
 *    taking data setup parameters
 */

static int     points_scan, stage_step;
static int     multi_count, reverse_flag, look, scan;
static long int     shots_step, stage_beg;
static double  *d_norm[4];
static char    file_name[7], file_num[3], data_file[12];
static int     pedestal[3], multi_flag;
static int     gate_chan_a, gate_chan_b, point_scan_tak;
static char    comment[1000];

struct videoconfig vc;
struct rccoord rcoord;
extern int     tcolor;
extern int     ecolor, box_color;
extern long int     rect_xmin,    rect_ymin,    rect_xmax,
rect_ymax;
time_t ltime;

void look_data(void)
```

```
{
     int  trig, sresult;
     int  i, chan;
     int  numwritten, ch;
     FILE * stream;
     char inputs2[2], test[4];
     char *input, *input2;

     inputs2[0] = 'y';
     stream = NULL;

     _getvideoconfig(&vc);
     while  ((inputs2[0]  ==  'y'  ||  inputs2[0]  ==  'Y'  ||
            inputs2[0] == 'c' || inputs2[0] == 'C') ) {

          _setvideomode(_DEFAULTMODE);
          _settextcolor(tcolor);
          _settextposition(10, 15);
          rcoord = _gettextposition();
          if (inputs2[0] == 'y' || inputs2[0] == 'Y') {
               _outtext(multi_string(7));
               rcoord.row++;
               _settextposition(rcoord.row, rcoord.col);


               input = gets(file_name);
          }
          strcpy(data_file, file_name);
          strcat(data_file, ".");
          if (inputs2[0] == 'y' || inputs2[0] == 'Y') {
               _setvideomode(_DEFAULTMODE);
               _settextposition(10, 15);
               rcoord = _gettextposition();
               _outtext(multi_string(8));
               rcoord.row++;
               _settextposition(rcoord.row, rcoord.col);


               input = gets(file_num);
               strcat(data_file, file_num);
          }

/*        If C is hit the next data file with the same name
 *   but incremented by one number is read in.  For example if
 *   the user is looking at "dan.1", and  the  'C' or 'c' key
 *   is hit "dan.2" would be read in.
 */
          if (inputs2[0] == 'c' || inputs2[0] == 'C') {
               i = atoi(file_num);
               i++;
               itoa(i, file_num, 10);
               strcat(data_file, file_num);
```

```
}
if ((stream = fopen(data_file, "rb")) == NULL) {
     fclose(stream);
     rcoord.row++;
     _settextcolor(ecolor);
     _settextposition(rcoord.row, rcoord.col);
     _outtext( prompt_string(4));
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     _outtext( prompt_string(5));
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     _outtext( prompt_string(7));


     input2 = gets(inputs2);
}
rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext(data_file);
rcoord.row++;



if (stream != NULL) {
     for (i = 0; i != 3; i++) {
          ch = fgetc(stream);
          test[i] = (char) ch;
     }
     test[i] = '\0';
     if (strcmp(test, "djr") != 0)
          printf("not setup file");
     else {
          fscanf(stream, "%li", &ltime);
          fscanf(stream, "%i", &gate_chan_a);
          fscanf(stream, "%i", &gate_chan_b);
          fscanf(stream, "%i", &points_scan);
          fscanf(stream, "%i", &stage_step);
          fscanf(stream, "%li", &shots_step);
          fscanf(stream, "%li", &stage_beg);
          fscanf(stream, "%i", &pedestal[0]);
          fscanf(stream, "%i", &pedestal[1]);
          fscanf(stream, "%i", &pedestal[2]);
          fscanf(stream, "%i", &reverse_flag);
          fscanf(stream, "%i", &multi_count);
          fscanf(stream, "%i", &scan);
          for  (i  =  0;  (i  <  999)  &&  (ch  =
               fgetc(stream)) != '~'; i++)
               comment[i] = (char) ch;
          comment[i] = (char) ch;
          i++;
          ch = fgetc(stream);
```

```
comment[i] = (char) ch;
if (d_norm[0] != NULL)
     free (d_norm[0]);
if (d_norm[1] != NULL)
     free (d_norm[1]);
if (d_norm[2] != NULL)
     free (d_norm[2]);
if (d_norm[3] != NULL)
     free (d_norm[3]);

d_norm[0] = (double *)calloc(points_scan,
                sizeof(double));
d_norm[1] = (double *)calloc(points_scan,
                sizeof(double));
d_norm[2] = (double *)calloc(points_scan,
                sizeof(double));
d_norm[3] = (double *)calloc(points_scan,
                sizeof(double));

if (d_norm[0] == NULL || d_norm[1] == NULL
    || d_norm[2] == NULL || d_norm[3] ==
    NULL) {
     if (d_norm[0] != NULL)
          free (d_norm[0]);
     if (d_norm[1] != NULL)
          free (d_norm[1]);
     if (d_norm[2] != NULL)
          free (d_norm[2]);
     if (d_norm[3] != NULL)
          free (d_norm[3]);
     d_norm[0]    =    (double    *)
                  calloc(points_scan,
                  sizeof(double));
     d_norm[1]    =    (double    *)
                  calloc(points_scan,
                  sizeof(double));
     d_norm[2]    =    (double    *)
                  calloc(points_scan,
                  sizeof(double));
     d_norm[3]    =    (double    *)
                  calloc(points_scan,
                  sizeof(double));
}
for (i = 0; i < 4; i++) {
     if (d_norm[i] == NULL) {

          printf("Allocation Failure \n");
          printf("hit a key to continue");
          while (i == 0)
               i = kbhit();
          i = getche();
```

```
                        return;
                    }
                }

            for (i = 0; i != 4; i++)
                    numwritten    =    fread(d_norm[i],
                            sizeof(double),
                            points_scan, stream);

                fclose(stream);

            }
            if (inputs2[0] == 'y' || inputs2[0] == 'Y') {
                _setvideomode(_DEFAULTMODE);
                _settextposition(10, 15);
                _settextcolor(tcolor);
                rcoord = _gettextposition();


                _outtext(look_string(1));
                rcoord.row++;
                _settextposition(rcoord.row,rcoord.col);


                input2 = gets(inputs2);
                chan = atoi(inputs2);
            }

/*      plot() returns an integer that is to be used for
 *  displaying another channel or saving a file after a
 *  change to one of the data points
 */

            inputs2[0] = (char)( plot(chan) & 0x0FF);
            while (inputs2[0] == '0' || inputs2[0] == '1'
                || inputs2[0] == '2' || inputs2[0] == '3')
                inputs2[0] = (char)(plot(atoi(inputs2)) &
                0x0FF);
            if (inputs2[0] == 's')

/*      clesave() is a routine to save the data changed by
 *  the user.  It is a separate subroutine from save in order
 *  to keep the setup variables different from the variables
 *  in the file the user is looking at.
 */

                clesave(0);

        }
    }
    _setvideomode(_DEFAULTMODE);
    if (d_norm[0] != NULL)
```

```
            free (d_norm[0]);
        if (d_norm[1] != NULL)
            free (d_norm[1]);
        if (d_norm[2] != NULL)
            free (d_norm[2]);
        if (d_norm[3] != NULL)
            free (d_norm[3]);

/*          cend() frees space that was allocated for some of
 *      the plotting routines
 */
        cend();

}

/*          This plots the information that is above the window
 *      of data.  Note that this is a different subroutine than
 *      top() because the setup values should not be mixed
 *      between taking data and looking at data
 */

void plottop(n)
int  n;
{
        char *window_string(), *p, file_numt[3];
        int  num1, i;

        strcpy(data_file, file_name);
        strcat(data_file, ".");
        num1 = atoi(file_num);
        num1 = num1 + n;
        p = itoa(num1, file_numt, 10);
        strcat(data_file, file_numt);

        if (_setvideomode(_ERESCOLOR))
            ;
        else if (_setvideomode(_HERCMONO))
            ;
        else {

/*          msherc.com came with Microsoft C 5.1 and must be
 *      loaded for graphics to work on a hercules monitor
 */

            _outtext("Graphics not supported, is msherc.com
                      loaded if herc");
            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            _outtext("Hit a key to continue");
            while (i == 0)
                i = kbhit();
            i = getche();
```

```
            return;
        }

        _getvideoconfig(&vc);
        _settextcolor(tcolor);
        _settextposition(1, 1);
        rcoord = _gettextposition();
        _outtext(window_string(1));
        _outtext(data_file);
        rcoord.col = rcoord.col + 25;
        _settextposition(rcoord.row, rcoord.col);

        time(&ltime);
        _outtext((ctime(&ltime)));
        _settextposition(2, 1);
        rcoord = _gettextposition();
        _outtext(window_string(3));
        printf("%li", shots_step);
        rcoord.col = rcoord.col + 20;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(window_string(4));
        printf("%li", stage_beg);
        rcoord.col = rcoord.col + 20;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(window_string(5));
        printf("%li", (long int) stage_beg + (long int)
               points_scan * (long int)stage_step);
        if (multi_count != 0) {
            _settextposition(3, 1);
            rcoord = _gettextposition();
            _outtext(window_string(6));
            rcoord.col = rcoord.col + 20;
        _settextposition(rcoord.row, rcoord.col);
            _outtext(window_string(7));
            printf("%i", scan);
            if (reverse_flag == 2)    {
                rcoord.col = rcoord.col + 20;
                _settextposition(rcoord.row, rcoord.col);
                _outtext(window_string(8));
            }
        }
    }


    int  plot(int n)

    {
        int  i, ch, j, imin, imax;
        float     max, min;

        ch = 'n';
        plottop(0);
```

```
    _getvideoconfig(&vc);
    _settextcolor(tcolor);
    _settextposition(25, 5);
    rcoord = _gettextposition();
    _outtext("Input q to quit, y for another, or s to save");

    rect_xmin = 50;
    rect_ymin = 50;
    rect_xmax = 550;
    rect_ymax = 300;
    _setcolor(box_color);
    _rectangle(_GBORDER,    (short)    rect_xmin,    (short)
               rect_ymin,   (short)    rect_xmax,    (short)
               rect_ymax);
    max = (float)*(d_norm[n]);
    min = (float) *(d_norm[n]);
    imin = 0;
    imax = 0;

/*       Find the max and min values of the data
*/

    for (i = 0; i < points_scan; i++) {
         if ( *(d_norm[n] + i) > max) {
              max = (float) *(d_norm[n] + i) ;
              imax = i;
         } else if ( *(d_norm[n] + i) < min) {
              min = (float) *(d_norm[n] + i);
              imin = i;
         }
    }

/*       Set bounds of plot window.
*/

    rbound1((float)1., min - (max - min) * .1,
            (float)points_scan, max + (max - min) * .1);
    for (i = 0; i < points_scan; i++)
         rpoint1((float)i + 1, (float)*(d_norm[n] + i));
    i = 0;
    j = 1;

/*       csave allocates some space for the plotting routines
 *  that is specifically needed for cursor movement.
*/

    csave((float)i + 1., (float)*(d_norm[n] + i));

/*       The next section of code moves the cursor around the
 *  screen, and allows editing of the data.
*/
```

```
ch = _bios_keybrd(_KEYBRD_READ);
while (ch != 4209 && ch != 5497 && ch != 0x0b30 && ch !=
0x0231 && ch != 0x0332 && ch != 0x0433 && ch != 8051 &&
    ch != 11875) {

     if (ch == 18176 && j != 1) {
          j = 1;
          i = 0;
          rpointc((float)j, (float)*(d_norm[n] + i));
     }
     if (ch == 20224 && j < points_scan) {
          j = points_scan;
          i = points_scan - 1;
          rpointc((float)j, (float)*(d_norm[n] + i));
     }
     if (ch == 18688 ) {
          j = imax + 1;
          i = imax;
          rpointc((float)j, (float)*(d_norm[n] + i));
     }
     if (ch == 20736 ) {
          j = imin + 1;
          i = imin;
          rpointc((float)j, (float)*(d_norm[n] + i));
     }
     if (ch == 18432 && j  < points_scan - 10) {
          j = j + 10;
          i = i + 10;
          rpointc((float)j, (float)*(d_norm[n] + i));
     }
     if (ch == 20480 && j  > 10) {
          j = j - 10;
          i = i - 10;
          rpointc((float)j, (float)*(d_norm[n] + i));
     }

     if (ch == 19200 && j != 1) {
          j--;
          i--;
          rpointc((float)j, (float)*(d_norm[n] + i));
     } else if (ch == 19712 && j < points_scan) {
          i++;
          j++;
          rpointc((float)j, (float)*(d_norm[n] + i));
     } else if (ch == 4709) {
          edit(i, n);
          cend();

/*        Must replot the data after editing.
*/

          plottop(0);
```

```
                _getvideoconfig(&vc);
                _settextcolor(tcolor);
                _settextposition(25, 5);
                rcoord = _gettextposition();
                _outtext("Input q to quit, y for another, or s
                         to save");

                _setcolor(box_color);
                _rectangle(_GBORDER, (short) rect_xmin, (short)
                           rect_ymin,    (short)    rect_xmax,
                           (short) rect_ymax);
                max = (float)*(d_norm[n]);
                min = (float) *(d_norm[n]);
                imin = 0;
                imax = 0;
                for (i = 0; i < points_scan; i++) {
                     if ( *(d_norm[n] + i) > max) {
                          max = (float) *(d_norm[n] + i) ;
                          imax = i;
                     } else if ( *(d_norm[n] + i) < min) {
                          min = (float) *(d_norm[n] + i);
                          imin = i;
                     }
                }
                rbound1((float)1., min - (max - min) * .1,
                        (float)points_scan, max + (max - min)
                        * .1);
                for (i = 0; i < points_scan; i++)
                     rpoint1((float)i + 1, (float) *(d_norm[n]
                             + i));
                i = 0;
                j = 1;
                csave((float)i + 1., (float)*(d_norm[n] + i));
          }
          ch = _bios_keybrd(_KEYBRD_READ);

     }
     return(ch);
}


void edit(int i, int n )
{

/*        There are three cases for editing.  If the data
*    point is the first or last data point there is only one
*    point adjacent and that becomes the new value of the
*    first or last data point.  Otherwise the new value is the
*    average of the two adjacent data points.
*/

     if (i == 0)
```

```
            *(d_norm[n]) = *(d_norm[n] + 1);
        else if (i == points_scan - 1)
            *(d_norm[n] + i) = *(d_norm[n] + i - 1);
        else
            *(d_norm[n]  +  i)  =  (*(d_norm[n]  +  i  -  1)  +
            *(d_norm[n] + i + 1)) / 2.;

}


void clesave(n)
int  n;
{
     int  i;
     int  num1, numwritten;
     char *p, file_numt[3];
     FILE * stream;

     if (file_name[0] == '\0')
          return;
     stream = fopen(data_file, "wb");
     fprintf(stream, "djr");
     fprintf(stream, "%li ", ltime);
     fprintf(stream, "%i %i ", gate_chan_a, gate_chan_b);
     fprintf(stream, "%i %i ", points_scan, stage_step);
     fprintf(stream, "%li %li ", shots_step, stage_beg);
     fprintf(stream, "%i %i %i ", pedestal[0], pedestal[1],
             pedestal[2]);
     fprintf(stream, "%i %i %i ", reverse_flag, multi_count,
             scan);
     fprintf(stream, " %s", comment);
     for (i = 0; i != 4; i++)
          numwritten  =  fwrite(d_norm[i],  sizeof(double),
                                  points_scan, stream);
     fclose(stream);

}
```

M. PROGRAM LISTING NICE

```
/*      Program "Nice"
 *      Daniel Russell
 *      Aug. 6,1990
 *
 *              The file Nice.c contains the following subroutines.
 *
 *              void screen()
 *              camerror()
 *              void stage_return()
 *              gp_stage_wait()
 *              se_stage_wait()
 */

/*              The code was written for an IBM PC clone with a
 *      80386 processor and 80837 coprocessor.  The intended use
 *      was both experiment control and data acquisition.  This
 *      section of code actually takes the data and plots it.
 *      The rest of code does all initialization, saving of new
 *      data and or plotting of old data, and some special
 *      functions.
 *                      Data is collected through a Camac based data
 *      acquisition system.  The actual data gathering software
 *      is assembler code that was modified code provided by DSP
 *      Technology, the producer of the CAMAC crate controller.
 *      This code is in the subroutine dani, which I will not
 *      provide, since it is in assembler.  I would like to
 *      describe it briefly though.  The code provided by DSP was
 *      not fast enough for our  intended purpose.  We wished to
 *      collect two channels of data a 8kHz, the repetition rate
 *      of the laser, and do four floating point operations.  the
 *      two channels of data are a signal channel(norm) and a
 *      reference channel that we wished to keep a running
 *      average of, and a running average of the ratio of the two
 *      which is the signal due to exciting the molecule.  The
 *      code was modified in two basic ways.  It was modified to
 *      take advantage of 80826-80836 specific instructions that
 *      were faster.  Secondly the code was written two allow the
 *      floating point coprocesser actually be a coprocesser.
 *      Once a data point was collected, the floating point
 *      operations were started simultaneously on the 80837, with
 *      the next collection of data points by the 80836.  The two
 *      processes then ran concurrently.
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <graph.h>
#include <ctype.h>
```

```c
#include <malloc.h>
#include <dos.h>
#include <decl.h>
#include <process.h>
#include <malloc.h>
#include "dan.h"
#include <bios.h>
#include <conio.h>

/*      The following variables are used for plotting two of
 *   the three possible variables.  The norm or signal due to
 *   excitation can be plotted and or one of the other data
 *   points, the reference or the signal.  The reason for
 *   allowing plotting of two different pieces of data is to
 *   allow the user to see both the signal and to monitor the
 *   reference, which is directly related to laser power.
 */

float    max, min, max1, min1;

/*      The variables max, min, max1, and max1, are the
 *   maximum and minimum of the two data channels that are
 *   being displayed to allow appropriate scaling of the
 *   data in the data window.
 */

extern int    channel_dis1, channel_dis2;

/*      The  variables  channel_dis1  and  channel_dis2
 *   determine  which  of  the  three  data  variables  are
 *   displayed.
 */

extern int    channel_color1, channel_color2, box_color;

/*      The variables channel_color1, channel_color2, and
 *   box_color define the color of the two channels displayed
 *   and the color of the box.
 */

extern float  ymin_tick1, ymax_tick1, ymin_tick2, ymax_tick2;

/*      The variables ymin_tick1 etc are the real minimum
 *   and maximum values for the data window.  This is so that
 *   the tick marks on the xaxis and yaxis of the plot are
 *   reasonable and all the data points can be seen.
 */

extern long int    rect_xmin,    rect_ymin,    rect_xmax,
rect_ymax;
extern struct videoconfig vc;
extern struct rccoord rcoord;
```

```
/*        These variables are system dependent and are based
 *    the graphics coordinate system defined in the Microsoft
 *    C 5.1 graphics library.  The variables rect_xmin,
 *    rect_ymin rect_xmax, and rect_ymax are dependent on what
 *    type of video board, although the values in this program
 *    will work on a Hercules video card, EGA, and VGA although
 *    the VGA will be running in EGA mode.
 */


/*        The following variables are values for experimental
 *    conditions.
 */

int  stage_d;

/*        The variable stage_d keeps track of the direction
 *    the stage should move next.
 */

extern int     multi_count, reverse_flag, scan;

/*        These variables are initialized in the setup.
 *    multi_count is the number of times that the stage should
 *    be scanned, reverse_flag is 1 if data is taken in only
 *    one direction of stage movement and 2 if data is to be
 *    taken in both directions. Taking data in both directions
 *    is more efficient and will also help cancel any long term
 *    drift in time of the concentration of the molecule that
 *    you are studying if the drift is small.
 */

extern int     points_scan, stage_step;
extern long int     stage_beg;

/*        These variables describe the number of points the
 *    stage is going to scan ( basically the time base of the
 *    experiment), the number of stage steps between these
 *    points, and where the stage should begin.
 */

double    dtemp[4];

/*        This a temporary place to store data in the
 *    assembler code for the subroutine dani.
 */

extern double  *d_norm[4];

/*        This an array of pointers to arrays where the data
 *    is stored. The arrays are dynamically allocated earlier
 *    in the program. The number of arrays is 4 due to the
```

```
*    fact that at a later point in time someone may want to
*    also collect the actual laser power as a function of
*    time.
*/

extern int     ibsta, iberr, ibcnt;
int  yaxis;
int  se_talk(),  gp_talk(),  (*talk)(),  (*v_stage_wait)(),
gp_stage_wait(), se_stage_wait();
unsigned int   set_up , return1;

/*        The  above  variables  are  associated  with
*    communication subroutines.  ibsta, iberr, ibcnt are the
*    GPIB-PC status, error, and count variables.  If ibsta &
*    ERR are true an error has occurred and the user is
*    notified of the error and what type of error it was.
*    yaxis is the variable the GPIB-PC software associates
*    with the Klinger stage.  talk is a pointer to a function
*    that is defined at run time depending on whether or not
*    the GPIB-PC interface is running.  If the GPIB-PC
*    interface is running talk=gp_talk which is the subroutine
*    provided with the GPIB-PC card to talk to the GPIB-PC,
*    other wise talk=se_talk which is subroutine to talk to
*    the RS-232 port.  The same rules apply to v_stage_wait,
*    which is the pointer to the function that will handle
*    waiting for the stage to finish moving.  set_up and
*    return are the initial setup values for the RS-232 port
*    and a return variable to check for errors.
*/


void screen(void)
{
    char *stage_direction;
    unsigned int   step1, step2, step3, step4;

/*        These are temporary variables to hold partial stage
*    movements in case the desired stage movement is a larger
*    number than a 16 bit integer can hold.  This is a
*    constraint due to the stage controller.
*/

    int  stage_beg_l,  stage_total_l, stage_step_length;

/*        These variables are the length of character strings
*    that are sent to the stage controller
*/

    char inputs[3],stage_s[10];
    char stage_total_s[10],stage_beg_s[10];

/*   space for various strings used in communication
```

```
*/

     int  point_scan_tak, point_scan_tak2, i;
     int  data_space[17], e = 0, flag;

/*        point_scan_tak and point_scan_tak2 are the number of
 *   points in the scan that have been taken, although
 *   point_scan_tak starts counting at 1 and point_scan_tak2
 *   starts at 0. i is dummy counter  for a loop, and
 *   data_space is more temporary space for the assembler
 *   code. e is an error flag from the assembler code. flag is
 *   a variable to find out if there is a data point out of
 *   the window bounds.
*/

     long int  vector, stage_total;

/*        Unfortunately Microsoft used an interrupt vector
 *   that was reserved by intel for a later instruction on the
 *   80826  and 80836 which checked to see if the number in
 *   register ax was within the bounds set by a user.  If it
 *   was not within the bound, interrupt 5 occurred.  vector
 *   is a variable that stores the address for microsoft's
 *   interrupt routine so that it can be restored when the
 *   program exits. stage_total is the variable that stores
 *   the total number of stage steps in a scan.
*/

     char *input,  *go;

     _getvideoconfig(&vc);
     go = "G\r";

     yaxis = ibfind ("YAXIS");
     talk = gp_talk;

/*        Check to see if GPIB-PC is operating
*/

     if ((*talk)(yaxis, "+W\r", 3) < 0) {
          set_up = 0;
          set_up = set_up | _COM_NOPARITY;
          set_up = set_up | _COM_STOP1;
          set_up = set_up | _COM_CHR8;
          set_up = set_up | _COM_9600;
          return1 = _bios_serialcom(_COM_INIT, 0, set_up);
          return1 = _bios_serialcom(_COM_STATUS, 0, 0);
          talk = se_talk;
          v_stage_wait = se_stage_wait;
          if (return1 < 0) {
               finderr();
```

```
                return;
            }
        } else {
            talk = gp_talk;
            v_stage_wait = gp_stage_wait;
        }

/*          Initialize the stage.
*/

        if ((*talk)(yaxis, "FS01\r", 5) < 0) {
            error();
            return;
        }
        if ((*talk)(yaxis, "RW4000\r", 7) < 0) {
            error();
            return;
        }
        if ((*talk)(yaxis, "AC.1\r", 5) < 0) {
            error();
            return;
        }

        if (stage_beg > 0) {
            if (stage_d == 1)
                stage_direction = "+W\r";
            stage_d = -1;
        }

          else {
            stage_direction = "-W\r";
            stage_d = 1;
        }
        if ((*talk)(yaxis, stage_direction, 3) < 0) {
            error();
            return;
        }
        if (stage_beg < 65000 && stage_beg > -65000) {
            stage_beg_l   =   sprintf(stage_beg_s,   "NW%li\r",
                                 labs(stage_beg));
            (*talk)(yaxis, stage_beg_s, stage_beg_l);
            if ((*talk)(yaxis, "MW\r", 3) < 0) {
                error();
                return;
            }

        } else {
            step1 = (stage_beg) / 4;
            step2 = (stage_beg - step1) / 3;
            step3 = (stage_beg - step1 - step2) / 2;
            step4 = (stage_beg - step1 - step2 - step3);
```

```
        stage_step_length  =  sprintf(stage_s,  "NW%u\r",
                                    step1);


        (*talk)(yaxis, stage_s, stage_step_length);
        (*talk)(yaxis, "MW\r", 3);
        if (ibsta & ERR) {
             error();
             return;
        }

        v_stage_wait();
        stage_step_length  =  sprintf(stage_s,  "NW%u\r",
                                    step2);
        (*talk)(yaxis, stage_s, stage_step_length);
        if ((*talk)(yaxis, "MW\r", 3) < 0) {
             error();
             return;
        }

        v_stage_wait();
        stage_step_length  =  sprintf(stage_s,  "NW%u\r",
                                    step3);
        (*talk)(yaxis, stage_s, stage_step_length);
        if ((*talk)(yaxis, "MW\r", 3) < 0) {
             error();
             return;
        }

        v_stage_wait();
        stage_step_length  =  sprintf(stage_s,  "NW%u\r",
                                    step4);
        (*talk)(yaxis, stage_s, stage_step_length);
        if ((*talk)(yaxis, "MW\r", 3) < 0) {
             error();
             return;
        }
   }
   v_stage_wait();

   if (stage_step > 0) {
        if (stage_d == 1)
             stage_direction = "+W\r";
        stage_d = -1;
   }
       else {
        stage_direction = "-W\r";
        stage_d =  1;
   }
   stage_total    =    (long    int)stage_step    *    (long
                  int)(points_scan);
   stage_total_l   =   sprintf(stage_total_s,   "NW%li\r",
```

```
                                  stage_total);
     stage_step_length     =    sprintf(stage_s,    "NW%i\r",
                                    stage_step);


     if ((*talk)(yaxis, stage_s, stage_step_length) < 0) {
          error();
          return;
     }
     if ((*talk)(yaxis, stage_direction, 3) < 0) {
          error();
          return;
     }


     rect_xmin = 50;
     rect_ymin = 50;
     rect_xmax = 550;
     rect_ymax = 300;

/*   setbound gets Microsoft's interrupt 5 vector and newbound
 *   stores a new one to be used by the assembler code in data
 *   acquisition.
*/
     setbound(&vector);
     newbound();

/*   In the initialization program 0 is usually used as a flag
 *   for not in use so if multi_flag == 0 then there will only
 *   be one scan
*/
     if (multi_count == 0)
          multi_count = 1;

     scan = 1;
     while (scan <= multi_count) {
          inputs[0] = 'y';
          if (scan != 1 ) {
               if (reverse_flag == 1)
                    stage_d = -stage_d;
               if (stage_d == 1)     {
                    stage_d = -1;
                    stage_direction = "+W\r";
               } else {
                    stage_d = 1;
                    stage_direction = "-W\r";
               }


               if ((*talk)(yaxis, stage_direction, 3) < 0) {
                    error();
                    return;
```

```
            }

            if ((*talk)(yaxis, stage_s, stage_step_length)
                 < 0) {
                 error();
                 return;
            }

        }
        point_scan_tak = 1;

        point_scan_tak2 = 0;

/*      top is a graphics routine which labels the top half
 *   of the data window with info concerning the current data
 *   scan
 */

        top(scan - 1);

/*      the following graphics routines write the data
 *   window.
 */
        _remappalette(6, _BLACK);
        _setcolor(6);
        _rectangle(_GFILLINTERIOR, 0, 50, 620, 330);
        _setcolor(box_color);
        _rectangle(_GBORDER, (short) rect_xmin, (short)
                   rect_ymin, (short) rect_xmax, (short)
                   rect_ymax);

        while ( point_scan_tak <= points_scan && (inputs[0]
              == 'Y' || inputs[0] == 'y')) {
            v_stage_wait();
            dani(data_space, dtemp, &e);

            if (e != 0) {
                 if (camerror(e, point_scan_tak) == 1) {
                      oldbound(&vector);
                      return;
                 }

            }

/*   This next line moves the stage the number of steps
 *   defined by stage_step
 */
            (*talk)(yaxis, "MW\r", 3);
             {
                 error();
                 return;
            }
```

```
/*   This initializes the window bounds for the first point
 *   and makes assumptions on the minimum size of a norm
 *   signal to make a good guess for the first window
 */

                    if (point_scan_tak == 1)  {
                         max = ( float)dtemp[channel_dis1] + .01;
                         max1 = ( float)dtemp[channel_dis2] + .01;
                         min = (float) dtemp[channel_dis1] - .01;
                         min1 = (float) dtemp[channel_dis2] - .01;
                         rbound1((float)1.,min,(float)points_scan,
                                   max);
                         rbound2( min1, max1);
                    }

                    for (i = 0; i != 4; i++) {
                         *(d_norm[i] + point_scan_tak2) = dtemp[i];

                    }
                    flag = 0;
                    if ((float)dtemp[channel_dis1] > ymax_tick1) {
                         flag = 1;
                         max = (float) dtemp[channel_dis1];
                    }
                    if ((float)dtemp[channel_dis1] < ymin_tick1) {
                         flag = 1;
                         min = (float)dtemp[channel_dis1];
                    }
                    if ((float)dtemp[channel_dis2] > ymax_tick2) {
                         flag = 1;
                         max1 = (float)dtemp[channel_dis2];
                    }
                    if ((float)dtemp[channel_dis2] < ymin_tick2) {
                         flag = 1;
                         min1 = (float)dtemp[channel_dis2];
                    }

/*   If the new data point is out of the window bounds the
 *   window is redrawn and new bounds are defined
 */

                    if (flag == 1) {
                         _setcolor(6);
                         _rectangle(_GFILLINTERIOR, 0, 45, 620,
                                   330);
                         _setcolor(box_color);
                         _rectangle(_GBORDER, (short) rect_xmin,
                                   (short) rect_ymin, (short)
                                   rect_xmax, (short) rect_ymax);
                         rbound1((float)1., min, (float)
                                points_scan, max);
                         rbound2( min1, max1);
```

```
                    for (i = 1; i < point_scan_tak; i++) {
                        rpoint1((float)i,   (float)
                                *(d_norm[channel_dis1] + i));
                        rpoint2((float)i,   (float)
                                *(d_norm[channel_dis2] + i));
                    }
                }
                _setcolor(channel_color1);
                rpoint1((float)point_scan_tak,    (float)
                        dtemp[channel_dis1]);
                _setcolor(channel_color2);
                rpoint2((float)point_scan_tak,    (float)
                        dtemp[channel_dis2]);
                point_scan_tak++;
                point_scan_tak2++;

/*   The keyboard is checked and if it was hit, this allows
 *   the user to continue, quit, or quit while saving the data
 *   collected to this point.
 */

                if (kbhit() != 0) {
                    i = getche();
                    _setcolor(6);
                    _rectangle(_GFILLINTERIOR,  0,  30,  620,
                            330);
                    _settextposition(10, 15);
                    rcoord = _gettextposition();
                    _outtext(window_string(9));
                    rcoord.row++;
                    _settextposition(rcoord.row,rcoord.col);
                    _outtext(window_string(10));
                    rcoord.row++;

                    _settextposition(rcoord.row,rcoord.col);
                    _outtext(window_string(11));
                    rcoord.row++;
                    _settextposition(rcoord.row,rcoord.col);

                    input = gets(inputs);
                    if (inputs[0] == 'Q' || inputs[0] == 'q'){
                        stage_return((long   int)
                                point_scan_tak2* (long
                                int)stage_step);

                        stage_return((long int)stage_beg);

/*   oldbound returns interrupt 5 back to its initial
 *   microsoft state before this subroutine returns.
 */

                        oldbound(&vector);
```

```
                    return;
                }
                else if (inputs[0] == 'S' || inputs[0] ==
                        's'){
                    stage_return((long int)
                            point_scan_tak2 * (long
                            int) stage_step);

                    stage_return((long int)stage_beg);
                    open_data_file();
                    save(scan - 1,point_scan_tak2);

                    oldbound(&vector);
                    return;
                }
                else if (inputs[0] == 'Y' || inputs[0] ==
                        'y') {
                    _setcolor(6);
                    _rectangle(_GFILLINTERIOR,  0,  45,
                            620, 330);

                    _setcolor(box_color);
                    _rectangle(_GBORDER,    (short)
                            rect_xmin,   (short)
                            rect_ymin,   (short)
                            rect_xmax,   (short)
                            rect_ymax);
                    rbound1((float)1.,   min,   (float)
                        points_scan, max);
                    rbound2( min1,  max1);
                    for (i = 1;i < point_scan_tak;i++) {
                        rpoint1((float)i,   (float)
                            *(d_norm[channel_dis1]
                            + i));
                        rpoint2((float)i,   (float)
                            *(d_norm[channel_dis2]
                            + i));
                    }
                }

            }

        }
        save(scan - 1,point_scan_tak2);
        scan++;
        point_scan_tak--;
        point_scan_tak2--;

/*   The next section of code takes data backwards as the
 *   stage reverses
 */
        if (reverse_flag == 2) {
```

```
if (stage_d == 1)     {
     stage_d = -1;
     stage_direction = "+W\r";
} else {
     stage_d = 1;
     stage_direction = "-W\r";
}

v_stage_wait();
(*talk)(yaxis, stage_direction, 3);
if((*talk)(yaxis, "MW\r", 3)<0)
{
     error();
     return;
}
top(scan - 1);
_remappalette(6, _BLACK);
_setcolor(6);

_rectangle(_GFILLINTERIOR, 0, 45, 620, 330);

_setcolor(box_color);
_rectangle(_GBORDER, (short) rect_xmin, (short)
           rect_ymin,    (short)    rect_xmax,
           (short) rect_ymax);

while ( point_scan_tak >= 1 && (inputs[0] ==
        'Y' || inputs[0] == 'y')) {
     v_stage_wait();
     dani(data_space, dtemp, &e);
     if (e != 0){
          if(camerror(e, point_scan_tak)==1){
               oldbound(&vector);
               return;
          }

     }
     (*talk)(yaxis, "MW\r", 3);

     {
          error();
return;
                    }

     if (point_scan_tak == points_scan)  {
          max = (float)dtemp[channel_dis1] + .4;
          max1 = (float)dtemp[channel_dis2] + .01;
          min = (float)dtemp[channel_dis1] - .1;
          min1 = (float)dtemp[channel_dis2] - .01;
          rbound1((float)1.,    min,    (float)
                  points_scan, max);
          rbound2( min1, max1);
```

```
}
for (i = 0; i != 4; i++) {
     *(d_norm[i] + point_scan_tak2) = dtemp[i];

}
flag = 0;
if ((float)dtemp[channel_dis1] > ymax_tick1) {
     flag = 1;
     max = (float)dtemp[channel_dis1];
}
if ((float)dtemp[channel_dis1] < ymin_tick1) {
     flag = 1;
     min = (float)dtemp[channel_dis1];
}
if ((float)dtemp[channel_dis2] > ymax_tick2) {
     flag = 1;
     max1 = (float)dtemp[channel_dis2];
}
if ((float)dtemp[channel_dis2] < ymin_tick2) {
     flag = 1;
     min1 = (float)dtemp[channel_dis2];
}

if (flag == 1) {
     _setcolor(6);
     _rectangle(_GFILLINTERIOR, 0, 45, 620,
                330);
     _setcolor(box_color);
     _rectangle(_GBORDER, (short) rect_xmin,
                (short)  rect_ymin,  (short)
                rect_xmax, (short) rect_ymax);
     rbound1((float)1.,    min,    (float)
             points_scan, max);
     rbound2( min1,  max1);
     for(i=points_scan;i>point_scan_tak;i--){
          rpoint1((float)i,    (float)
                  *(d_norm[channel_dis1] + i));
          rpoint2((float)i,    (float)
                  *(d_norm[channel_dis2] + i));
     }
}
_setcolor(channel_color1);
rpoint1((float)point_scan_tak,    (float)
        dtemp[channel_dis1]);
_setcolor(channel_color2);
rpoint2((float)point_scan_tak,    (float)
        dtemp[channel_dis2]);
point_scan_tak--;
point_scan_tak2--;
if (kbhit() != 0) {
     i = getche();
     _setcolor(6);
```

```
_rectangle(_GFILLINTERIOR, 0, 30, 620,
           330);
_settextposition(10, 15);
rcoord = _gettextposition();
_outtext(window_string(9));
rcoord.row++;
_settextposition(rcoord.row,rcoord.col);
_outtext(window_string(10));
rcoord.row++;

_settextposition(rcoord.row,rcoord.col);
_outtext(window_string(11));
rcoord.row++;
_settextposition(rcoord.row,rcoord.col);
_outtext(window_string(12));
rcoord.row++;
_settextposition(rcoord.row,rcoord.col);

input = gets(inputs);
if (inputs[0]=='Q' || inputs[0] == 'q') {
     stage_return((longint)point_scan_tak
                  *(long int)stage_step);

     stage_return((long int)stage_beg);

     oldbound(&vector);
     return;
} else if (inputs[0] == 'S' || inputs[0]
           == 's') {
     stage_return((long   int)
                  point_scan_tak2 * (long
                  int) stage_step);

     stage_return((long int)stage_beg);
     open_data_file();
     save(scan - 1, point_scan_tak2);

     oldbound(&vector);
     return;
} else if (inputs[0] == 'Y' || inputs[0]
           == 'y') {
     _setcolor(6);
     _rectangle(_GFILLINTERIOR,  0,  45,
                620, 330);

     _setcolor(box_color);
     _rectangle(_GBORDER,    (short)
                rect_xmin,   (short)
                rect_ymin,   (short)
                rect_xmax,   (short)
                rect_ymax);
     rbound1((float)1.,   min,   (float)
```

```
                                points_scan, max);
                        rbound2( min1,  max1);
                        for  (i  =  points_scan;  i  >
                             point_scan_tak; i--) {
                             rpoint1((float)  i  ,(float)
                                     *(d_norm[channel_dis1]
                                     + i));
                             rpoint2((float)i,   (float)
                                     *(d_norm[channel_dis2]
                                     + i));
                        }
                   }

              }

         }
         v_stage_wait();
         if (stage_d == 1)     {
              (*talk)(yaxis, "+W\r", 3);
         } else {
              (*talk)(yaxis, "-W\r", 3);
         }

         if ((*talk)(yaxis, "MW\r", 3) < 0) {

              error();
              return;
         }
         save(scan - 1, points_scan - point_scan_tak2 - 1);
         scan++;
    }
    stage_return((long  int)point_scan_tak  *  (long  int)
              stage_step);
}
stage_return((long int)stage_beg);
oldbound(&vector);
}


camerror(e, point_scan_tak)

int  e, point_scan_tak;

{
    extern double  dtemp[4];
    char inputs[3];
    char *input;
    int  i, data_space[16];

    struct videoconfig vc;
    struct rccoord rcoord;
```

```
     _getvideoconfig(&vc);
     while (e == 5) {
          _setcolor(6);
          _rectangle(_GFILLINTERIOR, 0, 30, 620, 330);
          _settextposition(10, 15);
          rcoord = _gettextposition();

          _outtext("Camac is not giving a LAM");
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);

          _outtext("DDG is now on internal");
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);

          _outtext("Input a Y to continue or a q to quit");
          rcoord.row++;
          _settextposition(rcoord.row, rcoord.col);
          input = gets(inputs);
          if (inputs[0] == 'Q' || inputs[0] == 'q') {

/*        If the user wants to quit the stage should be
 *   returned to the place where it began.  I am not sure if
 *   this section of code is bug free in keeping track of
 *   which direction the stage should go.
 */

               stage_return((long int) point_scan_tak * (long
                         int )stage_step);

               stage_return((long int)stage_beg);



               return(1);
          } else if (inputs[0] == 'Y' || inputs[0] == 'y' &&
                    point_scan_tak == 1)

/*        Here data collection continues and there are two
 *   cases,  the data window doesn't need to redrawn because
 *   no data has been taken yet or the window must be redrawn
 *   because data already exists
 */

               dani(data_space, dtemp, &e);

          else if (inputs[0] == 'Y' || inputs[0] == 'y' &&
                    point_scan_tak != 1) {

               dani(data_space, dtemp, &e);
               _setcolor(6);
               _rectangle(_GFILLINTERIOR, 0, 45, 620, 330);
```

```
                    _setcolor(box_color);
                    _rectangle(_GBORDER, (short) rect_xmin, (short)
                           rect_ymin,    (short)    rect_xmax,
                           (short) rect_ymax);
                    rbound1((float)1.,  min,  (float)points_scan,
                         max);
                    rbound2( min1,  max1);
                    if (stage_d == 1)     {

                         for (i = points_scan; i > point_scan_tak;
                              i--) {
                              rpoint1((float)i, (float)*(d_norm[0]
                                   + i - 1));
                              rpoint2((float)i, (float)*(d_norm[3]
                                   + i - 1));
                         }
                    }
                    if (stage_d == -1) {
                         for (i = 1; i < point_scan_tak; i++) {
                              rpoint1((float)    i,    (float)
                                   *(d_norm[channel_dis1] + i -
                                   1));
                              rpoint2((float)    i,    (float)
                                   *(d_norm[channel_dis2] + i -
                                   1));
                         }
                    }
                    dani(data_space, dtemp, &e);

               }
          }

          return(0);
     }


     void stage_return(back)
     long int  back;
     {
          char stage_s[10], *go;
          int  stage_step_length;
          unsigned int   step1, step2, step3, step4;

          go = "G\r";

          v_stage_wait();
          if ((back) > 0) {
               (*talk)(yaxis, "-W\r", 3);
          }
              else {
               (*talk)(yaxis, "+W\r", 3);
          }
```

```
if (back < 65000) {
     stage_step_length  =  sprintf(stage_s,  "NW%li\r",
                                 labs( back));
     (*talk)(yaxis, stage_s, stage_step_length);
     (*talk)(yaxis, "MW\r", 3);
     v_stage_wait();
} else {
     step1 = (stage_beg) / 4;
     step2 = (stage_beg - step1) / 3;
     step3 = (stage_beg - step1 - step2) / 2;
     step4 = (stage_beg - step1 - step2 - step3);

     stage_step_length  =  sprintf(stage_s,  "N  %u\r",
                                 step1);


     (*talk)(yaxis, stage_s, stage_step_length);
     (*talk)(yaxis, "MW\r", 3);
     if (ibsta & ERR) {
          error();
          return;
     }

     v_stage_wait();
     if ((back) > 0) {
          (*talk)(yaxis, "-W\r", 3);
     }
        else {
          (*talk)(yaxis, "+W\r", 3);
     }


     stage_step_length  =  sprintf(stage_s,  "NW%u\r",
                                 step2);
     (*talk)(yaxis, stage_s, stage_step_length);
     if ((*talk)(yaxis, "MW\r", 3) < 0) {
          error();
          return;
     }

     v_stage_wait();

     if ((back) > 0) {
          (*talk)(yaxis, "-W\r", 3);
     }
        else {
          (*talk)(yaxis, "+W\r", 3);
     }


     stage_step_length  =  sprintf(stage_s,  "NW%u\r",
```

```
                                            step3);
                    (*talk)(yaxis, stage_s, stage_step_length);
                    if ((*talk)(yaxis, "MW\r", 3) < 0) {
                         error();
                         return;
                    }

                    v_stage_wait();
                    if ((back) > 0) {
                         (*talk)(yaxis, "-W\r", 3);
                    }
                        else {
                         (*talk)(yaxis, "+W\r", 3);
                    }


                    stage_step_length  =  sprintf(stage_s,  "NW%u\r",
                                            step4);
                    (*talk)(yaxis, stage_s, stage_step_length);
                    if ((*talk)(yaxis, "MW\r", 3) < 0) {
                         error();
                         return;
                    }

          }


}


gp_stage_wait()
{
     int  dummy = 0;
     /*                       (*talk)(yaxis, stage_direction,
                                        3);
               while ( ibsta & ERR) {
                    dummy++;
                    (*talk)(yaxis, stage_direction, 2);
                    dummy++;
               }*/
     char d[10];
     ibrd(yaxis, rd, 5);
     ibrd(yaxis, rd, 5);
     while ( rd[1] == 'A') {
          dummy++;
          ibrd(yaxis, rd, 5);
          dummy++;
     }
}


se_stage_wait()
```

```
{
     unsigned int   data;
     return1 = _bios_serialcom(_COM_STATUS, 0, data);
     data = 0;
     while (data != 16) {
          return1 = _bios_serialcom(_COM_STATUS, 0, data);
          data = return1 & 16;
     }
}
```

N. PROGRAM LISTING SAVE

```
/*   Program "Save"
*    Daniel Russell
*    Aug. 6,1990
*
*         The file Save.c contains the following subroutines.
*
*         void save()
*         double    sigma()
*         void norm()
*         void bound()
*
*/

#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <graph.h>
#include <conio.h>
#include <bios.h>
#include <math.h>
#include "dan.h"
#include <string.h>
#include <setjmp.h>

extern int     channel_dis1, channel_dis2;
time_t ltime;
extern int     tcolor;
extern int     ecolor;
extern int     channel_dis1, channel_dis2;
extern int     channel_color1, channel_color2, box_color;
extern struct videoconfig vc;
extern struct rccoord rcoord;
extern int     ddg, trig_check;
extern jmp_buf mark;

void save(n, point_scan_tak)

/*        Note that the variable point_scan_tak is passed to
*    save because the data collection may have been stopped
*    prematurely and points_scan may not be equal to the
*    number of data points that were actually taken.  n is the
*    number of scans taken in multi mode (start counting at 0
*    ).  This allows save to save the file with the right
*    final number.
*/

int  n, point_scan_tak;
{
     extern int     pedestal[3], multi_flag;
     extern int     gate_chan_a, gate_chan_b;
```

```
extern int     points_scan, stage_step;
extern int     multi_count, reverse_flag, scan;
extern long int     shots_step, stage_beg;
extern double  *d_norm[4];
extern char    file_name[7], file_num[3], data_file[12];
extern char    comment[1000];

int  i;
int  num1, numwritten;

char *p, file_numt[3];

FILE * stream;


if (file_name[0] == '\0')
     return;
strcpy(data_file, file_name);
strcat(data_file, ".");
num1 = atoi(file_num);
num1 = num1 + n;
p = itoa(num1, file_numt, 10);
strcat(data_file, file_numt);

stream = fopen(data_file, "rb");
if (stream != NULL) {
     _getvideoconfig(&vc);
     fclose(stream);
     _setvideomode(_DEFAULTMODE);
     _settextcolor(ecolor);
     _settextposition(10, 15);
     rcoord = _gettextposition();
     _outtext(data_file);
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     _outtext("File already exits");
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     _outtext("hit a key to continue");
     while (i == 0)
          i = kbhit();
     i = getche();
     open_data_file();
     stream = fopen(data_file, "wb");
}
fclose(stream);
stream = fopen(data_file, "wb");

fprintf(stream, "djr");
fprintf(stream, "%li ", ltime);
fprintf(stream, "%i %i ", gate_chan_a, gate_chan_b);
fprintf(stream, "%i %i ", point_scan_tak, stage_step);
```

```
     fprintf(stream, "%li %li ", shots_step, stage_beg);
     fprintf(stream, "%i %i %i ", pedestal[0], pedestal[1],
          pedestal[2]);
     fprintf(stream, "%i %i %i ", reverse_flag, multi_count,
          scan);
     fprintf(stream, " %s", comment);

     for (i = 0; i != 4; i++)
          numwritten  =  fwrite(d_norm[i],  sizeof(double),
                              point_scan_tak, stream);
     fclose(stream);

}


double    normtemp[8];
double    sigma(int n)

/*        This Subroutine calculates sigma for the three
*    different pieces of data, channel 0, channel 1, and norm.
*    normtemp[0] contains the sum of channel 0, and
*    normtemp[0+3] contains the sum of the squares of channel
*    3.  normtemp[1] contains the sum of channel 1 and
*    normtemp[2] contains the sum of norm.
*/

{
     extern long int    shots_step;
     extern double  normtemp[8];
     double    normtempn;
     normtempn = normtemp[n] * normtemp[n] /
               ((double)shots_step);
     normtempn = normtemp[n+3] - normtempn;
     normtempn = normtempn / ((double)(shots_step - 11));
     normtempn = sqrt(normtempn);
     return(normtempn);
}


void norm(void)

{
     char inputs2[2];
     char *input2;
     int  trig, sresult;
     extern long int    shots_step;
     int  j,  e = 0 , ddg, i;
     long int  vector;

     unsigned int   DATA[16];
     char buffer[100];
```

```
    extern double  normtemp[8];
    double    ntemp;

    setbound(&vector);

/*       Note that there is a different subroutine to change
 *  interrupt 5 for norm.  nnewbound is for norm, and
 *  newbound is for regular data collection.
*/

    nnewbound();
    inputs2[0] = 'y';
    while (inputs2[0] != 'q') {

         ndani(DATA, normtemp, &e);
         if (e == 5) {
              /*ddg=ibfind("ddg");
         ibwrt(ddg,"tm 0",4);
         ibloc(ddg);*/
              _setvideomode(_DEFAULTMODE);
              _settextposition(10, 15);
              rcoord = _gettextposition();
              _outtext("No LAM, DDG now on internal");
              rcoord.row++;
              _settextposition(rcoord.row, rcoord.col);
              _outtext("hit a key to continue");
              while (i == 0)
                   i = kbhit();
              i = getche();
              break;
         }

         _setvideomode(_DEFAULTMODE);
         _settextcolor(ecolor);
         _settextposition(10, 15);
         rcoord = _gettextposition();
         _outtext("hit q to quit");
         rcoord.row++;
         _settextposition(rcoord.row, rcoord.col);


         sprintf(buffer, "average of  ");
         _outtext(buffer);
         rcoord.row++;
         _settextposition(rcoord.row, rcoord.col);
         sprintf(buffer, "Channel 0  %5.2f ", normtemp[0] /
                 (double)shots_step);
         _outtext(buffer);
         rcoord.row++;
         _settextposition(rcoord.row, rcoord.col);
         sprintf(buffer, "Channel 1  %5.2f ", normtemp[1] /
                 (double)shots_step);
```

```
                _outtext(buffer);
                rcoord.row++;
                _settextposition(rcoord.row, rcoord.col);
                sprintf(buffer, "Norm    %le ", normtemp[2] /
                        (double)shots_step);
                _outtext(buffer);
                _settextposition(11, 45);
                rcoord = _gettextposition();
                ntemp = sigma(0);
                j = sprintf(buffer, "SD of ");
                _outtext(buffer);
                rcoord.row++;
                _settextposition(rcoord.row, rcoord.col);
                j = sprintf(buffer, "%E ", ntemp);
                _outtext(buffer);
                ntemp = sigma(1);
                rcoord.row++;
                _settextposition(rcoord.row, rcoord.col);
                j = sprintf(buffer, "%E ", ntemp);
                _outtext(buffer);
                ntemp = sigma(2);
                rcoord.row++;
                _settextposition(rcoord.row, rcoord.col);
                j = sprintf(buffer, "%.4f ", ntemp * 100.);
                _outtext(buffer);
                j = sprintf(buffer, "%% ");
                _outtext(buffer);
                rcoord.row++;
                _settextposition(rcoord.row, rcoord.col);
                if (trig_check == 0) {
                     trig = 0;
                     sresult = system("mode spe com");
                     while (kbhit() == 0 && trig == 0)

                          trig = check();
                     ibloc(ddg);
                     sresult = system("mode spe auto");
                     if (trig == -1)
                          longjmp(mark, -1);
                }


                inputs2[0] = getch();

          }
          oldbound(&vector);
     }


void bound(int n)
{
     struct videoconfig vc;
```

```
struct rccoord rcoord;
char inputs2[10];
char *input2;
int  trig, sresult;
extern int     low, high;
high = 0;
low = 0;
while (low == 0) {
     _setvideomode(_DEFAULTMODE);
     _settextcolor(ecolor);
     _settextposition(10, 15);
     rcoord = _gettextposition();
     _outtext("Input Low");
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     if (trig_check == 0) {
          trig = 0;
          sresult = system("mode spe com");
          while (kbhit() == 0 && trig == 0)

               trig = check();
          ibloc(ddg);
          sresult = system("mode spe auto");
          if (trig == -1)
               longjmp(mark, -1);
     }


     input2 = gets(inputs2);
     low = atoi(input2);
}
while (high == 0 ) {
     _setvideomode(_DEFAULTMODE);
     _settextcolor(ecolor);
     _settextposition(10, 15);

     _outtext("Input High");
     rcoord.row++;
     _settextposition(rcoord.row, rcoord.col);
     if (trig_check == 0) {
          trig = 0;
          sresult = system("mode spe com");
          while (kbhit() == 0 && trig == 0)

               trig = check();
          ibloc(ddg);
          sresult = system("mode spe auto");
          if (trig == -1)
               longjmp(mark, -1);
     }
```

```
                input2 = gets(inputs2);
                high = atoi(input2);
        }

}
```

LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
INFORMATION RESOURCES DEPARTMENT
BERKELEY, CALIFORNIA 94720