SPIN-LABEL ELECTRON PARAMAGNETIC RESONANCE CHARACTERIZATION OF HETEROGENEOUSLY FUNCTIONALIZED DENDRIMERS

by

Eric Dale Walter

A thesis submitted in partial fulfillment
Of the requirements for the degree

of

Doctor of Philosophy

in

Chemistry

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2004

©COPYRIGHT

by

Eric Dale Walter

2004

All Rights Reserved

APPROVAL

Of a thesis submitted by

Eric Dale Walter

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the college of Graduate Studies.

Dr. David Singel

Approved for the Department of Chemistry

Dr. Paul Grieco

Approved for the College of Graduate Studies

Dr. Bruce R. McLeod

STATEMENT OF PERMISSION TO USE

In presenting this dissertation in partial fulfillment of the requirements for a doctoral degree at Montana University, I agree that the Library shall make it available to borrowers under rules of the Library. I further agree that copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for extensive copying or reproduction of this dissertation should be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, Michigan 48106, to whom I have granted "the exclusive right to reproduce and distribute my dissertation in and from microfilm along with the non-exclusive right to reproduce and distribute my abstract in any format in whole or in part."

Eric Dale Walter

April 12, 2004

ACKNOWLEDGMENTS

I would like to thank the following people for their assistance with this project: Dr. David Singel and the members of his research group. Specifically, Dwight Schwartz for help with Awk and C programming, Alex Boulet for his coding of C++ programs, Roberts Usselman for his Labview programming and Karl Sebby for his assistance in obtaining EPR and MALDI-TOF spectra and chemical synthesis.

I would also like to thank Dr. Mary Cloninger and the members of her research group: Eric Woller and Joel Morgen for help with obtaining MALDI-TOF spectra. Lynn Sameulson and Dr. Mary Cloninger for synthesizing the nessesary isothiocyanate compounds.

Dr. John Peters with Dr. Paul Benton and Jolene Shuster provided Nitrogenase samples.

Mark Reinholz and Dwight Schwartz assisted with the construction of several electronics projects.

LIST OF TABLES

TABLE **PAGE**

1-1. Manufacturers specifications for PAMAM dendrimers 1

4-1. Pearson VII M and W values calculated on a grid of R_{max} and r_{min} 35

5-1. Comparison of calculated R_{max}, r_{min} for two calculation methods..... 60

5-2. Ratio of edge length to circumscribed diameter for selected polyhedra 65

LIST OF FIGURES

FIGURE **PAGE**

1-1. Generation 4 PAMAM dendrimer 2

2-1. General scheme for synthesis of heterogeneously functionalized G(4) dendrimers 10

2-2. MALDI-TOF spectra for partially spin-labeled G(4). 11

2-3. Average loading as determined by MALDI..... 12

2-4. MALDI-TOF spectra for partially spin-labeled G(0). 13

2-5. MALDI-TOF spectra for partially spin-labeled G(1) 14

2-6. MALDI-TOF spectra for partially spin-labeled G(2) 14

2-7. MALDI-TOF spectra for partially spin-labeled G(4) 15

2-8. MALDI-TOF spectra for partially spin-labeled G(5) 15

2-9. MALDI-TOF spectra for partially spin-labeled G(6) 16

2-10. Line widths from MALDI-TOF spectra of G(4) partially loaded with TEMPO 17

2-11. Fraction of dendrimers (X_g) with N_r sites reacted for a normal distribution..... 19

3-1. EPR spectra of G(4) PAMAM partially loaded with TEMPO 21

3-2. EPR spectra of frozen solutions of 4-aminoTEMPO..... 22

3-3. Illustration of peak height ratios 23

3-4. A/B ratios for partially and heterogeneously functionalized G(4) PAMAM 24

3-5. Fourier transform deconvolution 27

LIST OF FIGURES - CONTINUED

Figure Page

3-6. PearsonVII functions of constant width (W) and varying shape (M) 29

3-7. Pearson VII parameters for partially and heterogeneously functionalized G(4) PAMAM 30

4-1. Pseudo-code for rms.c 33

4-2. Pseudo-code for the C programs ranbroad.c and avoidbroad.c 34

4-3. Illustration of 3 loading models 36

4-4. Pearson VII and W values calculated for 55 spins with R_{max} = 68 Å and r_{min} = 7 Å 37

4-5. A/B ratios for 3 loading models 38

4-6. Effect of a loading distribution on calculated dipolar line widths 39

4-7. Energy level differences for two limiting cases 41

4-8. Shift in energy in terms of T for various ratios of Δ / T 42

4-9. Comparison of convoluted spectra when change in the lbf as a function of position of the spectrum is taken into account 43

4-10. Effect of ellipsoid distortion 46

4-11. Effect of dimensionality on dipolar broadening 48

4-12. Effect of allowing spins to occupy a spherical shell 49

5-1. EPR spectra of G(0) PAMAM partially loaded with TEMPO 52

LIST OF FIGURES - CONTINUED

Figure Page

5-2. EPR spectra of G(1) PAMAM partially loaded with TEMPO 52

5-3. EPR spectra of G(2) PAMAM partially loaded with TEMPO 53

5-4. EPR spectra of G(3) PAMAM partially loaded with TEMPO 53

5-5. EPR spectra of G(4) PAMAM partially loaded with TEMPO 54

5-6. EPR spectra of G(5) PAMAM partially loaded with TEMPO 54

5-7. EPR spectra of G(6) PAMAM partially loaded with TEMPO 55

5-8. A/B ratios for generations 0 through 6 partially loaded with TEMPO..... 55

5-9. Pearson VII parameters best fit to experimental EPR spectra for generations 0 though 6 partially loaded with TEMPO 56

5-10. "Pake pattern" 57

5-11. Pseudo-code for repeated splittings program..... 58

5-12. Repeated splitting simulations for 2 spins 59

5-13. Comparison of lbf's calculated for 100% spin labeled G(1) 61

5-14. Calculation of Dipolar width 62

5-15. Comparison of calculations with different shell thicknesses with data for G(5)..... 63

LIST OF FIGURES – CONTINUED

Figure Page

5-16. Comparison of calculations with different shell thicknesses with data for G(6) 63

5-17. Calculations of the possible number of 70Å spheres tangent to an inner sphere 66

6-1. Pulse sequences 68

6-2. Echo decay (T_2) 70

6-3. Fit of echo recovery 70

6-4. T_1 and T_2 variation with temperature 71

6-5 Field swept T_1 spectra 73

6-6. Field swept T_1 spectra, peak normalized 73

7-1. Difference in dipolar broadening between open and closed conformation of CCMV predicted for site-directed spin labeling of different amino acids 78

7-2. 2-D plot of the difference in dipolar broadening for 2 different spin-label sites in the nitrogenase Fe Protein 79

TABLE OF CONTENTS

1. INTRODUCTION 1

PAMAM Dendrimers 1
Heterogeneous Funtionalization 3
SL-EPR 3
Distance Measurement 4
Dipolar Interactions of Spin-Labeled Dendrimers 5
Related Systems 7

2. SYNTHESIS AND CHARACTERIZATION 9

Hetrogeneous Functionalization 9
MALDI-TOF Mass Spectrometry 11
Average Loading 12

3. SL-EPR OF HETEROGENOUSLY FUNCTIONALIZED G(4) 20

Experimental Conditions 20
Peak Height Ratios 20
Convolution and Deconvolution 25
Obtaining the lbf by Fitting 28

4. COMPUTER SIMULATIONS 31

Spherical Approximation 31
Three Models of Inter-Siter Interactions 32
Calculation of an lbf 32
Distribution of Loadings 37
Dipolar Hamiltonian 39
Axial Ellipsoid Distortion 43
Dimensionality 46
Shell Model 49

5. GENERATIONAL TRENDS 51

EPR Data and Analysis 51
A/B Ratios 51
Pearson VII Parameters 51
Computer Simulations 56
Simulations with Distribution 61
Simulations with the addition of Shell 62
Simulations of Hemagutination Results 64

TABLE OF CONTENTS-CONTINUED

Polyhedra 64

6. TIME-DOMAIN TECHNIQUES 67

Introduction 67
Acquisition of Pulsed Operation at W-band 67
Relaxation Rates 69
2-Dimensional Spectra 72

7. CONCLUSION 76

Heterogeneous Functionalization of PAMAM Dendrimers 76
Virus Capsids 77
Nitrogenase Fe Protein 78

REFERENCES CITED 81

APPENDICES 89

Appendix A: EPR SPECTRA 90
Appendix B: RMS.C 96
Appendix B: RMS.C 96
Appendix C: RANBROAD.C 101
Appendix D: CLUSTBROAD.C 107
Appendix E: AVOIDBROAD.C 114
Appendix E: AVOIDBROAD.C 114
Appendix F: KINSUM.AWK 121
Appendix F: KINSUM.AWK 121
Appendix G: XCONVOLUTE.AWK 125
Appendix H: OBLATE.C 127
Appendix I: PROLATE.C 132
Appendix J: RANSOL.C 137
Appendix K: SOLBROAD.C 141
Appendix L: RANSURF.C 148
Appendix M: RANLINE.C 153
Appendix N: SPSH.C 158
Appendix O: RANDOM.CPP 163
Appendix P: AVOID.CPP 173
Appendix Q: CLUSTER.CPP 185
Appendix R: RMS70-10000.C 196
Appendix S: PDB.AWK 201
Appendix T: PROBROAD.C 205

ABSTRACT

Dendrimers are roughly spherical macromolecular polymers that can provide a molecular framework for the attachment of various functionalities. The range of application can be vastly expanded when attached with a heterogeneous mix of functionalities tailored to meet a specific need. The effective application requires the acquisition of several new techniques. First, chemical control of the average ratio of functional groups and analytical methods to determine the ratio. Secondly, a determination of the distribution of ratios. And finally, a method whereby the spatial distribution is determined. The ratio and distribution of ratios were determined by MALDI-TOF Mass Spectrometry. The average ratio was easily controlled by using stoichiometric ratios of reactants and the distribution of ratios was found to follow a normal distribution. This was determined by the analysis of the MALDI spectral line-widths. The spatial distribution was determined by a Spin-Label Electron Paramagnetic Resonance technique. By adapting and combining mathematical and spectroscopic techniques from other related systems, it was found that the EPR spectrum was sensitive to the spatial distribution of spin-labels included as one of the attached functional groups. The dipolar interaction between spins lead to an increase in spectral line-broadening that could be compared to computer simulations of different spatial distributions. For all cases studied the distribution was found to be random.

CHAPTER 1

INTRODUCTION

PAMAM Dendrimers

Dendrimers are hyperbranched polymeric macromolecules[1] that have proved to be a useful platform for tethering bioactive functional groups. Part of their utility arises from the wide range of sizes that are available; the dimension of almost any bio-macromolecule can be matched[2]. The terminal groups may be functionalized with a variety of groups, with the possibility of several different groups on the same dendrimer (hereafter referred to as heterogeneous functionalization)[3, 4].

Table 1-1. Manufacturers specifications for PAMAM dendrimers.

Generation	M.W.	Diameter (Å)	Number of terminal groups
0	517	15	4
1	1430	22	8
2	3256	29	16
3	6909	36	32
4	14215	45	64
5	28826	54	128
6	58048	67	256
7	116493	81	512
8	233383	97	1024
9	467162	114	2048
10	934720	135	4096

With in the confines of this work, the dendrimer will be a PAMAM (polyamidoamine) dendrimer with an ethylene diamine core, also known by the manufacturer (Dendritech) by the trademarked term “Starburst dendrimer” or SBD. Within this family of dendrimers are different generations, each generation possessing

double the number of terminal amines as the previous. Figure 1-1 shows the structure of a generation 4 (hereafter designated G(4)) with the underlying generations shown in different colors. The accompanying Table 1-1 gives the theoretical molecular weights, numbers of endgroups and diameters as supplied by the manufacturer.

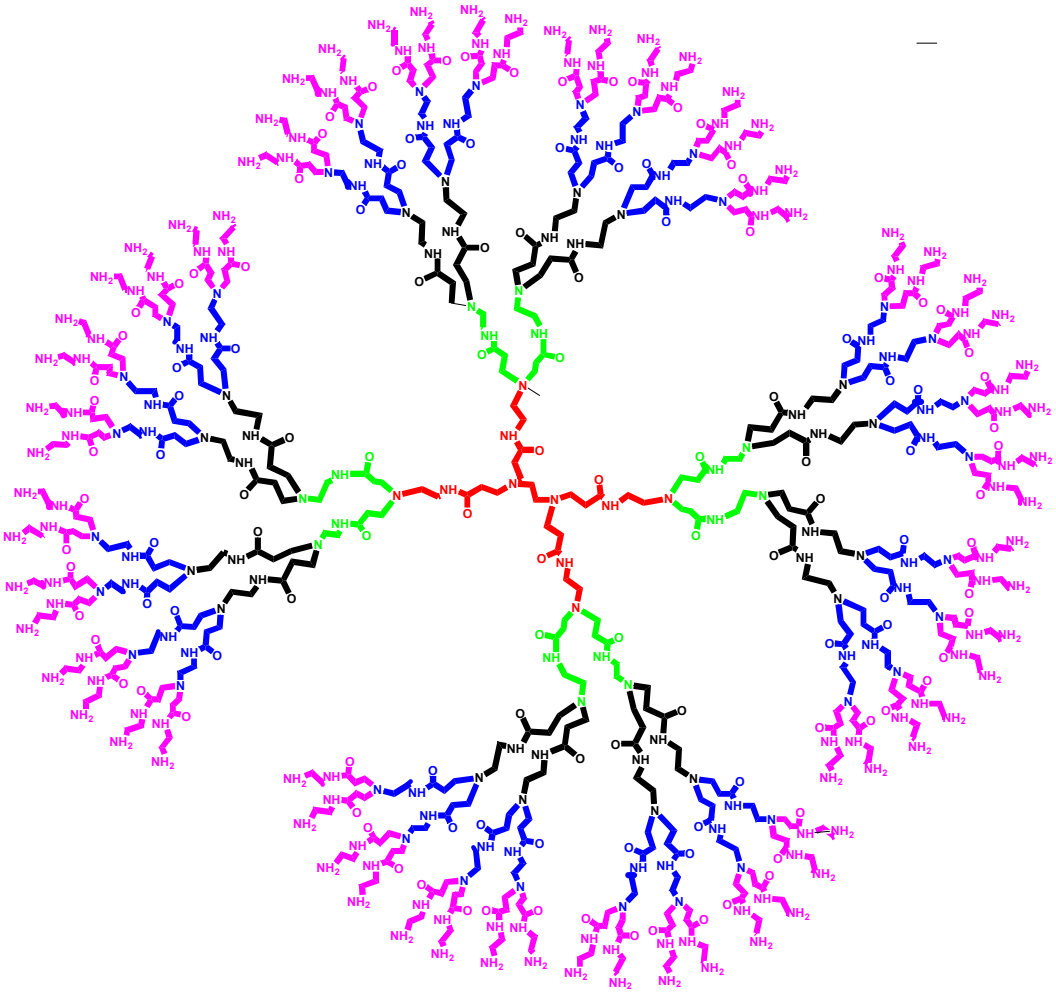


Figure 1-1. Generation 4 PAMAM dendrimer with different generations highlighted by color. G(0) red, G(1) green, G(2) black, G(3) blue and G(4) pink.

Heterogeneous Funtionalization

Impetus for this work was provided by our collaborator Dr. Mary Cloninger, whose work involves the investigation of mulitivalent protein-carbohydrate interactions[5, 6]. Dendrimers can provide a framework of variable sizes (different generations) that can be functionalized not only with carbohydrate groups, but also with spectroscopic probes, solubilizing agents and possibly diagnostic or therapeutic agents[7, 8].

In order to successfully bind to lectins multivalently, it is necessary to have the carbohydrate groups at distances that match the bindings site on the protein. If fully functionalized (loaded) with sugars, the range of distances available will just be related to the size of the dendrimer. However, for partial or heterogeneous loading, the available distances will depend on the spatial distribution of the sugars. If the spatial distribution is random, then a minimal amount of loading will yield all possible distances (for the size of the dendrimer) in a combinatorial fashion. If the there is clustering the maximum distance will be a function of loading: low loadings will only give short distances. Conversely, if the functional groups avoid each other, low loadings will only results in large distances.

SL-EPR

Electron Paramagnetic Resonance (EPR) Spectroscopy can be used to obtain information about long range distances in macromolecules[9-14]. Specifically, a technique known as Site-Directed Spin-Label (SDSL) EPR[15, 16] is has been developed to measure distances in the range of 5-25 Å (it is possible to extend the maximum

distance to 35 Å or even 50 Å by using specialized pulsed EPR techniques[17-21]) in proteins. Spin-labels are introduced into the protein by the site directed mutagenesis of selected amino acid residues to cysteines. A spin-label is chemically attached to each mutated site, usually through a disulfide bond. The spin-label most commonly used is a stable nitroxide free-radical.

The nitroxide EPR spectrum has been well characterized and thoroughly studied. Analysis of the EPR spectrum can yield various pieces of information. If the spectrum is obtained from a fluid solution, the rotational correlation time (T_c) can be obtained and by inference, information about the mobility of the spin-label[22]. The local dielectric field has a known influence on the spectrum, having been determined empirically by comparing the values of the Zeeman and hyperfine coupling values from the EPR spectra of spin-labels in various solvents[23]. The accessibility of the spin-label can be determined by introducing broadening agents (oxygen, chromium oxalate, NiETDA) into the sample[24]. These broadening agents are paramagnetic molecules of varying size and polarity, if the spin-label is able to collide with them, the EPR spectrum will be broadened.

Distance Measurement

If two sites in the protein are labeled, then the distance, r, between the spin-labels can be calculated from the increase in spectral line-width due to the dipolar interaction between the labels, Equation 1-1. Several methods[9] of quantifying the dipolar line-broadening are reported in the literature: analysis of peak height ratios[25]; deconvolution[13]; and simulation of the spectrum[12]. These methods will be explored and employed in Chapters 3 and 4. The astute reader can now guess our strategy:

employ spin-labels as one of the groups in a heterogeneous functionalization of the dendrimer and abstract information about the spatial distribution from the dipolar interaction. The simulation and interpretation of the dipolar interaction will require the development of techniques to deal with 2-200 spin-labels, instead of the prior limit of exactly two.

$$H_{dipole} = S(S+1)(\frac{g^2\beta^2}{r^3})(1-3\cos^2\theta) \qquad [1\text{-}1]$$

The mathematical framework for the analysis of the dipolar interaction between many spins has been constructed for systems where the spins are in a continuum: paramagnetic species doped into a crystal, frozen solutions of paramagnets, and bulk objects that have had radicals formed by radiation damage. The magnitude of the dipolar interaction between two spins is proportional to the inverse cube of the distance between the spins, r. For two spins the exact dipolar Hamiltonian can be expressed as a function of r and θ (the angle between the external field and the vector connecting the two spins)(Equation 1-1). For many spins in an infinite lattice, the dipolar field induces a broadening of the spectrum that is proportional to the concentration of spins (Equation 1-2).

$$\Delta H_{dipole} = 2.3(g\beta\rho)[S(S+1)]^{1/2} \qquad [1\text{-}2]$$
$$\rho = density\ of\ spins$$

<u>Dipolar Interactions of Spin-Labeled Dendrimers</u>

For spins in an infinite lattice, but with the exact position of each spin known, the second moment of the dipolar interaction, $<H^2>$, can be calculated by using an analog of Equation 1-1 for the closest neighbors and Equation 1-2 to account for the rest of the continuum[26], this is shown in Equation 1-3. Our system has elements that deviate from these limiting cases: a finite lattice (the surface of the dendrimer), more than two spins, but the exact position of the spins is not known (while the terminal amines of a PAMAM dendrimer are thought to define the surface of a sphere, the exact positions are ill-defined). While the exact method of calculation will be expanded upon in Chapter 4, here it is sufficient to say that the first part of Equation 1-3 (excluding the integral) is used to calculate the dipolar broadening from computer generated arrangements of spins. These arrangements are generated for various lattice models to simulate the spatial distribution of spins on a heterogeneously loaded dendrimer.

$$< H^2 > \approx 3/5 g^2 \beta^2 S(S+1)(\sum r_{ij}^{-6} + 4\pi\rho \int_{a_0}^{\infty} \frac{r^2 dr}{r^6}) \qquad [1\text{-}3]$$

$$\rho = density\ of\ spins,\ \ a_0 = dist.\ to\ first\ spin\ not\ in\ summation$$

These simulations must be compared to experimental spectra; to obtain a unique result it is necessary to simulate spectra from many different levels of loading. Equation 1-3 allows a calculation of the dipolar interaction, however, many different distributions of r_{ij} values will give the same aggregate result. If the amount of dipolar broadening as a function of loading is simulated, however, additional constraints are introduced. When combined with reasonable guesses as to the size and shape of a dendrimer a unique characterization can be obtained. Experimentally, this approach requires a series of

samples with different levels of spin-label loading, the details of sample preparation are given in Chapter 2.

The fitting of these experiment spectra will allow the spatial distribution of the spin-labels to be determined. More interesting however is the spatial distribution of other non-paramagnetic functionalities. Therefore, for each other substituent, two series of heterogeneously functionalized dendrimers are synthesized. In the first series, a certain loading level of the alternate is first established, then the spin-label is allowed to bind to all of the remaining sites. In this way the spatial distribution of the non-paramagnetic group can obtained: it must be occupying the sites of not occupied by the spin-labels. The other series has the same loading levels, but the spin-label is added first. Spectra from this second series are needed to insure that there are no effects resulting from interaction between the two substituents.

Related Systems

Although the dendrimer products and techniques that are developed to characterized them are new, there has been work previously reported on similar techniques and compounds. Spin-labeled dendrimers have been made by several groups for a variety of reasons. Using different chemistry to attach the labels, Tomalia et. al. have reported several studies where PAMAM dendrimers, with low loadings of spin-labels (10% of sites), have been used to probe interactions with other macromolecules; DNA[27, 28], vesicles[29-31], surfactants[32-35], and polynucleotides[36, 37]. The interpretation of the EPR data assumes a random distribution of spin-labels, however, they give no evidence to support this assumption. Another group spin-labeled all sites on

five generations of DAB (polypropylene imine) dendrimers with the goal of monitoring dendrimer dynamics and hydrogen bonding between terminal groups[38]. This was done by observing the exchange interaction as evidenced in room-temperature (fluid) EPR. Spin-labeled dendrimers have been proposed as useful relaxation agents for MRI[39-41]. EPR has also been used to study other polymers, either spin-labeled or containing an organic radical[42, 43].

Tsvetkov used echo-detected EPR to determine the spatial arrangement of radicals[44]. These radicals were formed in frozen solutions by radiation damage. By studying the phase relaxation due to spin-spin interactions, it was shown that there were areas of high local concentration and that the radicals were formed along the path of the α–particle. We will attempt to extend the use of pulsed EPR (Chapter 6) to not only determine that the spins are not homogeneously distributed, but also determine the exact nature of that distribution.

CHAPTER 2

SYNTHESIS AND CHARACTERIZATION

Hetrogeneous Functionalization

A wide variety of heterogeneously functionalized PAMAM dendrimers were synthesized for this project. They fall into two main categories: partially functionalized with the EPR spin-label 2,2,6,6 tetramethyl piperidine oxyl (TEMPO), (dendrimer generations G(0) through G(6)) and those heterogeneously functionalized with TEMPO and some other group (G(4) only). For each of these cases, a product with the following TEMPO loading (as a percent of total dendrimer amino groups) was synthesized: 5%, 10%, 25%, 50%, 75%, 90%, 95%, 100%. In the case of heterogeneous loading, the non-TEMPO group's loading was simply the complement.

Spin-labeled PAMAM dendrimers were synthesized following the general scheme shown for G(4) in Figure 2-1. PAMAM dendrimer was purchased from either Dendritech or Aldrich as a solution in either water or methanol. Solvent was removed (lyophilization if solvent was water, or rotary evaporation, then redisolution in water and lyophilization, if solvent was methanol) and a stock solution made up in DMSO (25 mM in terminal amines, unless noted).

Isothiocyanates **1-4** were synthesized in the lab of Dr. Mary Cloninger here at MSU. Isothiocyanate **5** (t-butyl isothiocyanate) was purchased from Aldrich. Stock solutions of the isothiocyanate reactants were also made in DMSO at the same concentration as the dendrimer (25 mM). Stoichiometry was then simply proportional to

volume. Total reaction volume was in the range of 200-400 µl. To preserve a constant dendrimer concentration in the partially functionalized case, a complementary volume of DMSO was added. Problems with the solubility of t-butyl loaded G(4) necessitated a 3 fold decrease in concentration from the stock solutions for those reaction mixtures.

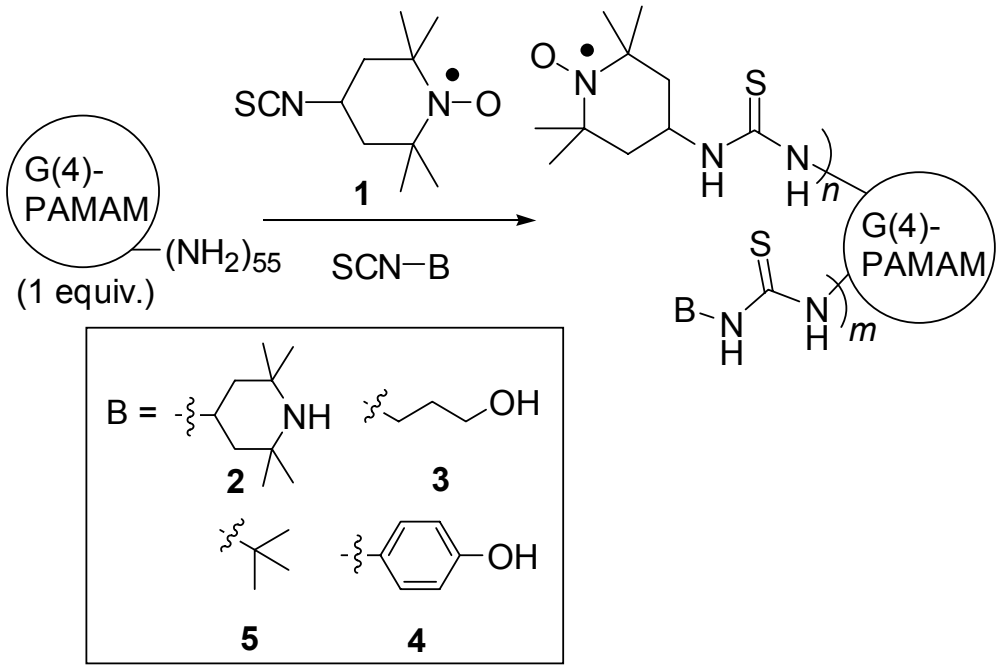


Figure 2-1. General scheme for synthesis of heterogeneously functionalized G(4) dendrimers.

For partial loading with one reactant, the reaction mixture was allowed to react for at least 48 hours. For sequential heterogeneous loading of two reactants, the first was added and allowed to react for 48 hours, then the second reactant was added and a further 48 hours allowed for reaction. For simultaneous heterogeneous loading, the two isothiocyanates were first mixed in the proper stoichiometric ratio and then the dendrimer solution added. The time of 48 hours was chosen as a safe minimum time for completion

of the the reaction, as assessed by the absence of change in the spectrum of the reaction mixture after 24 hours by EPR (for TEMPO), NMR (for diamagnetic groups) and by MALDI-TOF and UV-Visible (for all functional groups). All reactions were carried out at room temperature.

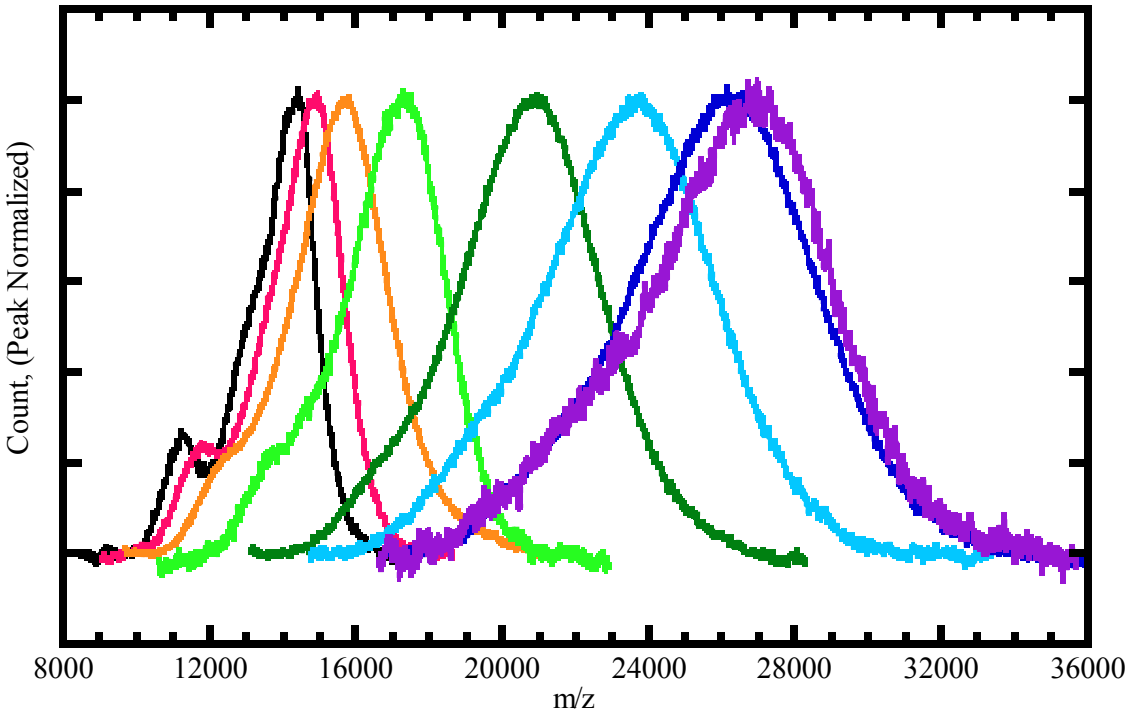


Figure 2-2. MALDI-TOF spectra for partially spin-labeled G(4). 0% (Black), 5% (Red), 25% (Orange), 50% (Lt. Green), 75% (Dk. green), 90% (Lt. Blue), 95% (Dk. Blue), 100% (Purple)

MALDI-TOF Mass Spectrometry

Of the aforementioned spectroscopic techniques, MALDI-TOF[45, 46] proved to be the most useful in determining the average loading[6]. Spectra were obtained on a Bruker Biflex III in linear mode. Calibration was done with Trypsinogen and BSA as external standards. The matrix was IAA (trans-indole acetic acid) at 35 mg/ ml in DMF.

Optimal spectra were obtained with 9 µl of matrix mixed with 1µl of 1mM dendrimer solution.

Average Loading

Loading was determined by the shift in mass of the dendrimer MALDI signal at maximum amplitude, divided by the mass of the reactant. For sequential reaction, it is necessary to test both before the second reactant is added and after. While NMR can give a more precise answer, (the ratio of the integrated amplitudes of the amine protons vs. thiourea protons) it cannot be used when one of the reactants is paramagnetic.

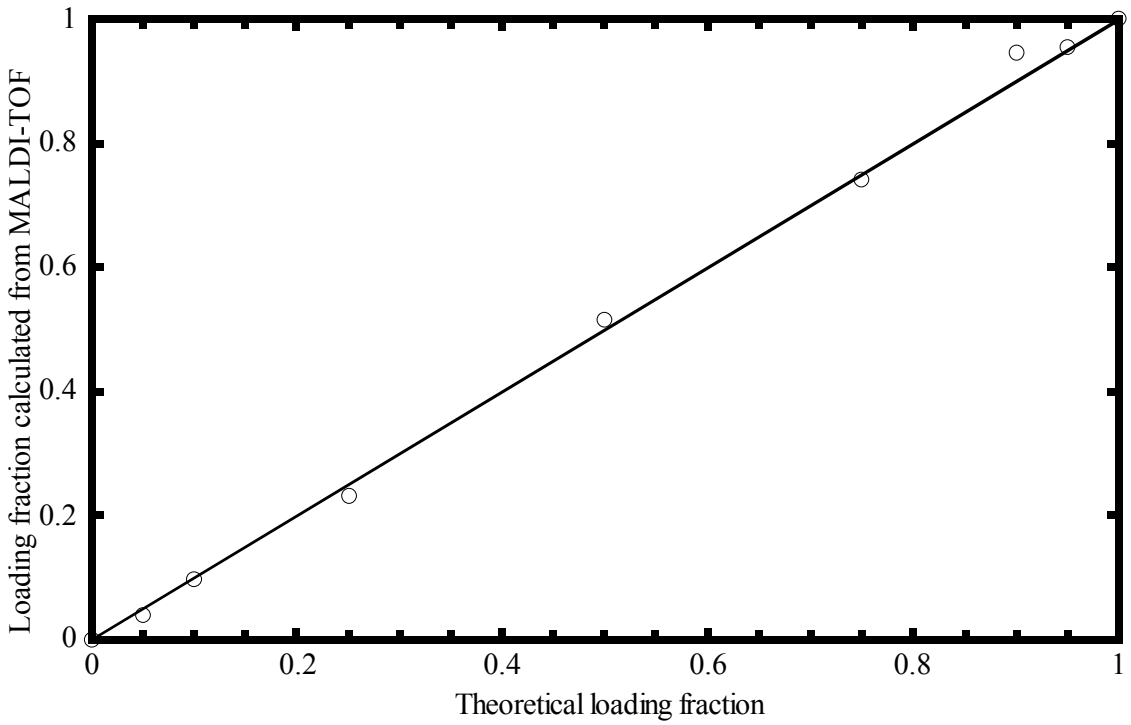


Figure 2-3. Average loading as determined by MALDI (open circles) compared with theoretical values, points falling on the solid line would be in perfect agreement.

Integration of the EPR signal can be used to calculated total TEMPO concentration, but it cannot give a ratio to a diamagnetic group (it is possible to lyophilize the product, get a weight, then calculate a ratio, however, the very small batch size of our products made this procedure unwieldy).

Average loading was found to be in good agreement with the theoretical loading ratios calculated on the basis of the number of reagent equivalents to which the dendrimer was exposed. Minor deviations observed are attributable to variations in concentrations, volumetric accuracy of the glassware used and difficulty in finding the exact shift in mass. Figure 2-2 shows MALDI spectra for TEMPO loaded G4 PAMAM. Shift in center peak divided by total shift is plotted vs. theoretical loading fraction in Figure 2-3.

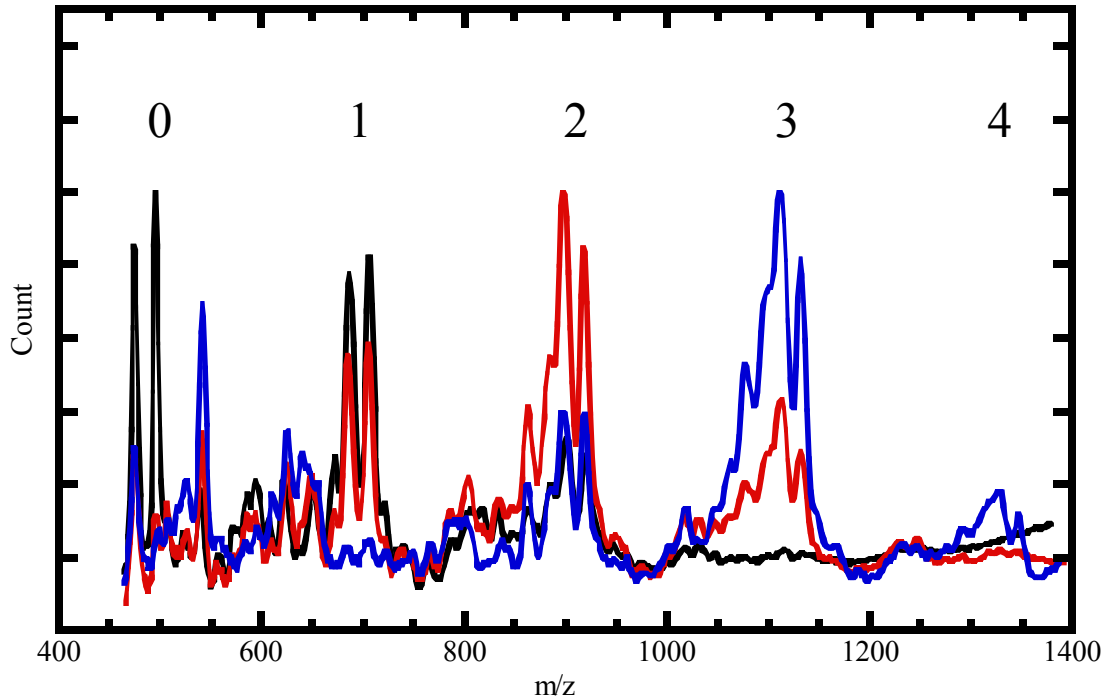


Figure 2-4. MALDI-TOF spectra for partially spin-labeled G(0). 25% spin-labeled (Black), 50% spin-labeled (Red), 75% spin-labeled (Blue). Numbers above peaks indicate number of spin-labels bound.

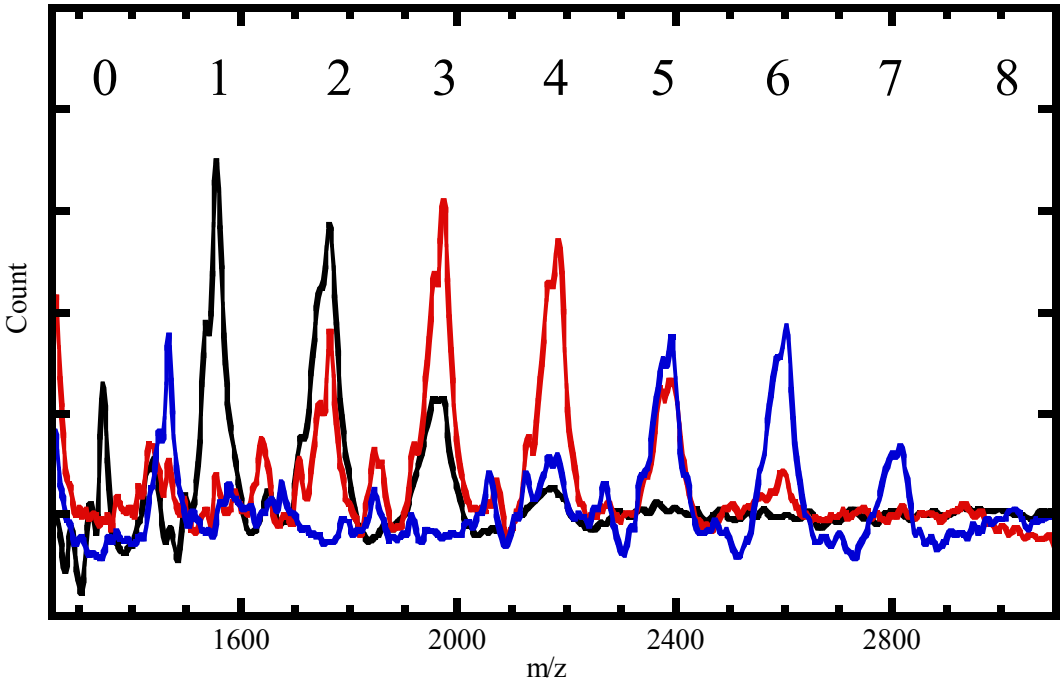


Figure 2-5. MALDI-TOF spectra for partially spin-labeled G(1). 25% spin-labeled (Black), 50% spin-labeled (Red), 75% spin-labeled (Blue). Numbers above peaks indicate number of spin-labels bound.

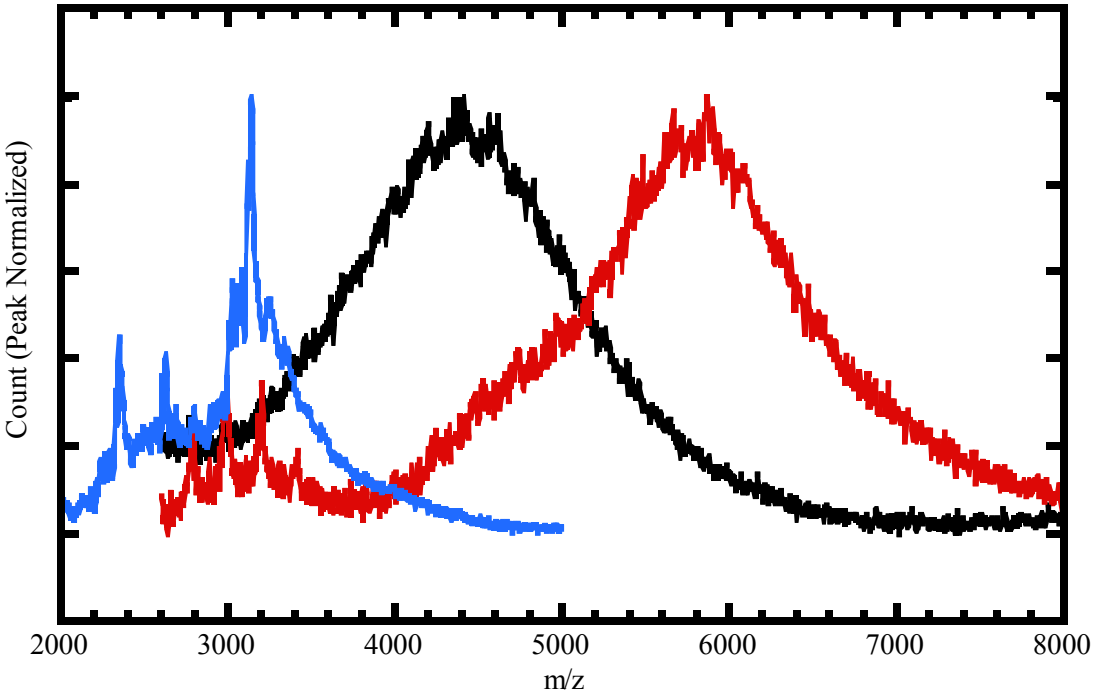


Figure 2-6. MALDI-TOF spectra for partially spin-labeled G(2). 0% spin-labeled (Blue), 25% spin-labeled (Black), 50% spin-labeled (Red).

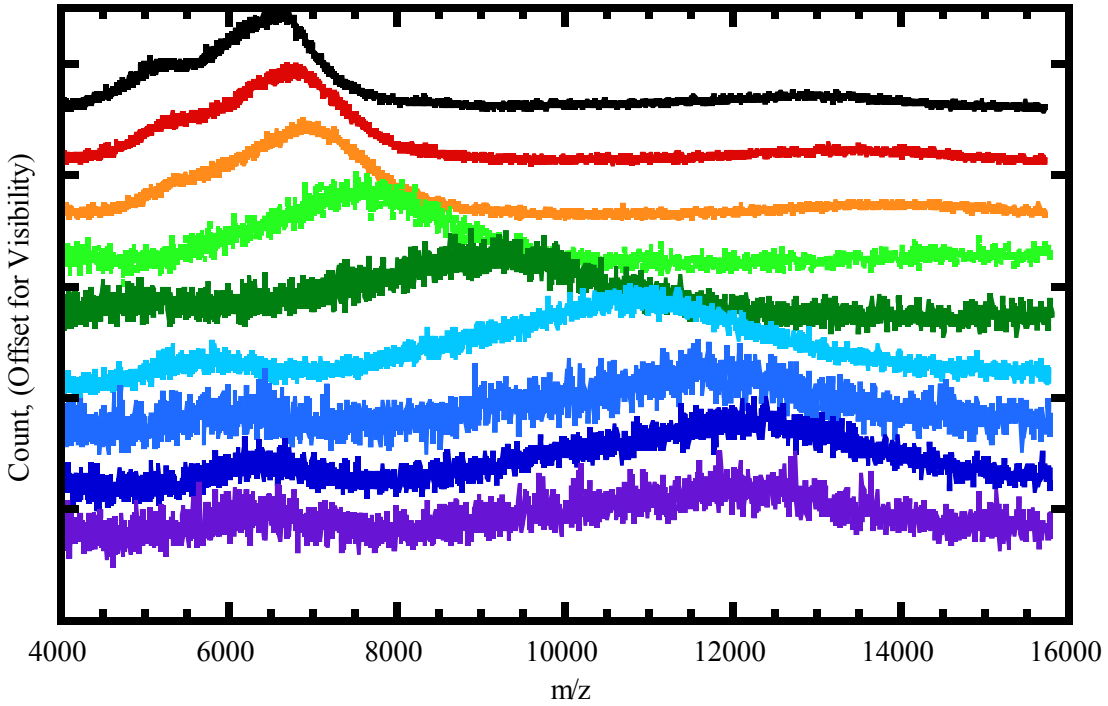


Figure 2-7. MALDI-TOF spectra for partially spin-labeled G(3). 0% (Black), 5% (Red), 10% (Orange), 25% (Lt. Green), 50% (Dk. green), 75% (Lt. Blue), 90% (Med. Blue), 95% (Dk. Blue), 100% (Purple)

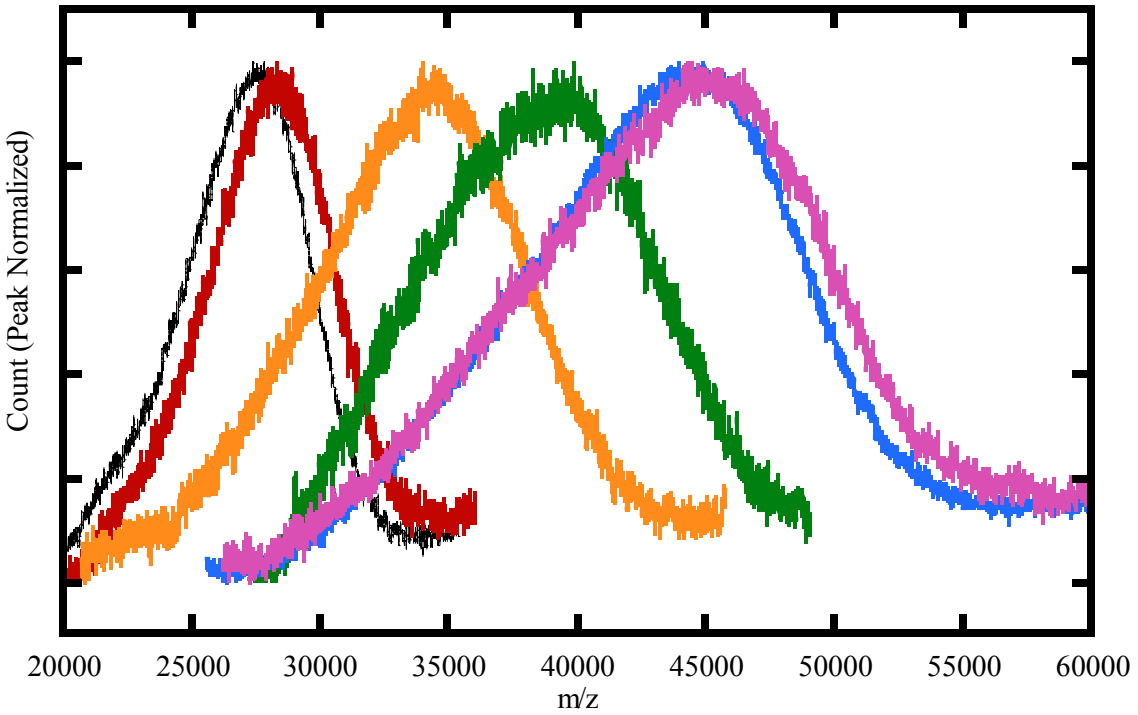


Figure 2-8. MALDI-TOF spectra for partially spin-labeled G(5). 5% (Black), 25% (Red), 50% (Orange), 75% (Green), 95% (Blue), 100% (Purple)

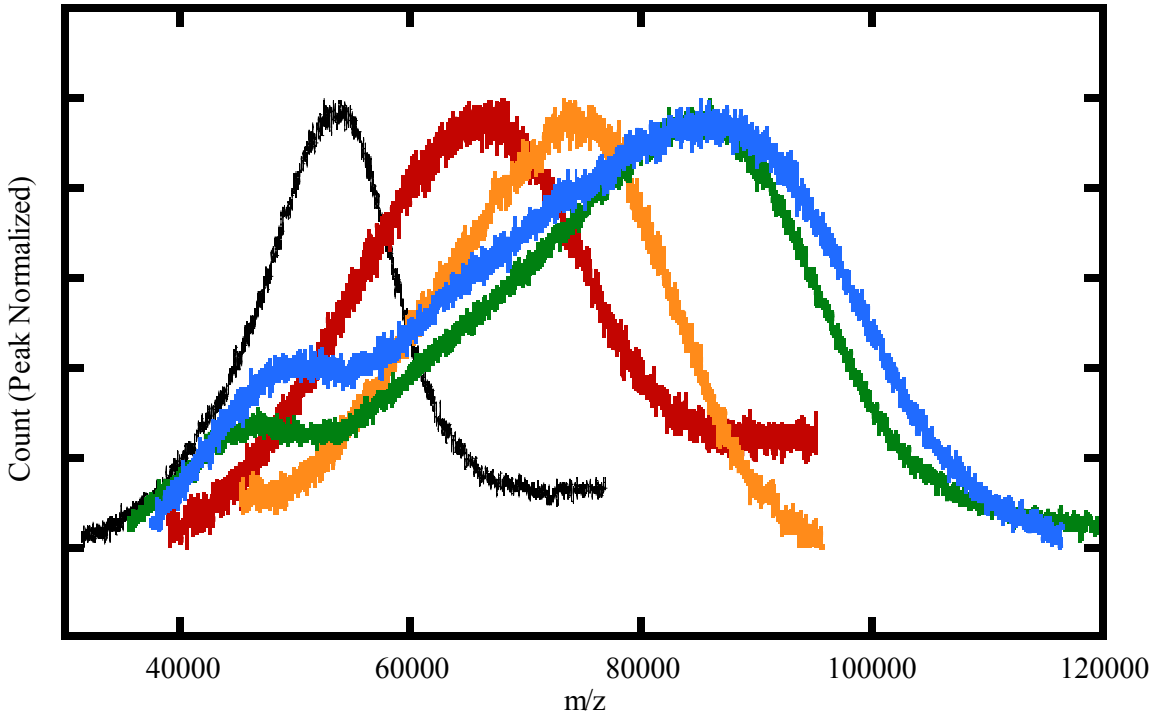


Figure 2-9. MALDI-TOF spectra for partially spin-labeled G(6), 5% (Black), 50% (Red), 75% (Orange), 90% (Green), 100% (Blue).

Distribution of Loading Levels

The distribution of loading was also investigated by MALDI-TOF. Stack plots for the other generations, G0 through G6, are shown in Figures 2-4 to 2-9. The analysis of G0 and G1 is rather simple, in comparison to the higher generations: peaks for each degree of loading are separate from neighboring peaks. These individual peaks can be integrated and the loading distribution for each compound calculated.

The larger dendrimers provide more of a challenge. The linewidth of the un-reacted dendrimer is about ten times the mass shift associated with a single TEMPO. This linewidth is due to the compiling of errors in the synthesis of the dendrimer. We can, however, observe a change in the MALDI linewidth with loading. Because the distribution of masses corresponds to a distribution of numbers of reactive endgroups,

linewidth should increase with loading. For instance, for G(4), there are some molecules with no defects or 64 reactive sites; there are also molecules with many defects and as few as 44 sites. When completely reacted the new mass for each is simply shifted upward by the mass of the reactant multiplied by the number of reactive sites.

The difference in mass between of molecules having 64 and 44 sites increases by 20 times the mass of the reactant. At intermediate loadings, the increase in width scales with fractional loading. This effect is therefore linear with loading.

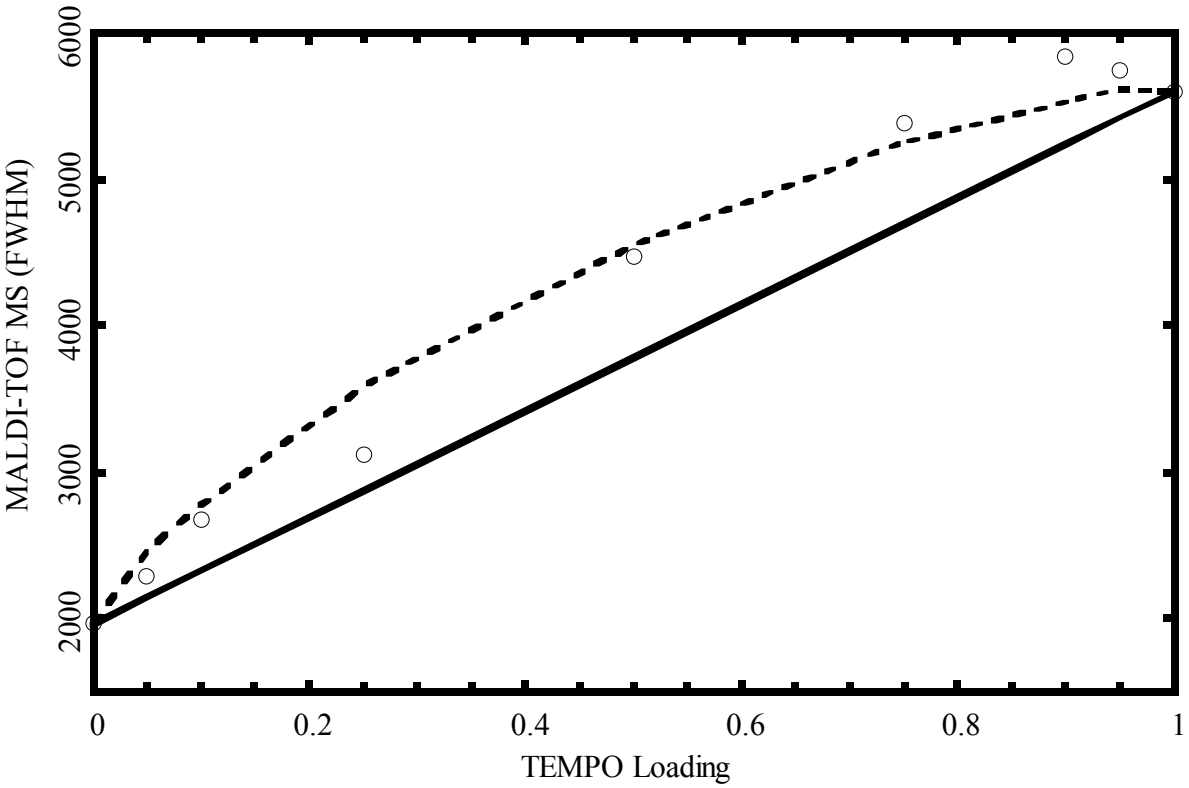


Figure 2-10. Line widths from MALDI-TOF spectra of G(4) partially loaded with TEMPO (spectra shown in Figure 2-2) are plotted vs. loading (open circles). Solid line is predicted for lack of distribution. Dashed line shows line widths predicted by Equations 2-1 and 2-2.

The effect of a distribution of products - a dendrimer to dendrimer variation in the number of attached groups - would perturb this linear behaviour. While such a distribution would generally augment the line-width, it is necessarily zero at 0% and 100% loading. The experimental MALDI linewidths for TEMPO loaded G(4) (solid points, Figure 2-10) show a non-linear progression with loading. We suspected that the loading distribution might follow a normal distribution law – that is:

$$X_g = \{ L^{Nr} (1-L)^{(N-Nr)} N! \} / \{N_r! (N-N_r)!\}, \qquad [2\text{-}1]$$

in which, X_g is the fraction of molecules with N_r reacted sites on a dendrimer of N total reactive sites, and L is the loading. At large N, this distribution function, tends toward Gaussian shapes with widths of $[4NL(1-L)]^{1/2}$; the maximal width, namely $N^{1/2}$, occurs for the half-loaded dendrimer.

X_g, calculated from Equation 2-1, is plotted against N_r for relevant values of L in Figure 2-11. By numerical simulations, we ascertained that the convolution of the observed MS peaks with a Gaussian function enlarges their widths according to:

$$W=(W_{MS}^{3/2}+W_g^{3/2})^{2/3}, \qquad [2\text{-}2]$$

in which W is the resultant width of a peak, after convolution of a MS peak of width W_{MS} with a Gaussian function of width W_g.

To test this idea of a random loading distribution, we computed the loading dependent width, as a function of L. We employed Equation 2-1 with N=55 to determine loading dependent distribution functions. The widths of these functions were combined,

via Equation 2-2, with values read from the straight line in Figure 2-10 to obtain the resultant values plotted as the curved line in Figure 2-10. The agreement between this line and the experimental data is very satisfactory: notwithstanding the limitations of the poor spectral resolution in the MALDI-TOF MS of this system, a simple analysis makes a strong case for a normal distribution of the loading. The data, moreover, clearly rule out any broad distributions in which there are substantial numbers of dendrimers having N_r/N far removed from the overall loading value. Such distributions would lead to a pattern of broadening that is inconsistent with the experimental observations summarized in Figure 2-11.

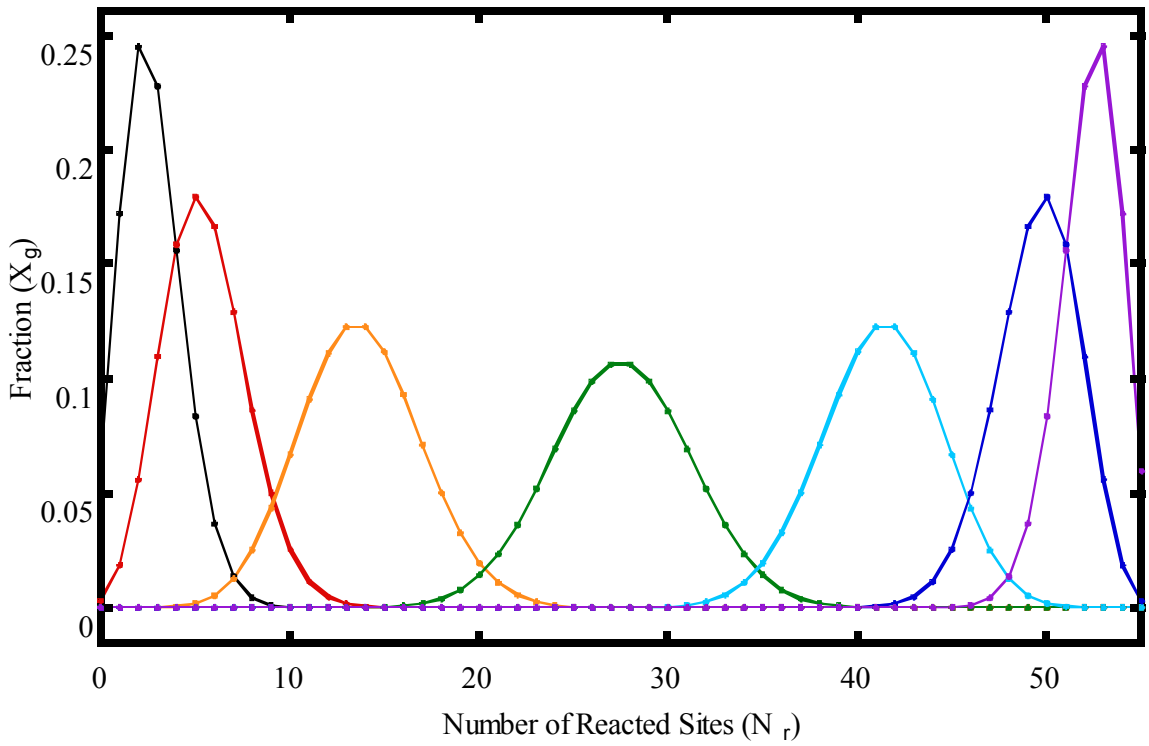


Figure 2-11. Fraction of dendrimers (Xg) with Nr sites reacted for a normal distribution for the following average loadings: 5% (Black), 10% (Red), 25% (orange), 50% (Green), 75% (Lt. Blue), 90% (Dk. Blue), 95% (Purple).

CHAPTER 3

SL-EPR OF HETEROGENOUSLY FUNCTIONALIZED G(4)

Experimental Conditions

EPR spectra were obtained on a Varian E-109 X-band spectrometer with a lab built computer interface. Low temperature spectra were taken with a 3mm ID quartz epr tube in a liquid nitrogen filled finger dewar. A 3:1 ratio of DMSO/glycerol was used as a solvent and dendrimer concentration was approximately 100 uM. The highest power attenuation (60 dB, which corresponds to a power of .0002 mW) was used to avoid power saturation effects. The amount of saturation was gauged by the linearity of the ratio of the double integral of the EPR spectrum vs. the concentration of TEMPO standards (47 ul – 300 mM). The field was typically scanned over a range of 200 G with 100 kHz modulation at an amplitude of 1 G. Scan rate was 100 G/min. and a time constant of 250 ms. was employed. Spectra for G(4) partially loaded with TEMPO are shown in Figure 3-1. Spectra for the heterogeneously loaded G(4) dendrimers are shown in Appendix A.

Peak Height Ratios

A series of solutions containing only the spin-label (no dendrimer) were used to probe the limits of detection for dipolar broadening. The spectra show noticeably increased line width at concentrations greater than 2mM (Figure 3-2). One measure of broadening in a nitroxide epr spectrum that has traditionally been employed is the so called "d/d_1"ratio[25]. It is simply the ratio of the differences in amplitudes between the outer two peaks and the inner two peaks (illustrated in Figure 3-2). We chose to use a

somewhat simpler peak height ratio, that between the first two peaks, labeled A and B in Figure 3-2. Plotting its value versus concentration served as a test of its validity: it is linear for our range of concentrations, as predicted by theory[26] (Figure 3-3).

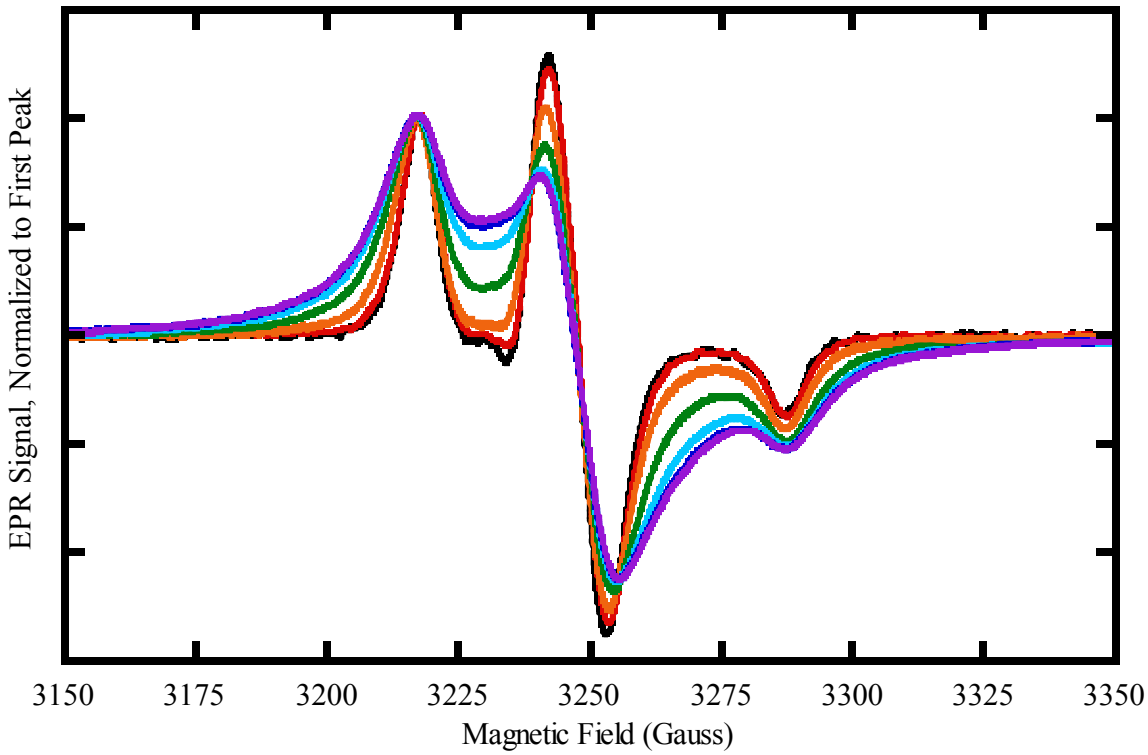


Figure 3-1. EPR spectra of G(4) PAMAM partially loaded with TEMPO. %5 (Black), 10% (Red), 25% (Orange), 50% (Green), 75% (Lt. Blue), 90% (Dk. blue), 95% (Purple).

While either of these peak height ratio methods works reasonably well for comparing linebroadening in a simple system, they both suffer from several limitations. Firstly, if the density of spins is not homogeneous in the sample, spins that are more isolated will have an inordinate amount of influence on the average peak height ratio. This is because the (double) integral of the spectrum is proportional to the number of spins in the sample; to preserve this ratio of area to number of spins, the broader spectral components must have less magnitude. This condition of heterogeneous local

concentration is also not well characterized by a single parameter. Secondly, peak height ratios are influenced by other spectral parameters.

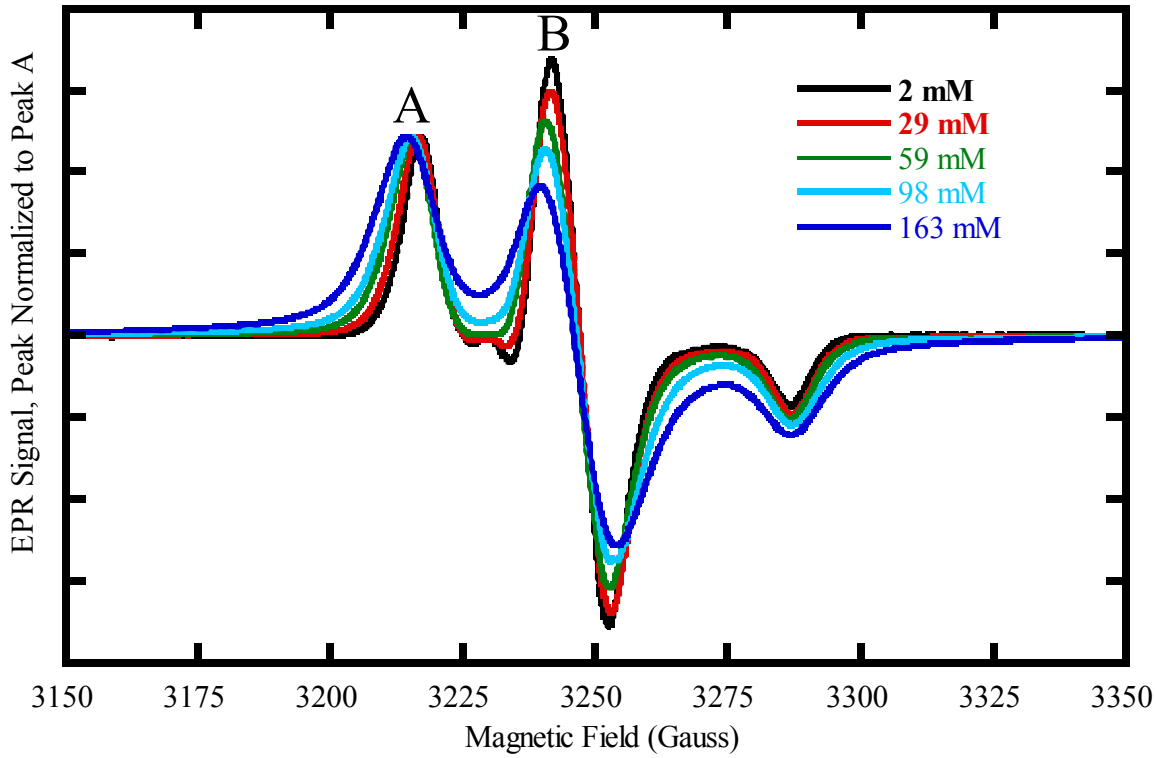


Figure 3-2. EPR spectra of frozen solutions of 4-aminoTEMPO.

For instance, the distance (in field units) between peaks will change with the local dielectric field, this can lead to large changes in peak height ratios for the same linewidth. Peak height ratios do not provide information about changes in shape (and therefore cannot fully characterize the spectrum). And lastly, these parameters do not arise directly from calculations of the dipolar broadening. With these limitations in mind, we will nonetheless employ the A/B ratio as a useful and informative parameter.

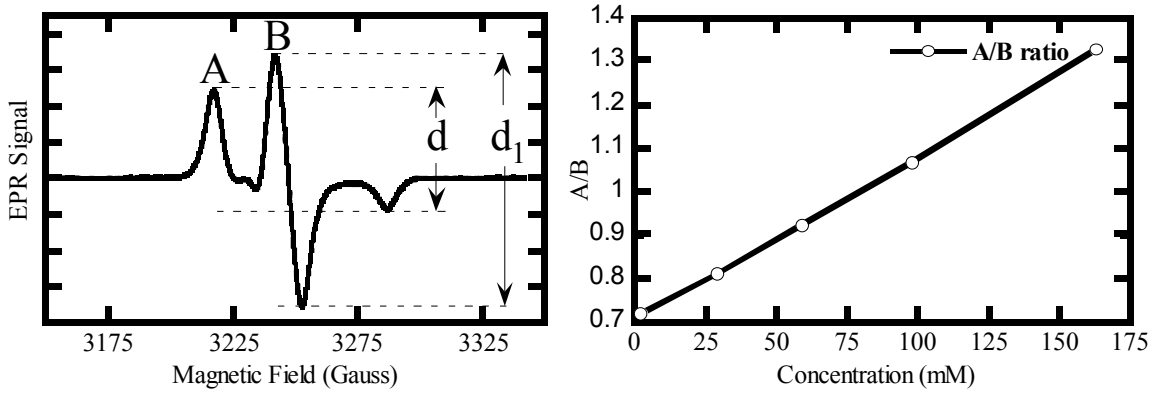


Figure 3-3. Right panel indicates parts of EPR spectrum used for peak ratio analysis. Left panel shows A/B ratio for EPR spectra of frozen TEMPO solutions (spectra in Figure 3-2).

It can be observed from the EPR spectra in Figure 3-1 and Appendix A that the peak positions do not shift with increased TEMPO loading or with heterogeneous functionalization. Nor, series to series, do the spectral shapes change appreciably for equal TEMPO loading. Therefore the A/B ratio can be used to compare linebroadening for different sequences of reaction and different non-TEMPO functional groups. A/B ratios for partially TEMPO loaded and heterogeneously loaded G(4) dendrimers are shown in Figure 3-4.

The A/B ratios shown in Figure 3-4 indicate that neither order of reaction nor type of reactant change the functional form of dipolar broadening with loading. Although the overall slope changes slightly with different functional groups, simulations (next chapter) will show that the trend with loading determines spatial distribution.

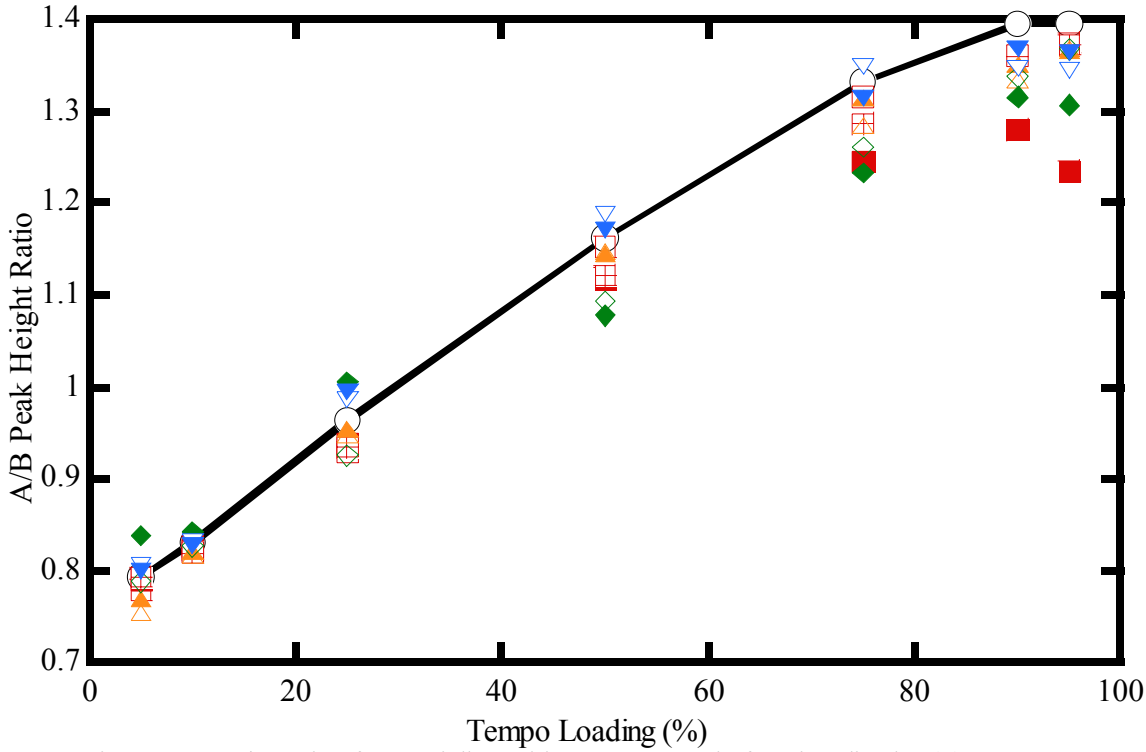


Figure 3-4. A/B ratios for partially and heterogeneously functionalized G(4) PAMAM. Open symbols represent reaction products when TEMPO isothiocyanate was added first, solid symbols for TEMPO added second, open with cross for simultaneous reaction. Color indicates second isothiocyanate (as shown in figure 2-1). None (black), 2,2,6,6 tetramethyl piperidine (red), propanol (orange), phenol (green) and t-butyl (blue).

While the spectral shapes of all the TEMPO loaded G(4) dendrimers are approximately the same (Appendix A) and the A/B ratios are in the same range as frozen TEMPO solution spectra (Figure 3-3), the dendrimer spectra are different from the TEMPO frozen solution spectra. This is obvious if the "valley" between peaks A and B is compared. Therefore, if the dipolar broadening of the dendrimer samples is to be characterized another method beyond the A/B ratio method mustbe used. Optimally, a set of parameters that fully describe the spectra should be found.

Convolution and Deconvolution

While the overall shape the spin-label EPR spectrum is complicated, the spectrum of each individual spin is not. In fact, for our purposes, we will consider it to be a spike (or delta function) whose position in the spectrum is governed by parameters not correlated with dipolar broadening. If each position in the spectrum is broadened equally, we may properly describe the broadened experimental spectrum (S_{exp}) as the convolution of a line-broadening function (lbf) with a basis spectrum (S_{iso})[12, 13] (in our case, the spectrum of an isolated spin label with no discernable dipolar broadening):

$$S_{\exp} = S_{iso} \otimes lbf \qquad [3\text{-}1]$$

Mathematically convolution can be expressed by the following integral:

$$S_{\exp}(t) = \int_{-\infty}^{\infty} S_{iso}(\tau) * lbf(t-\tau)d\tau \qquad [3\text{-}2]$$

Evaluation of this integral without having an expression for S_{iso} can be done in several ways. In the next chapter both a numeric and a matrix solution will be applied, here, however, only the Fourier transform method will be utilized. An expression for convolution in terms of Fourier transform follows:

$$S_{\exp} = Ft^{-1}\left(Ft(S_{iso}) * Ft(lbf)\right) \qquad [3\text{-}3]$$

Of course, we have S_{exp} and S_{iso} as the result of our experiments, it is the linebroadening function (lbf) that we want to obtain. Rearrangement of (3-3) to solve for lbf yields the equation for deconvolution:

$$lbf = Ft^{-1}\left[Ft(S_{\exp}) / Ft(S_{iso})\right] \qquad [3\text{-}4]$$

A Labview program was written to abstract the lbf from our experimental spectra. Labview was chosen because some degree of interactivity is necessary. Should some noise cause $Ft(S_{iso})$ to approach zero, the term in brackets in Equation (3-4) will diverge to infinity. This behavior is illustrated in Figure 3-3. The simplest solution to this problem is to mask out the higher harmonics before the inverse transform[13]; this procedure is equivalent to multiplying by a square function. Multiplication in inverse space is equal to convolution in real space, therefore the resulting lbf is convoluted with a (sin x/)x function (the Fourier transform of a square pulse).

$$Ft^{-1}\left[m_{m=1,t<a}^{m=0,t\geq a}(Ft(S_{\exp}) / Ft(S_{iso}))\right] = lbf \otimes (\sin\theta)/\theta$$
$$\theta \approx fraction\ \ masked * 2\pi \qquad [3\text{-}5]$$

Unfortunately this procedure can lead to distortion of the deconvoluted lbf. An alternative (or additional) method involves adding a constant to the entire $Ft(S_{iso})$ function (this is equivalent to "regularization" in the matrix method). This leads to a (sin x)/x function being added (as a opposed to being convoluted with) to the deconvoluted lbf, Equation 3-6. Because the square pulse is the same width as the spectral window, it has a minimal width (the wider a function is in Fourier space, the narrower it is in real space).

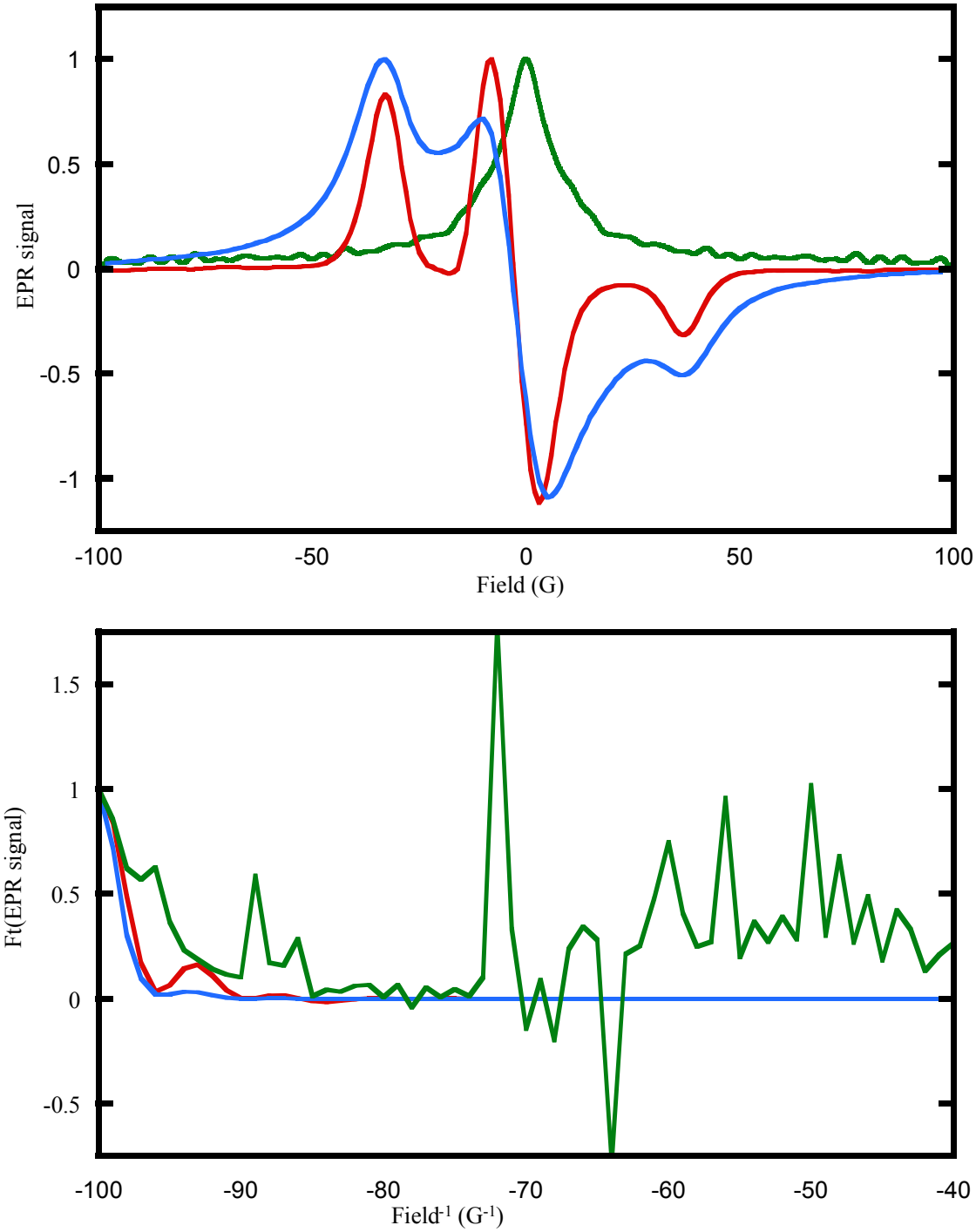


Figure 3-5. Top panel shows EPR spectra S_{iso} (red), 95% partially loaded G(4) (blue) and the lbf that results from their deconvolution (green). Bottom panel shows a portion of the same but in inverse (Fourier) space.

$$Ft^{-1}\left[(Ft(S_{\exp})/(Ft(S_{iso})+a)\right] \cong lbf + \alpha(\sin\theta)/\theta$$
$$\theta \propto width\ of\ spectrum, \quad \alpha \propto a \qquad [3\text{-}6]$$

Unfortunately, because of this (sin x)/x function added to the lbf, deconvolution is only useful for cases where the lbf is broader than this added function. Another disadvantage is the involvement of human judgment in choosing the noise reduction parameters. One advantage is that when the lbf is sufficiently large, it is possible to obtain an authentic line shape that is not constrained to a specific function.

Obtaining the lbf by Fitting

An alternate approach involves selection of a functional form for the lbf, and then simulating the experimental spectrum by convoluting S_{iso} with an lbf of this form . By using some sort of minimization criteria, optimized parameters for the lbf can be found. For our purposes, we found that a pseudo-Voight function, the Pearson VII (Equation 3-7), was able to adequately describe both our deconvoluted lbfs and the results of our simulations (Chapter 4).

$$\frac{1}{\left[1+\left(\frac{2(x-x_o)\sqrt{2^{(1/M)}-1}}{W}\right)^2\right]^M} \qquad [3\text{-}7]$$

The Pearson VII has the advantage of having only two adjustable parameters, full width at half maximum, W, and a shape parameter, M. The shape of the function varies

from Gaussian (M= ∞), lorentzian (M=1), to super-lorentzian (M<1) (Figure 3-5). A Labview program was written (by Robert Usselman) to parametrically vary M and W over a grid of values and convolute the resulting Pearson VII functions with S_{iso} to create a series of broadened spectra that are compared to experimental spectra.

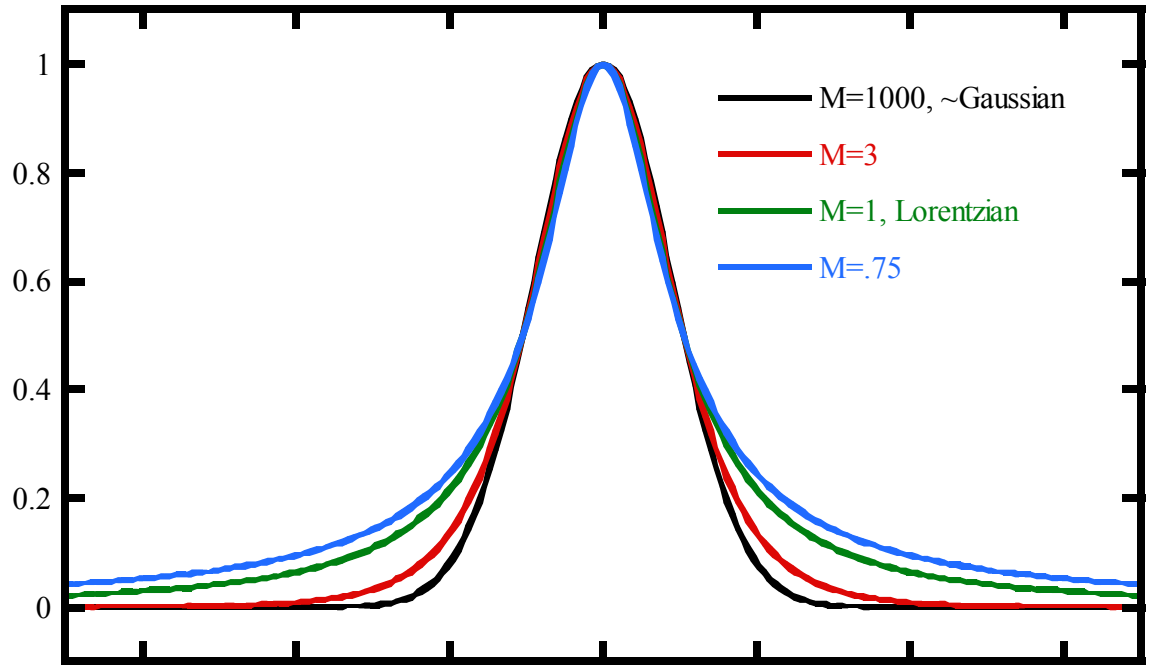


Figure 3-6. Pearson VII functions of constant width (W) and varying shape (M).

Broadened spectra are normalized to have the same double integral as the experimental spectrum and the rms deviation between the experimental and the simulated spectrum is calculated for each. These errors are plotted as a 2-D surface, thus not only can the optimal M and W be found but the surface can be examined to determine the standard deviation. The surface will also indicate if M and W are correlated and if the optimum point is robust.

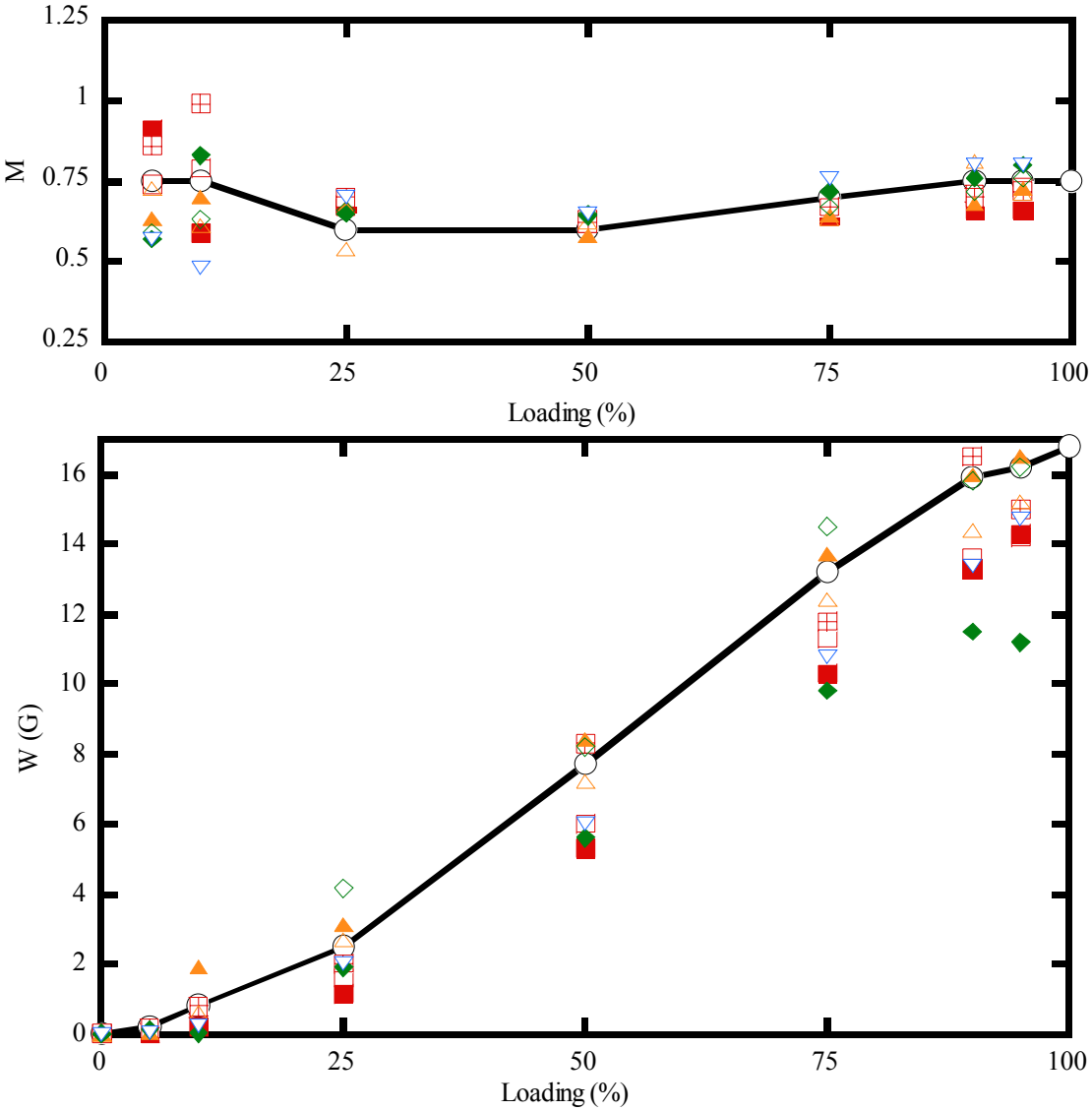


Figure 3-7. Pearson VII parameters for partially and heterogeneously functionalized G(4) PAMAM. Open symbols represent reaction products when TEMPO isothiocyanate was added first, solid symbols for TEMPO added second, open with cross for simultaneous reaction. Color indicates second isothiocyanate (as shown in figure 2-1). None (black), 2,2,6,6 tetramethyl piperidine (red), propanol (orange), phenol (green) and t-butyl (blue).

The optimum values of M and W found in this manner are shown in Figure 3-7 for partially and heterogeneously loaded TEMPO loaded G(4). These spectral parameters will allow us to easily compare experimental data to lbf's from computer simulations.

CHAPTER 4

COMPUTER SIMULATIONS

Spherical Approximation

In order to determine spatial distribution of partially and heterogeneously loaded dendrimers, a series of simulations was undertaken. The goal was to address three important questions: a) what is the spatial distribution of spin-labels attached to PAMAM dendrimers, b) is it distinguishable from other possible distributions and, c) since we are modeling the entire dendrimer, what other physical properties can be determined. If the positions of the functional groups could be modeled, the dipolar lbf (line-broadening function) could be calculated. The technique was developed to specifically to model G(4) PAMAM, in the next chapter it will be expanded to encompass all generations.

As a first step, it was decided to model dendrimer as having its endgroups occupy sites on the surface of a sphere. Several studies have suggested that G(3) and higher PAMAM dendrimers are roughly spherical in solution[51]. This model requires only two adjustable parameters, the sphere diameter, R_{max}, and a hard sphere cutoff distance between sites (or minimum approach) termed R_{min}, the number sites, N, is determined by the dendrimer generation.

There are, however, many arrangements of sites that are compatible with some particular set of R_{max}, r_{min} values. Therefore, a computer program, rms.c, was written (in C) whose output would be a set of x,y,z coordinates, chosen randomly, that are

compatible with a given R_{max}, r_{min} (see Figure 4-1 for pseudo-code, full code is in Appendix B) . A number of these spheres were constructed for each set of R_{max}, r_{min} values (typically 10-20).

Three Models of Inter-Siter Interactions

Each of these model dendrimers could then have a number of sites chosen as those functionalized with a spin-label. Three models of inter-site interactions were used to determine the spatial arrangement of spins, Figure 4-3. A highly positive interaction (or binding cooperativity) would lead to clustering of spins. A highly negative interaction would lead to avoidance of other spins. A lack of any interaction would lead to a random distribution of spins. Three separate programs where written to calculate the dipolar lbf for each loading model. Generic pseudo-code that is applicable to all three is shown in Figure 4-2 (full code for all three programs is in Appendices C, D and E.).

Calculation of an lbf

The programs calculate the dipolar lbf in a manner consistent with the following "rules":

1. Each spin has an lbf.
2. The area of each spin's lbf is the same, regardless of width.
3. In a system with many spins, the lbf of any one spin assumes a Gaussian profile.
4. The width of the Gaussian function associated with a spin can be calculated from the 2nd moment of the dipolar interaction.
5. The 2nd moment of the dipolar interaction for a system of many spins can be

calculated in the method of van Vleck[47], Equation 1-3.

6. lbf's are summed together to get an lbf for the system, but the parameters of lbfs cannot be averaged, only the functions themselves (for instance, a Gaussian function with a width of 5 units added to a Gaussian with a a width of 10 is not a Gaussian function with a width of 7.5, it is no longer a Gaussian at all and its width is much closer to 5 than 10).

```
Start Rmax loop, Step through Rmax values
        Start rmin loop, Step through Rmin values
                Start n loop, step through n values (n = number of spheres to create)
                        While i<#of spins (N)
                                i=i+1 (count # of spins successfully placed)
                                c=c+1 (count number tries)
                                Choose random x,y,z
                                Calculate distance from origin
                                Adjust x,y,z  so that distance from origin = Rmax/2
                                Calculate distance to sites already chosen
                                        If distance is less than Rmin
                                                If c > 1000000 then give up,
                                                report that these Rmax,rmin,N
                                                are incompatible
                                        Discard spin i
                                        Set i to i-1
                                Return to start of i loop
                        End i loop
                        Write text file containing x,y,z coordinates
                End of n loop
        End of rmin loop
End of Rmax loop
```

Figure 4-1. Pseudo-code for rms.c, a C program that chooses sites on the surface of a sphere.

```
Limit = 1000
For p= 2 to N  (N= total # of sites, p= # of sites occupied)
        Limit = 1000/p
        For n = 0 to Limit (this is for random only, cluster and avoidance use all
                possible combinations)
                Order sites for loading model (random, cluster, avoidance)
                Random: ordering just scrambles list
                Cluster: order is determined by distance to first spin
                Avoidance: order maximizes distance to all previous spins
                For i = 1 to p
                        For j = 1 to p
                                Calculate distance between site i and j
                                Store in 2-D array "sum" as follows:
                                Sum [i + (n*p)][1] = sum[i+(n*p)][1] + distance^-6
                        End j loop
                End i loop
        End n loop
End p loop
Take square root of each value in sum
Multiply sum by 19,525 to convert to half width of a Gaussian function (in Gauss)
For k = 2 to N
        For j = -250 to 250 (j steps through x-axis of lbf)
                For i = 0 to 3025 (3025 is # of rows in sum)
                        btemp = value of a Gaussian function at x=j and
                        with a width of sum[i][k]
                        broad[j][k] = broad[j][k] + btemp
                End i loop
        End j loop
End k loop
Write text file containing the array "Broad"
```

Figure 4-2. Pseudo-code for the C programs (ranbroad.c, clustbroad.c and avoidbroad.c) that calculate the dipolar lbf for an arrangement of sites output from rms.c. These were least squares fit to a Pearson VII function.

The results are shown in Table 4-1 for a grid of 10 Å steps in R_{max} and 2 Å steps in r_{min}. The step size was then reduced to 2 Å steps for R_{max} and 1 Å steps in r_{min}. The point of intersection at which both M and W match the experimental results is at R_{max} = 68 Å , r_{min} = 7 Å.

Table 4-1. Pearson VII M and W values calculated on a grid of R_{max} and r_{min}.

		R_{min} (Å)							
		2	**4**	**6**	**8**	**10**	**12**	**14**	**16**
					M values				
R_{max} (Å)	**80**	0.55	0.68	0.805	1.09	1.76	3	5	
	70	0.645	0.67	0.93	1.36	3.4	5.8	14.6	18
	60	0.6	0.666	1.15	2.3	3.84	8.5		
	50	0.6	0.8	1.4	3.4	9			
	40	0.59	0.889	2.8	12				
	30	0.6	1						
					W values (G)				
	80	10	8.86	9.7	10.2	9.6	8	6	
	70	12.4	11.94	13.9	14.6	13.6	10.9	9.8	9
	60	20	19.3	25.4	21.9	17.9	15.6		
	50	35	42	36	34	27			
	40	63	73.8	66	53.4				
	30	80	90						

Now all loading levels and all three models can be compared to experimental results. The M and W values from the computer simulations, along with the values from experiment, are shown in Figure 4-4. It can be seen that both parameters need to be compared: the functional form of the M values for cluster and random both match the experiment, while the functional form of the W values for avoidance and random are both close to experiment. The random model is the only one that is a reasonable match to both parameters.

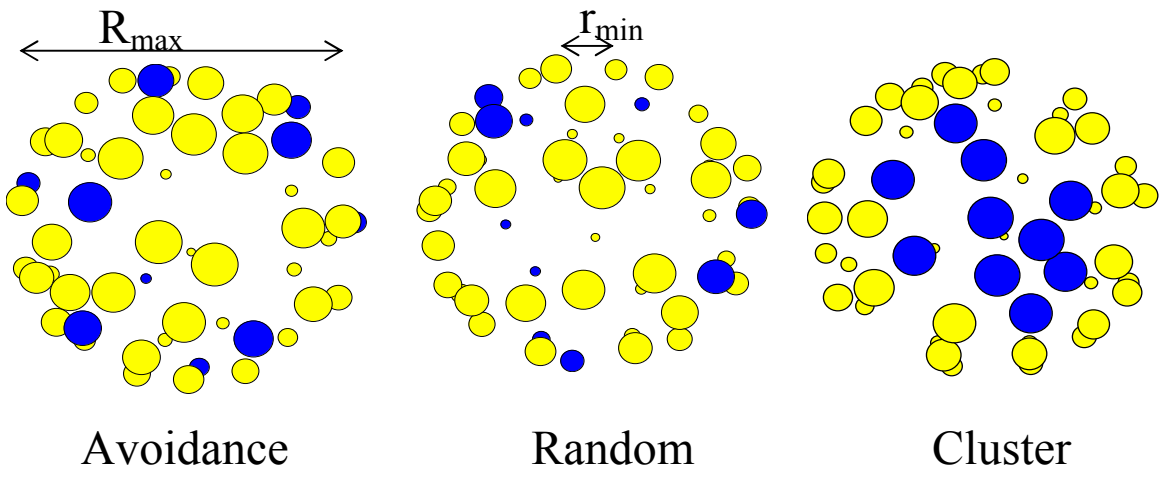


Figure 4-3. Illustration of 3 loading models. Darkened sites show an example of 10 out of 55 sites chosen by the algorithm for each model.

To check that the differences in M and W actually result in meaningful changes in the shape of the spectrum, the lbf's from the simulations were convoluted with S_{iso}, the same spectrum employed in the deconvolution and reconvolution of the experimental data. The A/B ratios of these broadened spectra were measured and plotted (Figure 4-5). It is obvious that there are large differences in the trend with loading and that the random model most closely resembles the experimental data.

Although the random model is a reasonable approximation of the experiment data, there is a definite difference. The random simulation tends be somewhat concave (in either Pearson VII W or A/B), or not as linear as the experimental data.

First, we checked to make sure that is was not an intermediate case, a mixture of random and cluster perhaps. But the addition of cluster to the random model would have the greatest effect at low loadings, whereas the concaveness of the W plot is fairly symmetrical. It was then decided to consider several refinements of the model.

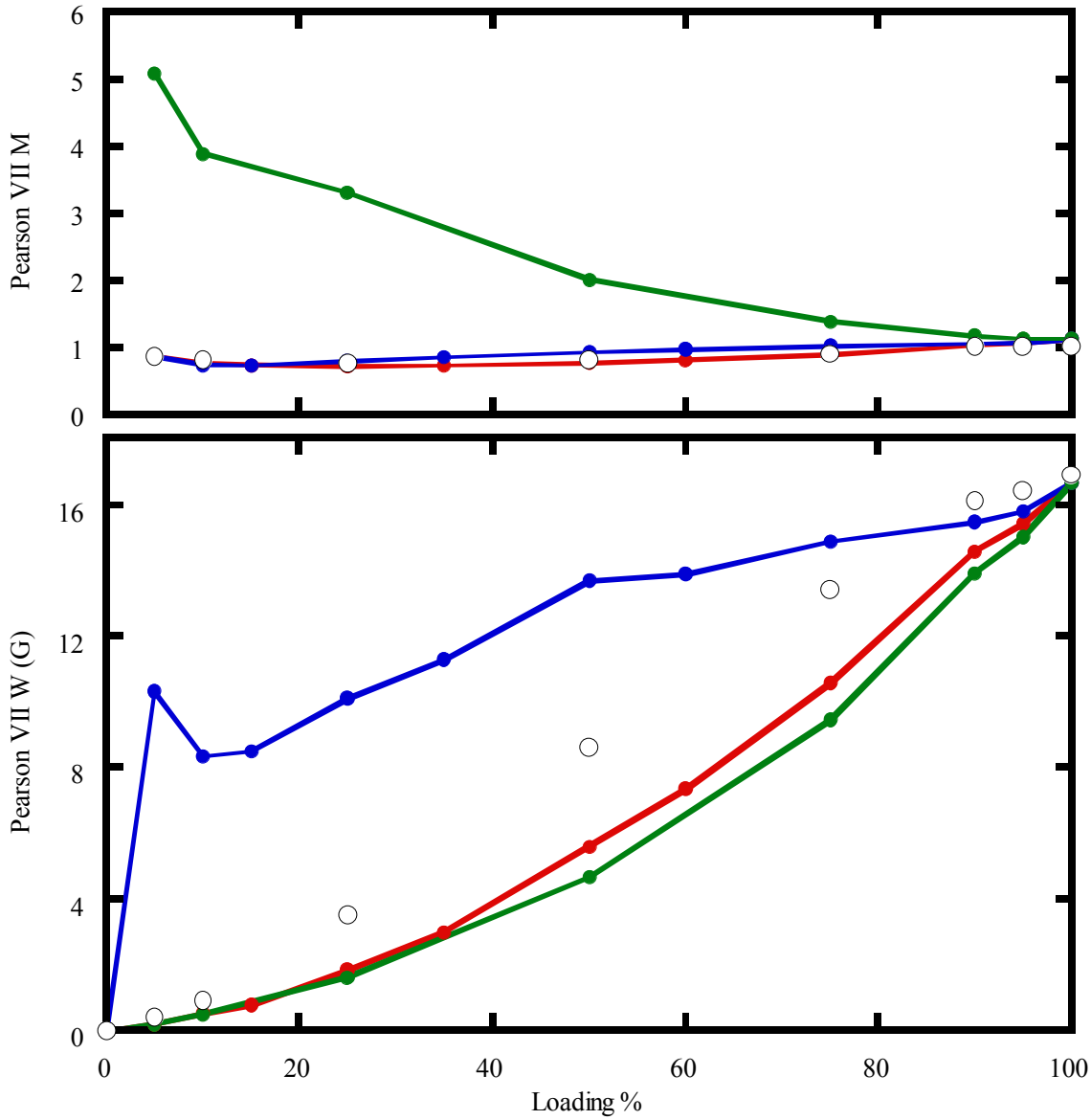


Figure 4-4. Pearson VII M and W values calculated for 55 spins with R_{max} = 68 Å and r_{min} = 7 Å. Cluster model (Blue), Random (Red), Avoidance (Green) and experimental data for partially loaded G(4) (Black circles).

Distribution of Loadings

We know from the MALDI results that there is a distribution of loading numbers associated with an average loading level. It seems reasonable that this distribution would

have an averaging effect on the functional form of the trend of dipolar broadening with loading and would tend to make the trend of W values more linear. An Awk script (Appendix F) was developed to combine the lbf's calculated from different loadings to simulate the effect of a loading distribution. It employs Equation 2-1 to determine the contribution that each loading makes to the lbf for an average loading with a normal distribution. The script also provides the option of scaling the width of the quasi-gaussian function by an arbitrary factor so that the trend with distribution size can be followed.

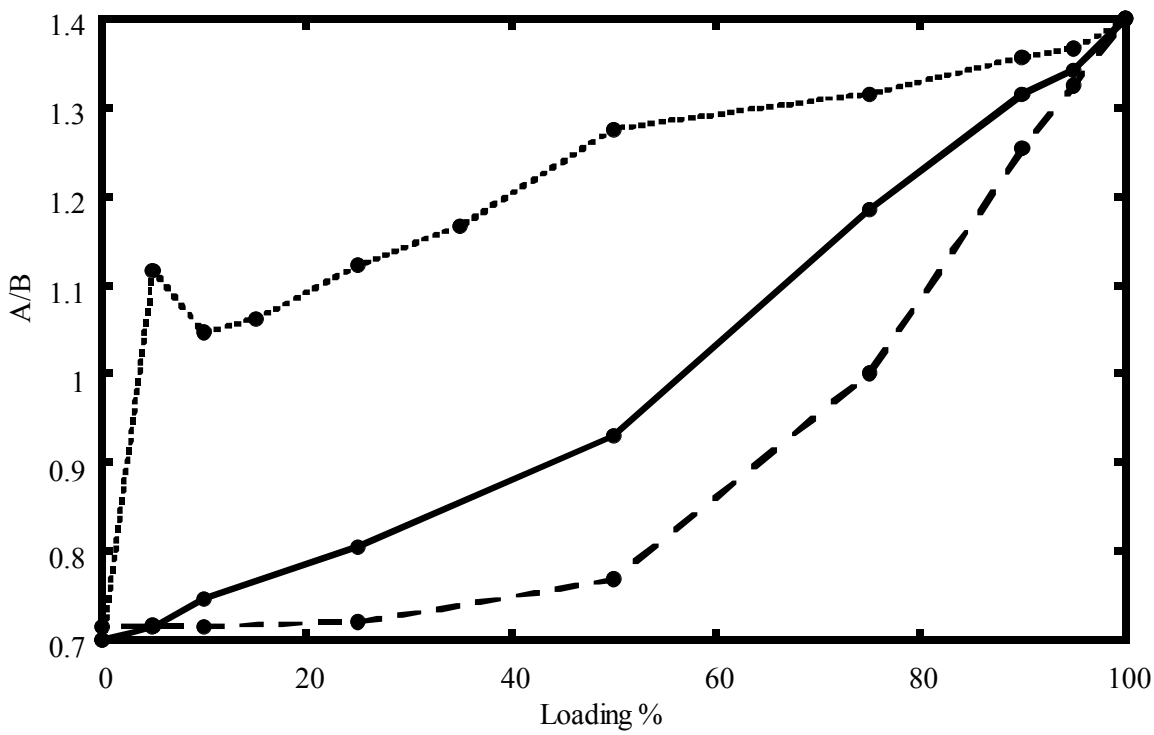


Figure 4-5. A/B ratios for 3 loading models: cluster (dotted line), random (solid line) and avoidance (dashed line).

The results of including the effects of a normal loading distribution are shown in Figure 4-6. A normal distribution has an almost unperceivable effect and, moreover, that effect is toward greater concavity. Wider distributions increase the concavity. While this

result seems counter-intuitive, it must be remembered that lbf's themselves are being averaged, not the parameter values of the functions. When equal amounts of a narrow and a broad lbf are averaged, the resulting function has a FWHM (full width at half maximum) that is closer to the narrow than the broad. This is because, for equal areas, the narrow has a greater height.

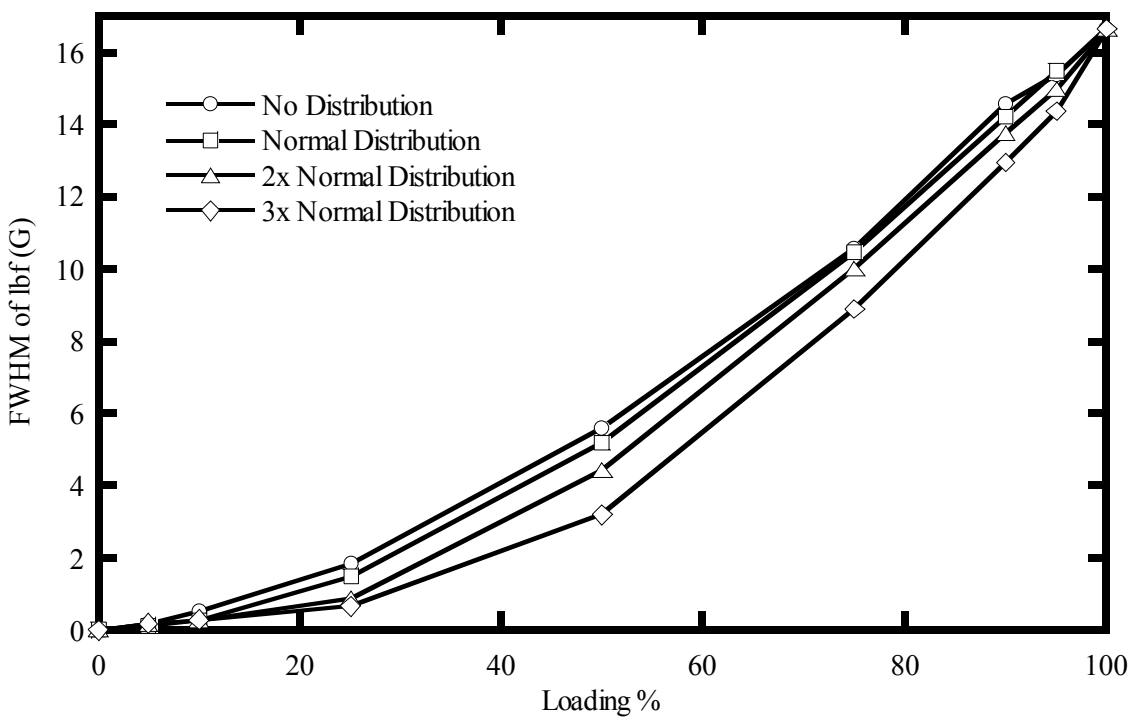


Figure 4-6. Effect of a loading distribution on calculated dipolar linewidths.

<u>Dipolar Hamiltonian</u>

$$\mathcal{H}_{dd} = \{S_{z1}S_{z2} - \tfrac{1}{4}(S_{1+}S_{2-} + S_{1-}S_{2+})\} * T \qquad [4\text{-}1]$$

$$\text{where } T = (g\beta g\beta / r_{ij}^{3})\,(1 - 3\cos^{2}\theta_{ij})$$

Thus far it has been assumed that the broadening of the spectrum could be modeled as convolution of a narrow basis spectrum (S_{iso}) with some line broadening

function (lbf). This requires the lbf to be identical for every point in the spectrum. As detailed below, this assumption is not rigorously correct, although is is a reasonalble approximation in many situations. Therefore it was undertaken to a) derive an equation for dipolar broadening that included dependence on spectral position; b) use this function to simulate broadened EPR spectra. Equation 4-1 shows the dipolar Hamiltonian with the first two terms of Slichter's "alphabet soup" form of the equation[49]. Equation 4-2 shows the same in matrix form[50].

$$\begin{bmatrix} \frac{E_1 + E_2}{2} + \frac{T}{4} & & & \\ & \frac{E_1 - E_2}{2} - \frac{T}{4} & \frac{T}{4} & \\ & \frac{T}{4} & \frac{-E_1 + E_2}{2} - \frac{T}{4} & \\ & & & \frac{-E_1 - E_2}{2} + \frac{T}{4} \end{bmatrix} \qquad [4\text{-}2]$$

Upon diagonalization, equations for adjacent energy levels can be subtracted to calculate transition energy (Figure 4-7). The transition energies are shown for two limiting cases; E1=E2 or degenerate (in NMR terminology the homonuclear case) and E1-E2>>T (the heteronuclear case). Of course all of coupled spins are electrons in our case, however, depending on their position in the spectrum relative to the electron being broadened, the energy difference can vary from zero to several multiples of T

. In Figure 4-8, Δ is defined as E1-E2, and it plots Δ/T vs. the shift in energy in terms of T. For equal values of T, different spins can have a different probability of having a neighbor at an energy difference greater or lesser than T. The probability to have a

neighbor within some range of Δ/T is proportional to the integral of the spectrum in that range divided by the integral of the entire spectrum. This means that spins in the center of the spectrum are more likely to have neighbors at low values of Δ/T and therefore have an average shift in energy larger than spins near the edge of the spectrum.

Transition		$E_1 = E_2$	$E_1 - E_2 >> T$
$1 \leftrightarrow 2$	$\frac{E_1 + E_2}{2} + \frac{T}{2} - \left\{ \left(\frac{E_1 - E_2}{2} \right)^2 + \left(\frac{T}{4} \right)^2 \right\}^{1/2}$	$E_2 + \frac{1}{4} T$	$E_2 + \frac{1}{2} T$
$1 \leftrightarrow 3$	$\frac{E_1 + E_2}{2} + \frac{T}{2} + \left\{ \left(\frac{E_1 - E_2}{2} \right)^2 + \left(\frac{T}{4} \right)^2 \right\}^{1/2}$	$E_1 + \frac{3}{4} T$	$E_1 + \frac{1}{2} T$
$2 \leftrightarrow 4$	$\frac{E_1 + E_2}{2} - \frac{T}{2} + \left\{ \left(\frac{E_1 - E_2}{2} \right)^2 + \left(\frac{T}{4} \right)^2 \right\}^{1/2}$	$E_1 - \frac{1}{4} T$	$E_1 - \frac{1}{2} T$
$3 \leftrightarrow 4$	$\frac{E_1 + E_2}{2} - \frac{T}{2} - \left\{ \left(\frac{E_1 - E_2}{2} \right)^2 + \left(\frac{T}{4} \right)^2 \right\}^{1/2}$	$E_2 - \frac{3}{4} T$	$E_2 - \frac{1}{2} T$

Figure 4-7. First column show transition, second gives the general equation for the energy of the transition, third column shows the splitting for the degenerate case (forbidden transitions shown in gray) and the fourth column gives the energy for the hetero limiting case.

To see what effect this has on the spectrum, an Awk program was written that calculates the appropriate scaling factor for the dipolar broadening at each point in the spectrum. Although we have made a point of not averaging line-widths, but the functions themselves, in this case we found that the average of the line-widths was essentially identical to the width of the average of the functions.

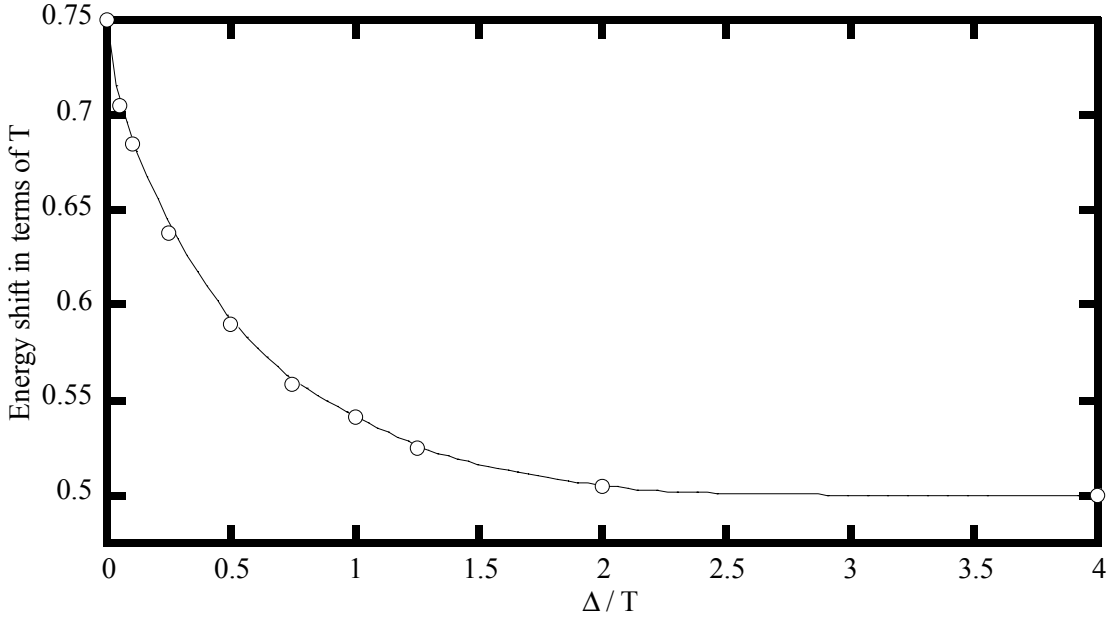


Figure 4-8. Shift in energy in terms of T for various ratios of Δ / T

The program then broadens S_{iso} at each field position by the amount calculated. Full program code is in Appendix G. An example of the resulting spectrum compared to a spectrum broadened in the normal manner is shown in Figure 4-9. Also shown is the width of the broadening function as a function of field. This example was chosen because it is approximates G(4) at 100% TEMPO loading, the experimental conditions where the effect is at a maximum. Deconvolution of the result is an lbf whose width (~16 Gauss) split the difference between the maximum (~17.5 Gauss) and minimum (15 Gauss) width of the convolution function.

In general the values of Δ will be the same for all loadings and the average value of T will increase will loading. At maximum loading the difference is only ~5-10% and would tend add to the concavity of the trend of lbf width with loading. So while this effect may be important for larger values of T, it is not a necessary improvement for our model.

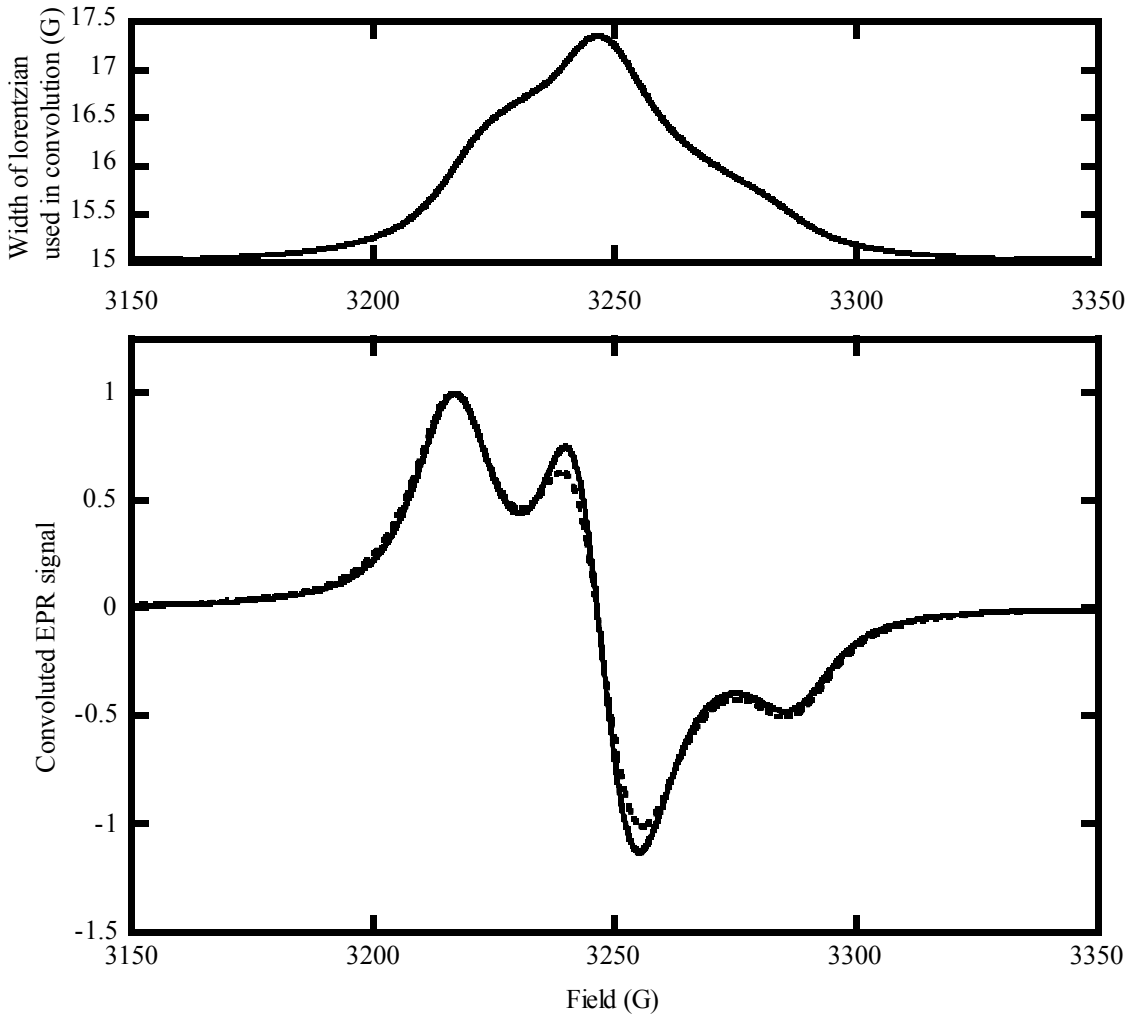


Figure 4-9. Comparison of convoluted spectra when change in the lbf as a function of position in the spectrum is taken into account. Bottom panel: S_{iso} convoluted with a 15 G lorentzian (solid line), S_{iso} convoluted with a lorentzian whose width is shown in the upper panel (dashed line).

Axial Ellipsoid Distortion

Another area where the model could be refined is the shape of the dendrimer itself. It is almost certainly true that dendrimer molecules depart from a spherical shape to some extent, the question is, how much distortion is necessary to have an effect on the dipolar broadening. The first departure from spherical symmetry to be tried was simple

distortion in the z direction: both prolate and oblate axial ellipsoids. A pair of programs (oblate.c and prolate.c) were written to construct coordinate files that could be used to calculate dipolar lbf's with our broadening programs (code is in Appendices H and I). The formula for an axial ellipsoid is shown in Equation 4-3.

$$\frac{x^2}{a^2} + \frac{y^2}{a^2} + \frac{z^2}{b^2} = 1 \qquad [4\text{-}3]$$

The area of the ellipsoids was constrained to be the same as our best fit sphere with R_{max} = 68 Å. Equations 4-4 and 4-5 are the formulas for the area of prolate and oblate ellipsoids, respectively, where e is the eccentricity.

$$\begin{aligned} &if\ \ a > b,\ \ e = (a^2 - b^2)^{1/2} / a \\ &area = 2\pi b(b + a * \sin^{-1}[e]/e) \end{aligned} \qquad [4\text{-}4]$$

$$\begin{aligned} &if\ \ a < b,\ \ e = (b^2 - a^2)^{1/2} / b \\ &area = 2\pi b(b + a * \sinh^{-1}[be/a]/[be/a]) \end{aligned} \qquad [4\text{-}5]$$

In order to maintain a constant area, proportional to a sphere of radius r, while varying the eccentricity, Equations 4-4 and 4-5 were set equal to the formula for the area of a sphere. These formulas were then rearranged to solve for a and b in terms of r and e, and are shown in Equation 4-6 and 4-7. Once a and b are determined, it is necessary to choose random sites on the surface of the ellipsoid. First, x and y coordinates in the range of a to –a and a z coordinate in the range of b to –b are chosen using a random

number generator. This chooses a random point in a rectangular solid that encloses the ellipsoid. To avoid over-populating the areas of the ellipsoid under the corners of the rectangular solid, points outside of the ellipsoid are discarded. A new term, s, is defined as a/b. The chosen point is assumed to lay on the surface of a unique ellipsoid with eccentricity e, and major axis at and minor axis bt, where at<a and bt<b. The x, y, and z coordinates are then adjusted to make at=a and bt=b.

$$for\ a > b$$

$$a = \sqrt{\frac{2r^2}{1 + \frac{1-e^2}{2e}\ln\frac{1+e}{1-e}}}, \quad b = a\sqrt{1-e^2} \qquad [4\text{-}6]$$

$$for\ a < b$$

$$b = \sqrt{\frac{2r^2}{1 + \frac{\sin^{-1} e}{e\sqrt{1-e^2}}}}, \quad a = \frac{b}{\sqrt{1-e^2}} \qquad [4\text{-}7]$$

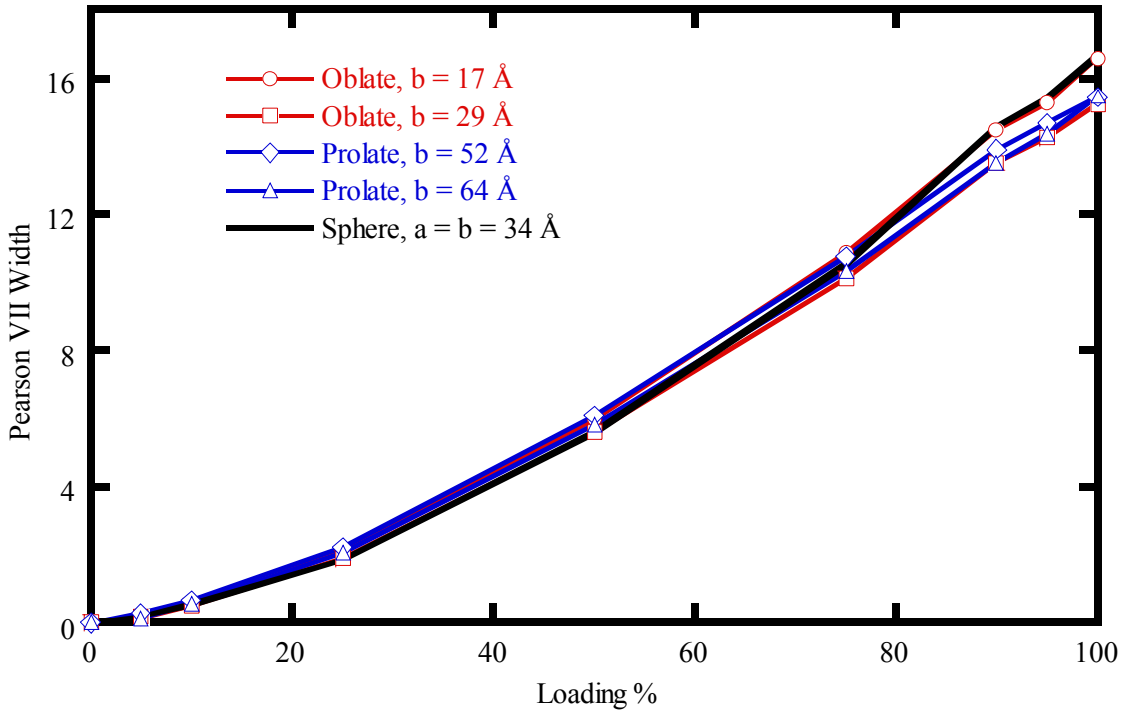


Figure 4-10. Effect of ellipsoid distortion. Axial ellipsoids of area equal to that of a 34 Å radius sphere were used to calculate lbf's. The Pearson VII W parameter is plotted vs. loading for several values of b (z axis dimension).

Dipolar lbf's can be generated from these coordinate files using the same programs (ranbroad.c, clustbroad.c and avoidbroad.c) developed for the spherical coordinate files. Figure 4-7 shows the results for the random case with ellipsoids of area equal to a 68 Å sphere with the z axis approximately doubled (prolate) to approximately halved (oblate) with very little overall effect.

Dimensionality

This stability with regard to a rather large distortion is somewhat puzzling; small changes in R_{max} and r_{min} have a much larger effect. To see if the concaveness of our simulated progression of lbf's with loading was the result of a peculiarity of the

simulation program or had some physical significance, a program to simulate our early test series of frozen solutions of free TEMPO. As was shown in Figure 3-3, this progression seems to be exactly linear. A program was developed to pick random sites in a 3-dimensional volume (full code is in Appendix J). A total of 2000 sites were chosen with the familiar R_{min} and concentration as the adjustable parameters. In order to calculate an lbf, the ranbroad.c program had to be modified (solbroad.c in Appendix K) to avoid edge effects. Only 1/3 of the 2000 spins that were closest to the center of the volume had an lbf calculated for them, however all spins were used in these calculations. The results are shown in Figure 4-11 (bottom panel), and is reassuringly linear.

This result would indicate that the dimensionality of the system is important, therefore, the simulations were expanded into 2-dimensional and 1-dimensional systems. Again, r_{min} and a concentration parameter (sites/$Å^2$ and sites/Å) were adjustable. The results of these simulations are shown in the top panels of Figure 4-11 along with equations fit to the data. The trend is clear: the increase in broadening is proportional to the increase in concentration raised to the 4-(# of dimensions). The curvature of our dendrimer simulations is mathematically correct: the surface of a sphere has the same pair correlation function as a flat 2-dimensional surface.

The experimental data, however, is much more linear. Could the dendrimer have some “3-dimensional” character? There is some evidence for this, it has been suggested that PAMAM generations 0-3 have a tight center and an open exterior, while generations 4 and higher have an open center and a tightly packed exterior[51]. This would suggest that sterics would favor backfolding in the higher generations.

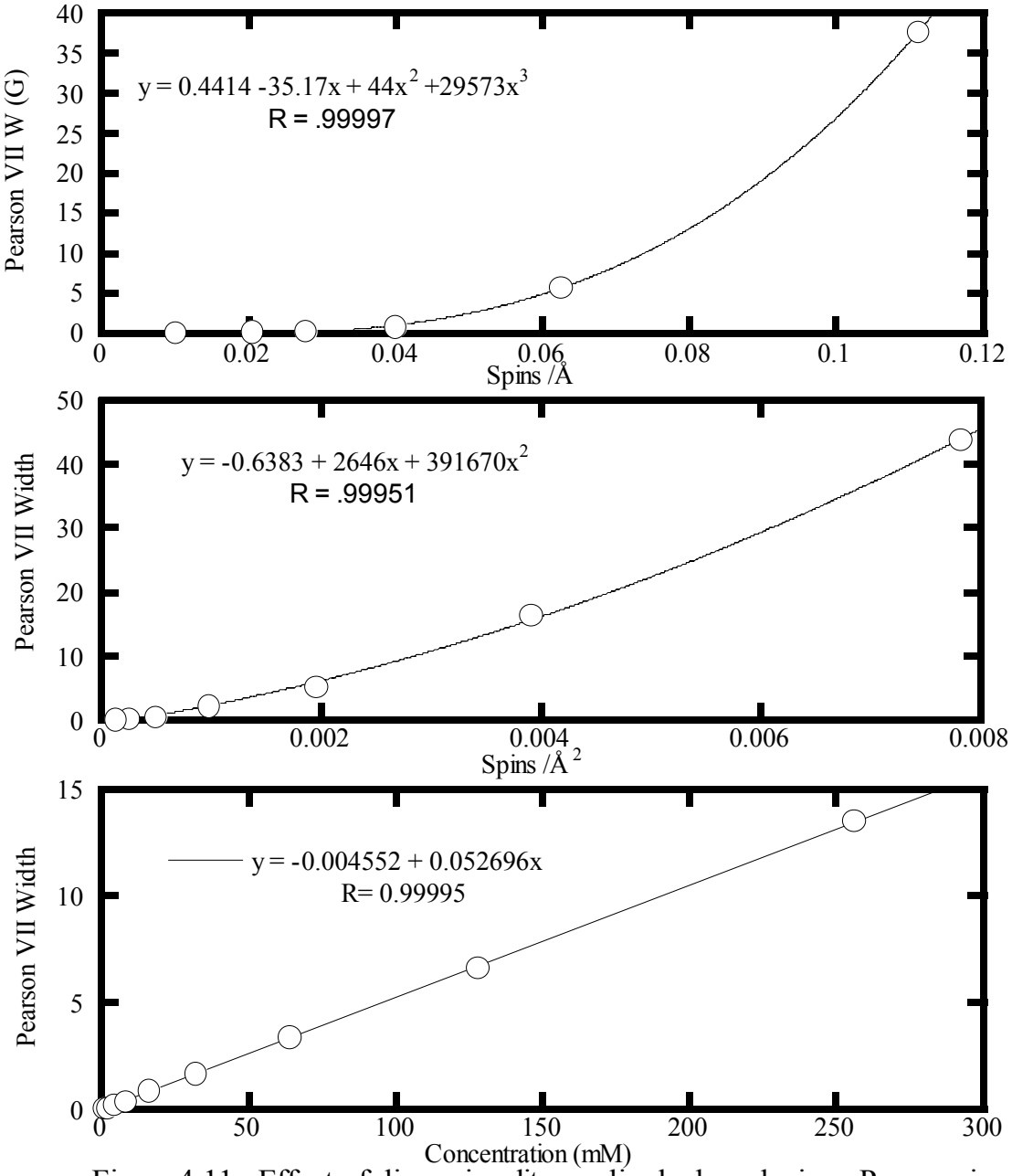


Figure 4-11. Effect of dimensionality on dipolar broadening. Progression with increased concentration for a 1-dimensional arrangement of spins (Top panel), 2-dimensional arrangement of spins (middle panel) and a 3-dimensional arrangement (bottom panel).

Shell Model

The geometry of the model could be modified to account for some backfolding by allowing the spins to occupy a shell of some adjustable thickness. This was accomplished by writing a program that would choose random sites in a spherical shell. Sites were chosen in the following manner: a random number generator was used to pick x,y,z values in the range R_{max} to $-R_{max}$, if $(x^2 + y^2 + z^2)^{1/2} > R_{max}$-shell thickness and $(x^2 + y^2 + z^2)^{1/2} < R_{max}$, and the distance to all other sites is greater than r_{min}, then the site is accepted (full code is in Appendix N). The coordinate files so generated can be used to calculate lbf's in the normal fashion. Figure 4-12 shows the results of these simulations.

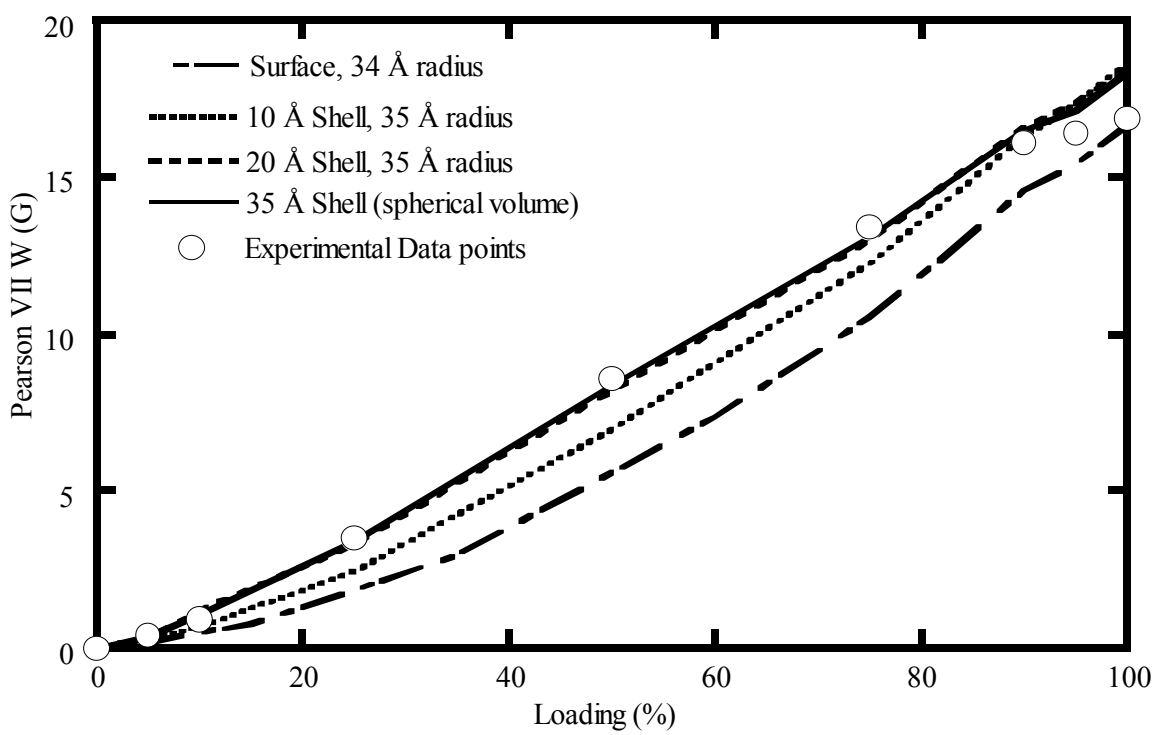


Figure 4-12. Effect of allowing spins to occupy a spherical shell.

Allowing the spins to occupy a greater volume actually increased broadening slightly, this was offset by increasing R_{max} from 68 Å to 70 Å so that the simulations

more closely matched the experimental data. Of all the refinements of the model, the shell ~20 Å thick is the closest to experiment. Is it physically significant, or is there something else that we have not tried yet. One test would be to try other dendrimer generations, the lower generations should have less backfolding, the higher generations more. These experiments are described in the next chapter.

CHAPTER 5

GENERATIONAL TRENDS

EPR Data and Analysis

With a full set of tools developed to analyze generation 4 PAMAM spin-labeled dendrimers, it was undertaken to expand to all generations 0 through 6 inclusive. Synthesis of partially loaded dendrimers was as described in Chapter 2. EPR spectra were collected in the same manner as the G(4) spectra in Chapter 3. Stack plots of the EPR Spectra are shown in Figures 5-1 to 5-7.

A/B Ratios

A/B ratios are plotted in Figure 5-8, G(0) through G(5) seem to be part of a consistent trend: The A/B (and therefore the broadening) is higher for higher generations and the trend with loading is more concave at lower generations. The plot for G(6) cuts across several other generations and seems to be very different, however, when the spectra themselves are examined, it can be seen that the spectra may be broaden to such an extent that the A/B ratio is longer a good parameter. In fact, for the highest loadings, peak B is no longer really a peak.

Pearson VII Parameters

More useful are the Pearson VII parameters for the lbf's (Figure 5-9), determined by the same method used in Chapter 3 for G(4).

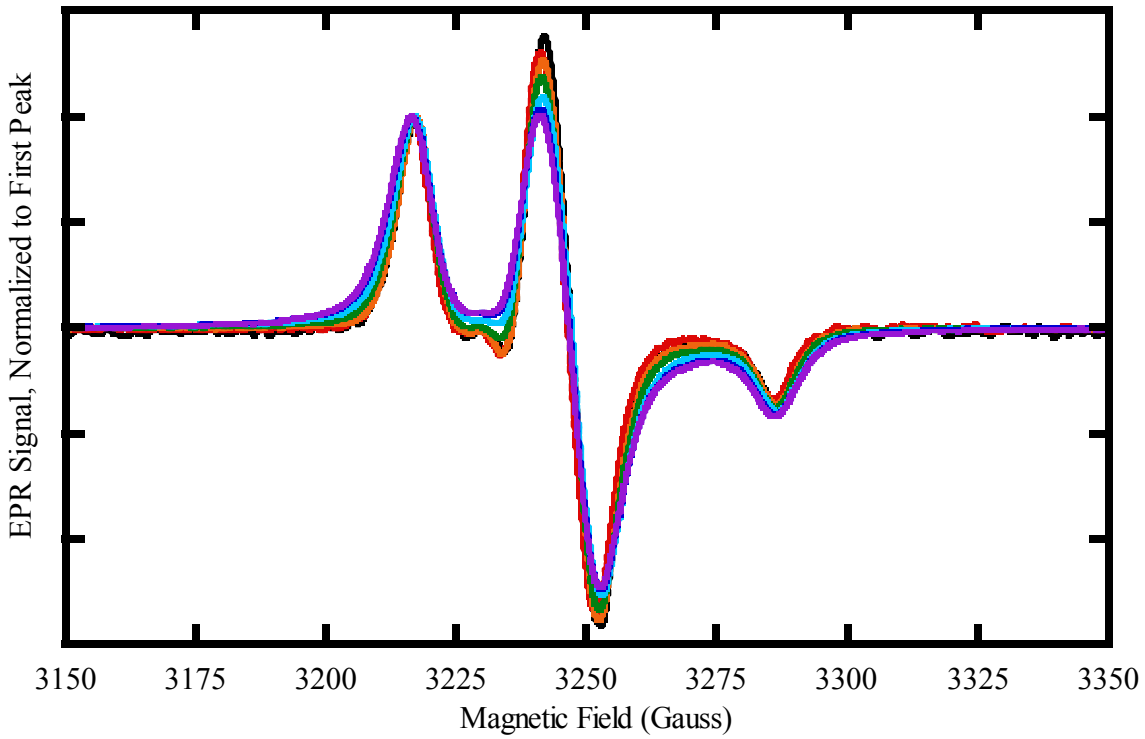


Figure 5-1. EPR spectra of G(0) PAMAM partially loaded with TEMPO. %5 (Black), 10% (Red), 25% (Orange), 50% (Green), 75% (Lt. Blue), 90% (Dk. blue), 95% (Purple).

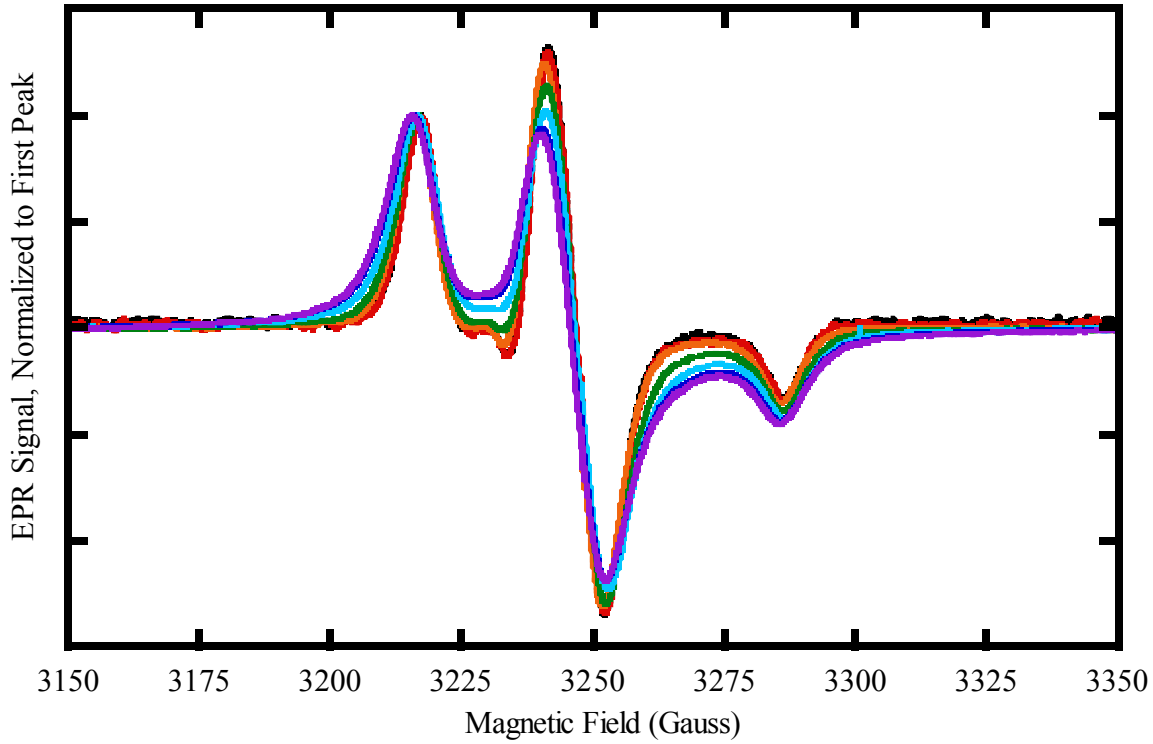


Figure 5-2. EPR spectra of G(1) PAMAM partially loaded with TEMPO. %5 (Black), 10% (Red), 25% (Orange), 50% (Green), 75% (Lt. Blue), 90% (Dk. blue), 95% (Purple).

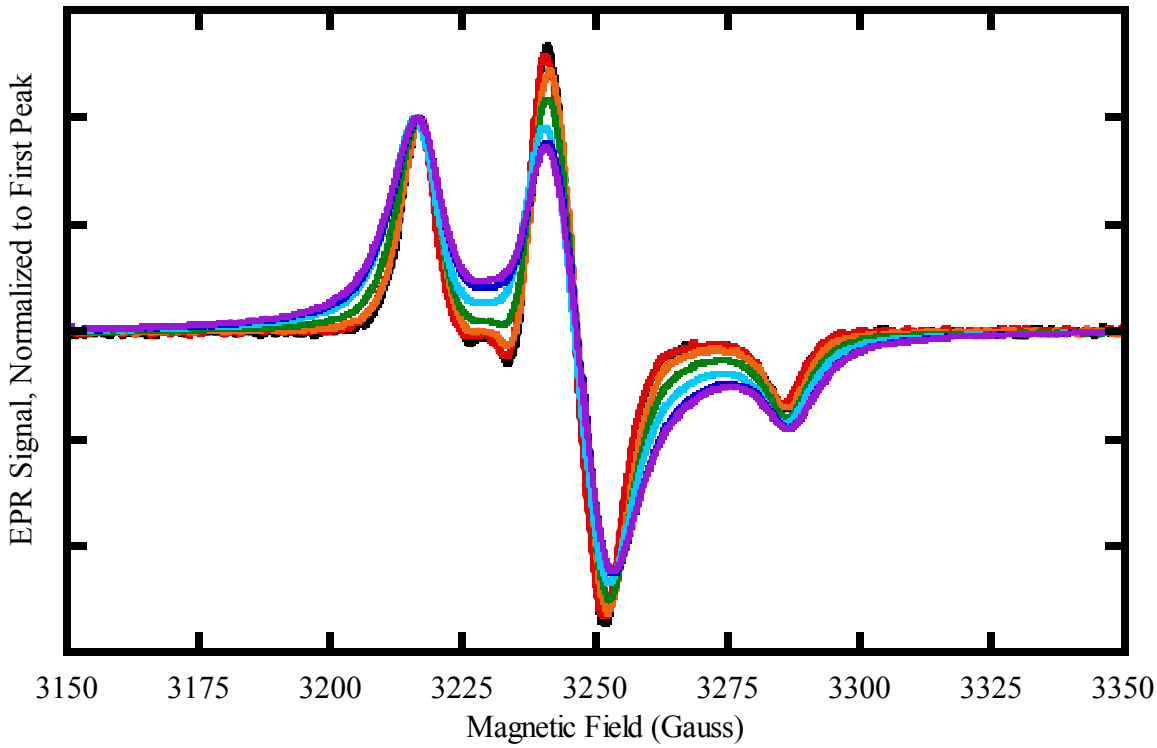


Figure 5-3. EPR spectra of G(2) PAMAM partially loaded with TEMPO. %5 (Black), 10% (Red), 25% (Orange), 50% (Green), 75% (Lt. Blue), 90% (Dk. blue), 95% (Purple).

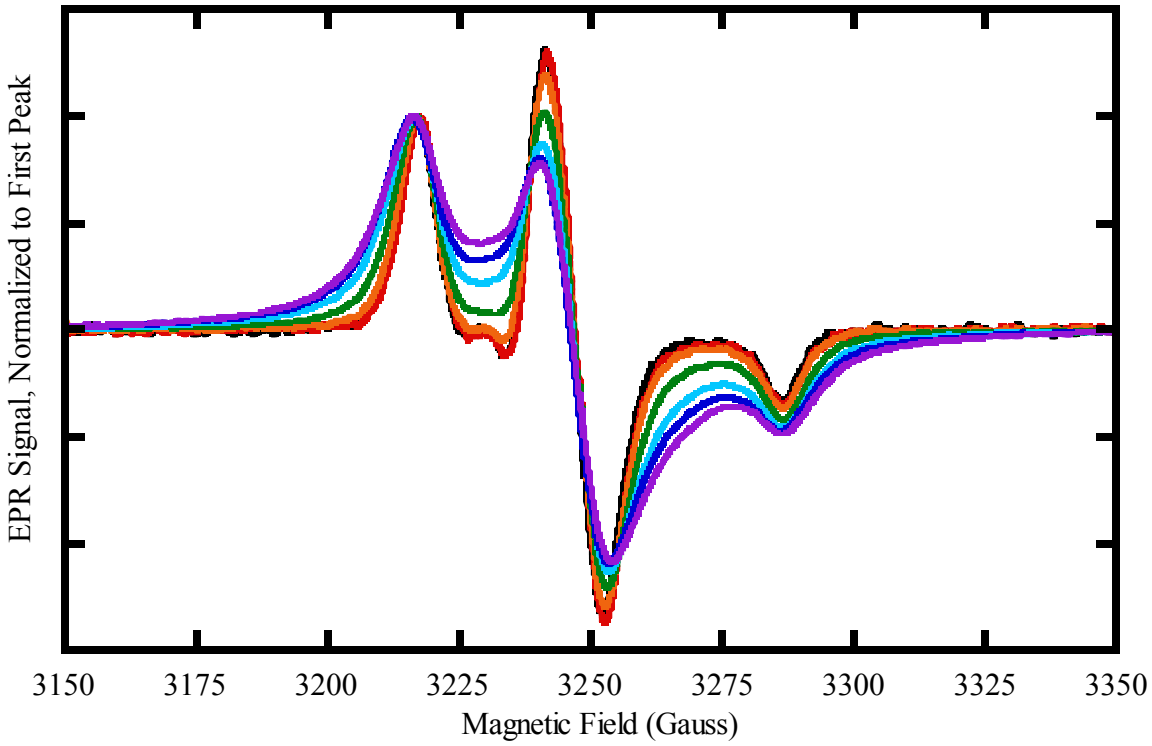


Figure 5-4. EPR spectra of G(3) PAMAM partially loaded with TEMPO. %5 (Black), 10% (Red), 25% (Orange), 50% (Green), 75% (Lt. Blue), 90% (Dk. blue), 95% (Purple).

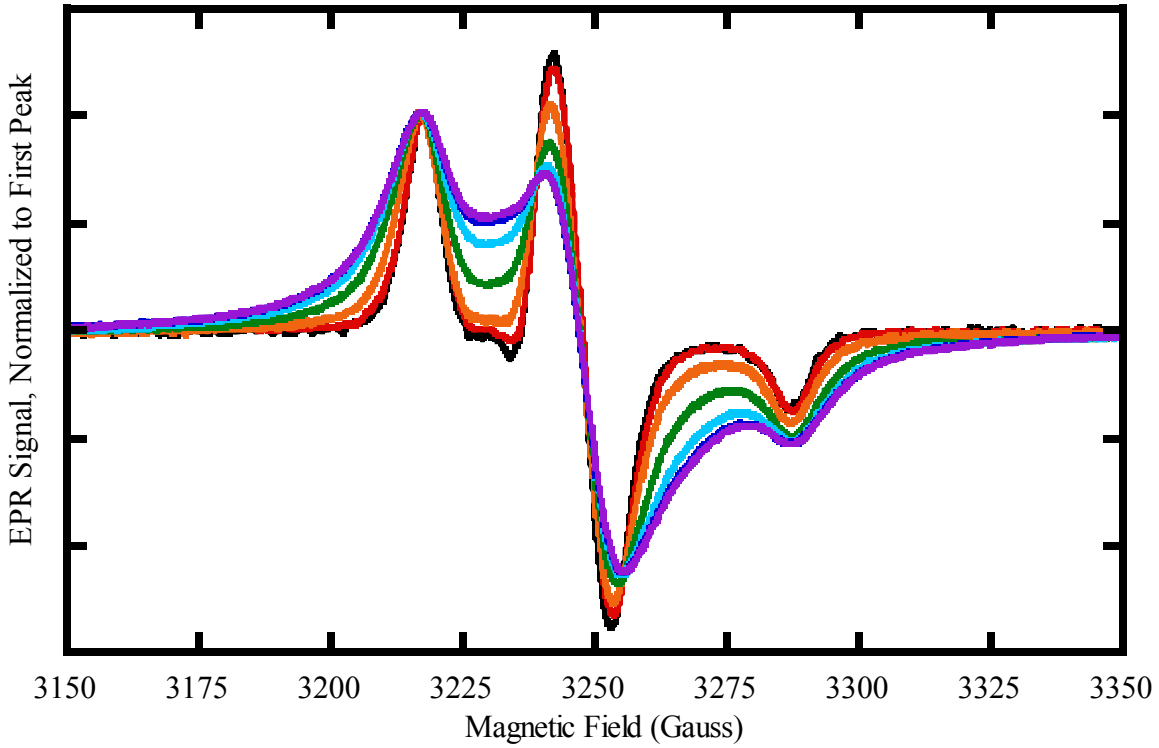


Figure 5-5. EPR spectra of G(4) PAMAM partially loaded with TEMPO. %5 (Black), 10% (Red), 25% (Orange), 50% (Green), 75% (Lt. Blue), 90% (Dk. blue), 95% (Purple).

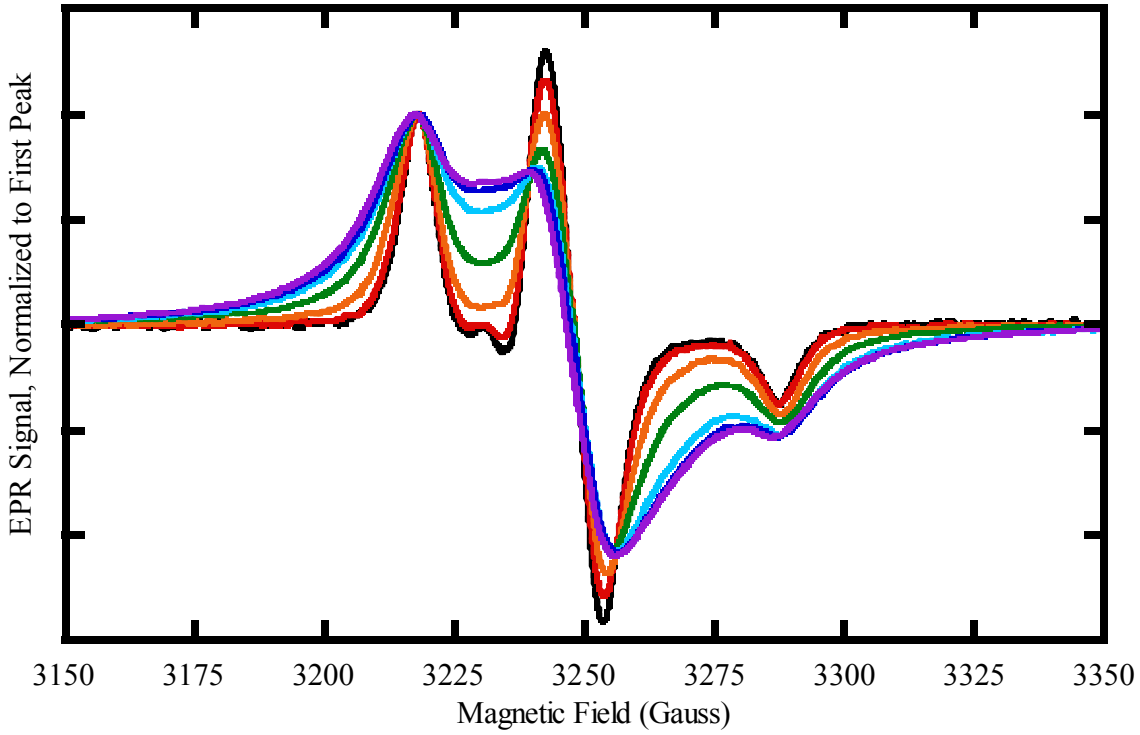


Figure 5-6. EPR spectra of G(5) PAMAM partially loaded with TEMPO. %5 (Black), 10% (Red), 25% (Orange), 50% (Green), 75% (Lt. Blue), 90% (Dk. blue), 95% (Purple).

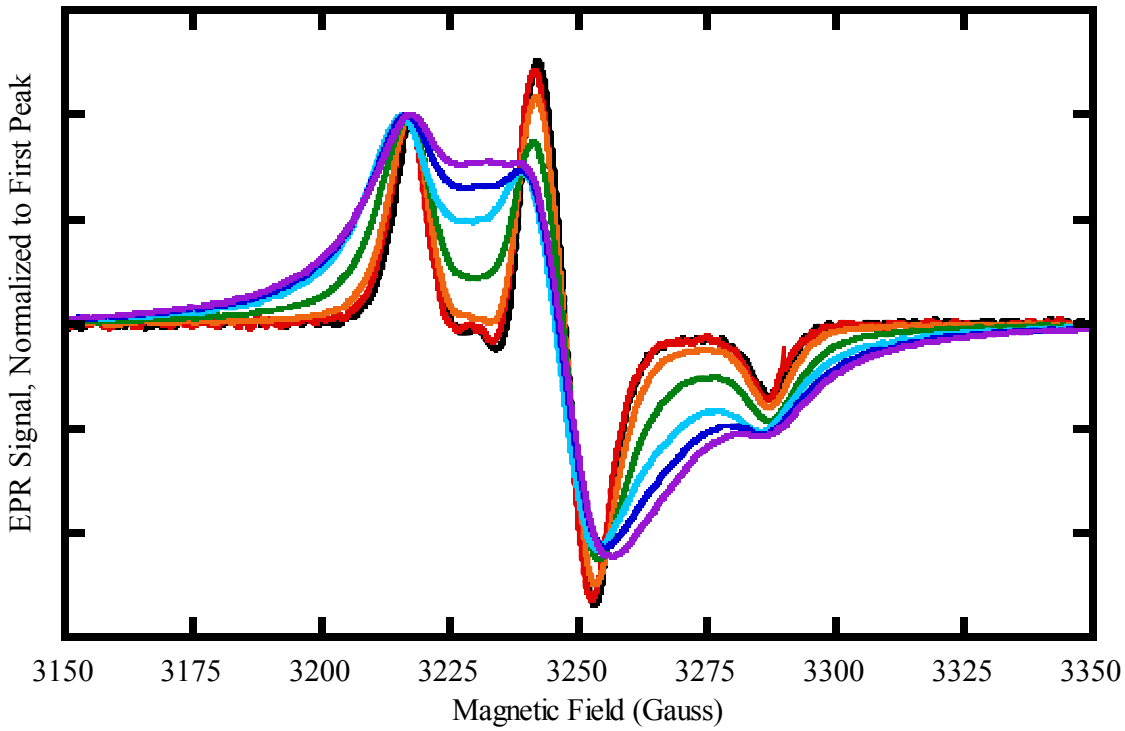


Figure 5-7. EPR spectra of G(6) PAMAM partially loaded with TEMPO. %5 (Black), 10% (Red), 25% (Orange), 50% (Green), 75% (Lt. Blue), 90% (Dk. blue), 95% (Purple).

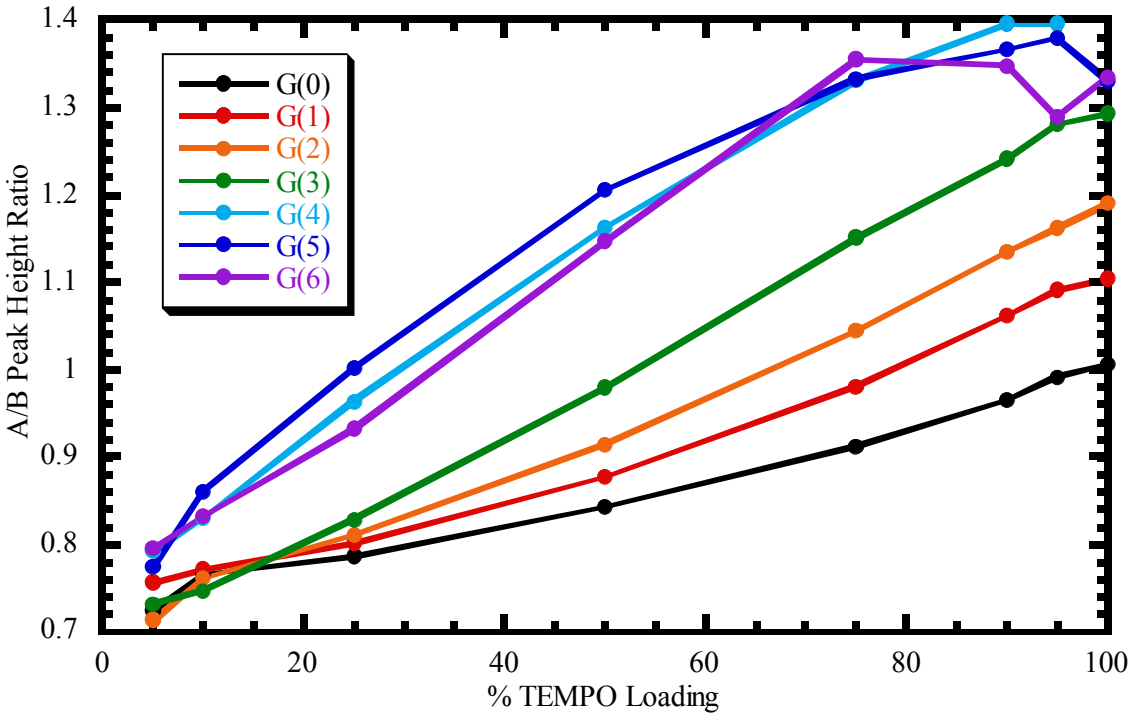


Figure 5-8. A/B ratios for generations 0 through 6 partially loaded with TEMPO. G(0) (Black), G(1)(Red), G(2) (Orange), G(3) (Green), G(4) (Lt. Blue), G(5) (Dk. blue), G(6) (Purple).

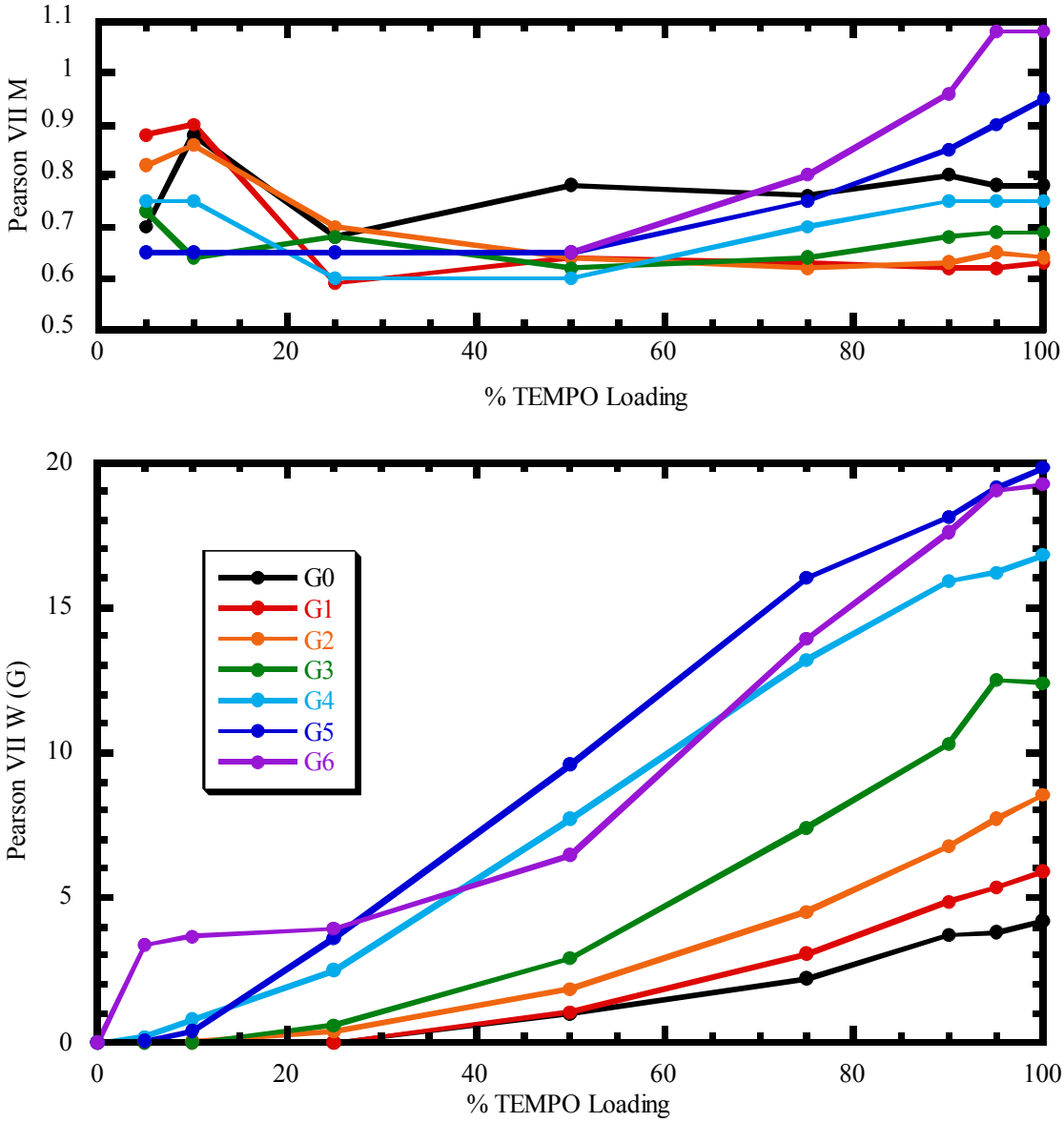


Figure 5-9. Pearson VII parameters best fit to experimental EPR spectra for generations 0 through 6 partially loaded with TEMPO. G(0) (Black), G(1)(Red), G(2) (Orange), G(3) (Green), G(4) (Lt. Blue), G(5) (Dk. blue), G(6) (Purple).

Computer Simulations

Simulations of the lbf's were undertaken, however, the lower generations required a more rigorous approach than was used for G(4). With its large number of spins, the lbf for an individual spin in G(4) can properly be assumed to be a Gaussian function[9].

G(0) on the other hand, has only 2 spins at 50% loading, and the lbf for 2 spins is definitely not a Gaussian. The lbf for 2 spins can be solved analytically from equation 4-1, the shape comes from the term $(1-3\cos^2\theta)$, and is commonly known as a "Pake" pattern[52, 53] (Figure 5-10). Analytic solutions for some specific arrangements of 3 and 4 spins have been reported (3 spins at the vertices of an equilateral triangle, 4 in a square pattern)[26], however, there is not a general solution for an arbitrary arrangement of >2 spins. Equation 4-1, for a specific angle, does not yield a function, but a shift in energy (or field), additional spins shift the energy further. It is therefore possible to calculate an

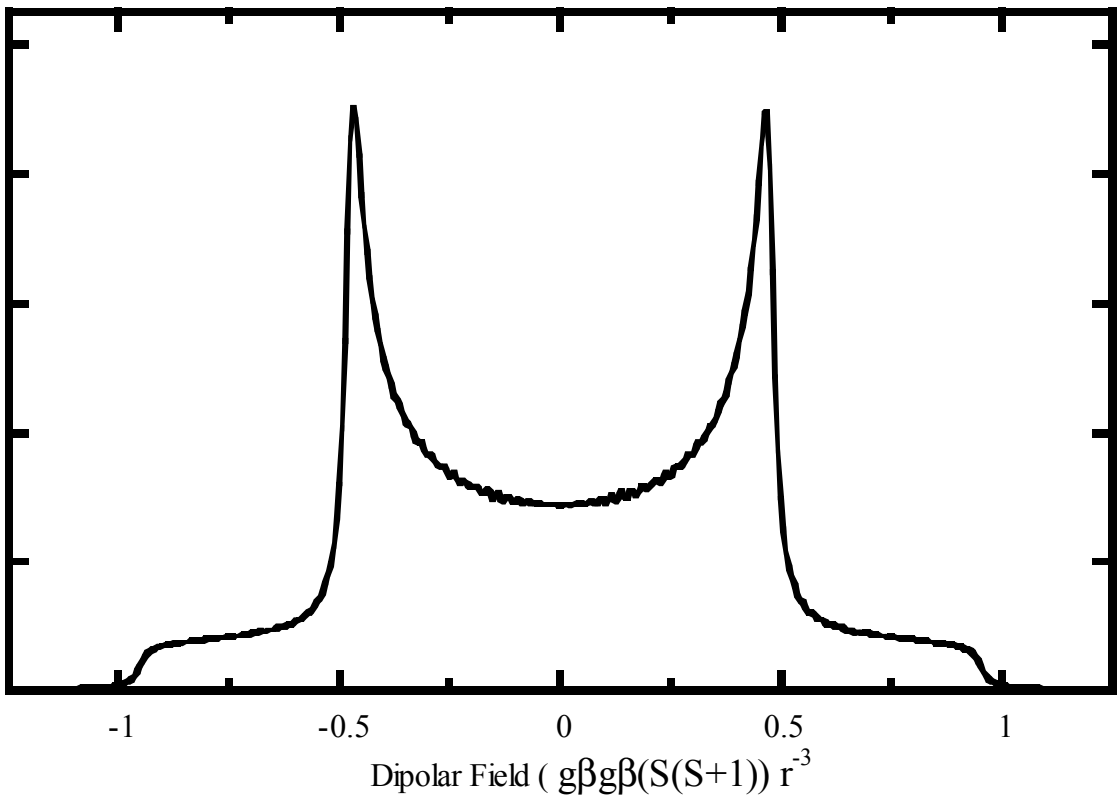


Figure 5-10. A "Pake pattern" for "unlike" spins, function would be broader by 3/2 for "like" spins.

lbf by stepping through angles, calculating a splitting, and summing up the splittings into a line-shape.

A program was written to calculate an lbf from these calculations of repeated splittings. Because of the increased complexity of the calculation, the program was written in C++ (algorithm and pseudo-code developed by Eric Walter, coded by Alex Boulet). The choosing of the sites on the surface of a sphere and the calculation of an lbf were both done in a single program. Pseudo-code is shown in Figure 5-11 (full code is in Appendices O, P and Q). Figure 5-12 shows the results of the simulations for 2 spins with different numbers of angles and configurations.

```
Choose random sites on a sphere compatible with a given Rmax and rmin.
Choose a field angle, (step through z, theta)
        Choose first spin, i
                For each other spin, j, calculate angle that the inter-spin vector
                makes with the field angle
                Calculate splitting as a function of angle and rij
                Round splitting to nearest field step
                Step through array containing previous splittings
                        For each value at a field point, write a point in a new array
                        of half the amplitude and at plus and minus the current
                        splitting, if there is already a point at this field step, then
                        add amplitudes.
                Replace old array with new array
                Next j
        Add splitting array to lbf total array for # of spins j+1
next field angle
```

Figure 5-11. Pseudo-code for repeated splittings program.

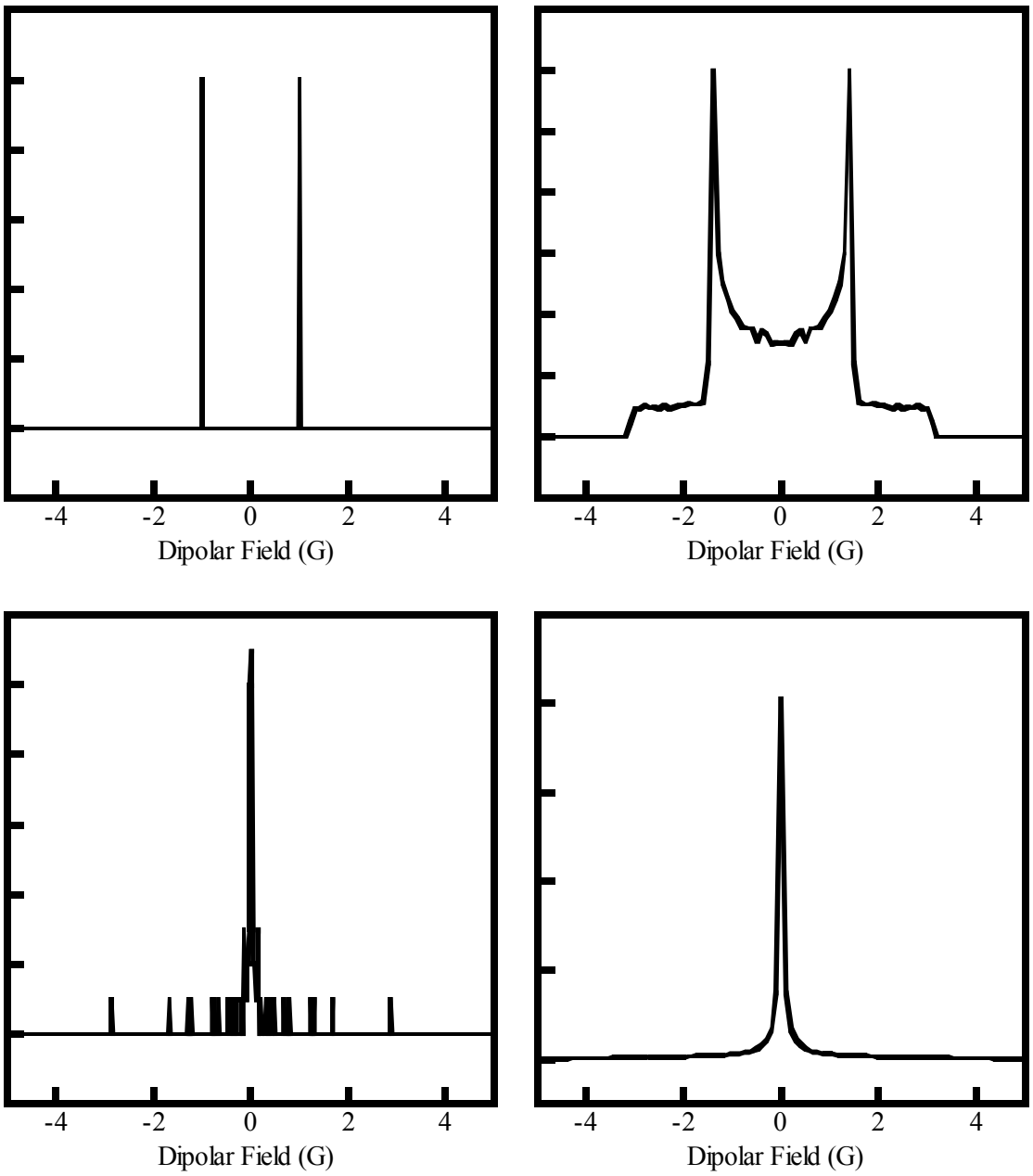


Figure 5-12. Repeated splittings simulations for 2 spins. Top left, 2 spins, 1 field angle, 1 r_{ij}, Top right, 2 spins, 40,000 field orientations, 1 r_{ij}, Bottom left, 2 spins, 1 field angle, 50 values of r_{ij}, Bottom right, 2 spins, 40,000 field orientations (for each r_{ij}), 21,000 values of r_{ij}.

Comparison between the two methods of lbf calculation was done for various systems. An example calculated functions for each method is shown in Figure 5-13, while table 5-1 compares Rmin, Rmax for different generations of dendrimers.

Table 5-1. Comparison of R_{max}, r_{min} values (in Å) from the two calculation methods.

Method	G(0)		G(1)		G(2)		G(3)		G(4)		G(5)		G(6)	
	R_{max}	r_{min}	R_{max}	r_{min}	R_{max}	r_{min}	R_{max}	r_{min}	R_{max}	r_{min}	R_{max}	r_{min}	R_{max}	r_{min}
Gaussian	24	4	35	4	42	4	55	4	68	7	85	6	120	6
Repeated Splittings	26	6	36	6	48	6	58	6	70	6	n/a		n/a	

While the difference for 55 spins (G(4)) is almost indiscernible, the parameters for the lower generations diverge rather substantially, Table 5-1. The method of repeated splittings was used to compare to experimental data for G(0) through G(3), and the assimilation to a Gaussian method was deemed suitable for G(4) to G(6).

While possibly more accurate, the time required for calculation disqualified the repeated splittings method for N_{spins}>55: calculations times for 55 spins and random loading are ~1 hour for the Gaussian method and ~1 week for the repeated splittings method. The calculation times for the repeated splitting method rises with the square of the number of spins, suggesting calculation times for 100 spins (G(5)) of 3 weeks and 13 weeks for 200 spins (G(6)).

The data generated for G(0)-G(2) seem to be systematically different than experiment for loadings up to ~50%, Figure 5-14. For these lower generations, the distribution of loading has a much greater effect than for the higher generations. As a general rule, the width of the distribution at 50% is about the square root of the total number of sites. We might then look at the loading distributions as errors associated with

our theoretical loading level: for G(4) with 55 spins it would be 50% ±7%, for G(0) with only 4 sites it would be 50% ±25%. Therefore the program developed to sum individual lbf's in a way that simulates a normal distribution of loadings was applied to the simulations of G(0)-G(2).

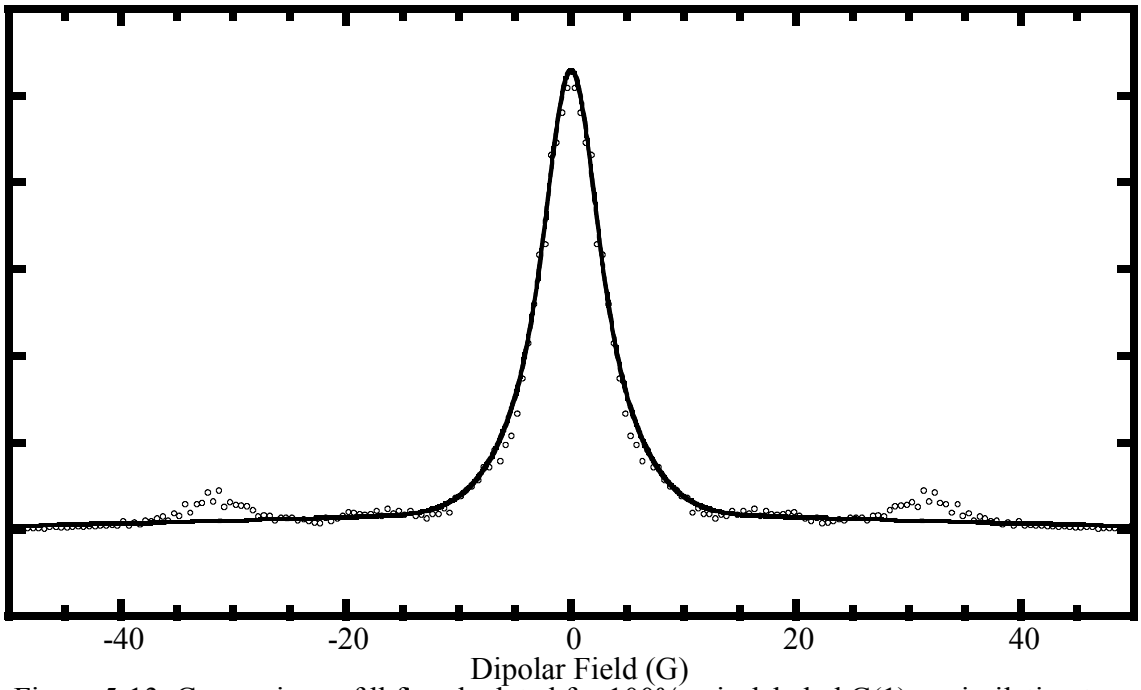


Figure 5-13. Comparison of lbf's calculated for 100% spin-labeled G(1), assimilation to a gaussian (solid line), repeated splittings (open circles).

Simulations with Distribution

As can be seen in Figure 5-14, the addition of a distribution improves the agreement between the simulations and the experimental data, however, the widths of the lbf's for the lowest loadings are still overestimated. This is due to the fact that there is an appreciable percentage of dendrimers with 0 or 1 spins and it is not possible to adjust the lbf to take this into consideration (0 spins do not have an lbf nor even a spectrum, 1 spin would have the same spectrum as S_{iso}, so the lbf would be a delta function).

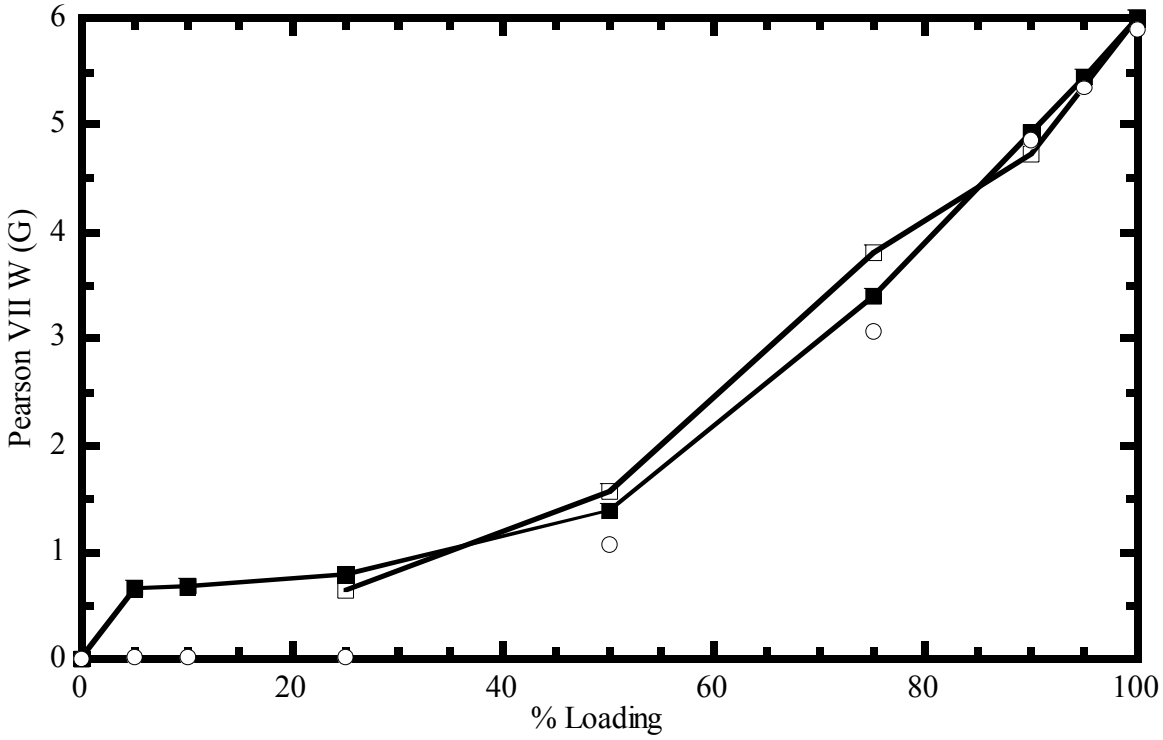


Figure 5-14. Calculation of Dipolar width, with (Solid squares) and without (Open squares) normal distribution, compared with experiment (Open circles) for G(1).

Simulations with the addition of Shell

We know that for G(4) it was necessary to allow the spins to occupy a shell of some thickness, rather than just the surface. Simulations for G(4)-G(6) were done with shell thicknesses varying from 0 to R_{max} /2 in 5 Å steps. Figure 5-15 and 5-16 show data calculated with a range of shell thicknesses compared with the experimental data. While G(4) (Figure 4-11) and G(5) (Figure 5-15) achieve good fits with shell thickness ranging from 10 Å to Rmax/2, the fit for G(6) (Figure 5-16) becomes much worse when the shell is over 15 Å. This is because very little extra volume is added for increasing the shell thickness beyond 20 Å for G(4) and G(5). In the case of G(6), at 15 Å the plot of Pearson VII W vs. loading is already indicative of a 3-dimensional arrangement, more shell thickness merely decreases the effective concentration.

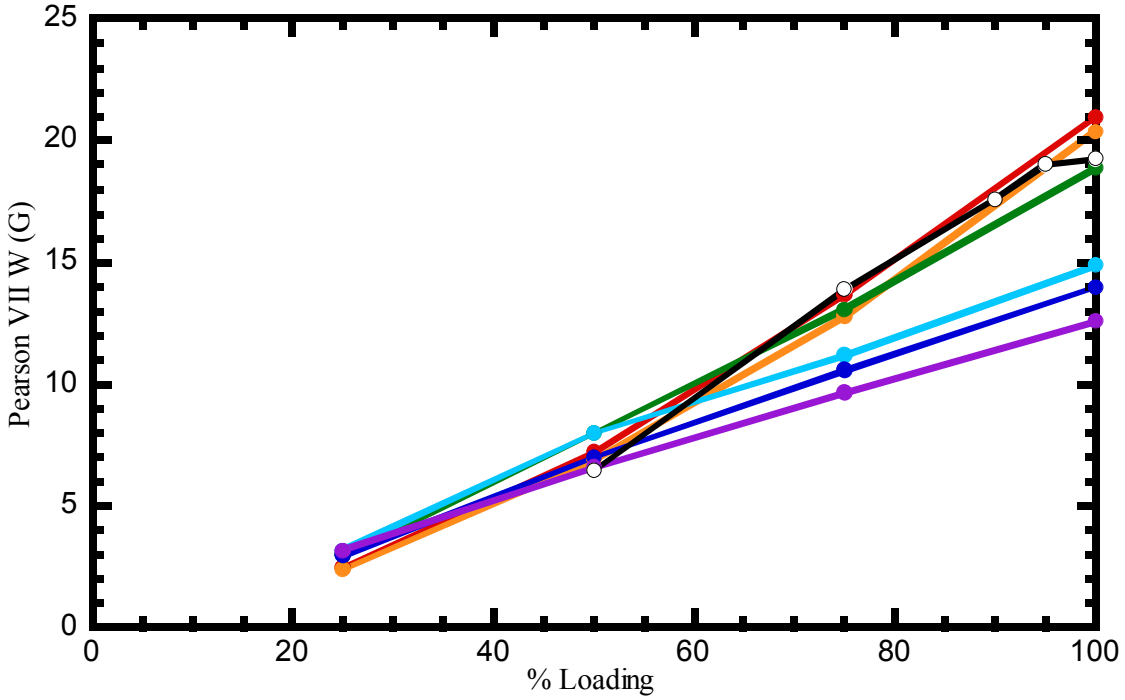


Figure 5-16. Comparison of calculations with different shell thicknesses with data for G(6). 0 Å (Red), 5 Å (Orange), 15 Å (Green), 20 Å (Lt. Blue), 25 Å (Dk. blue), 45 Å (Purple) and experiment (Black)

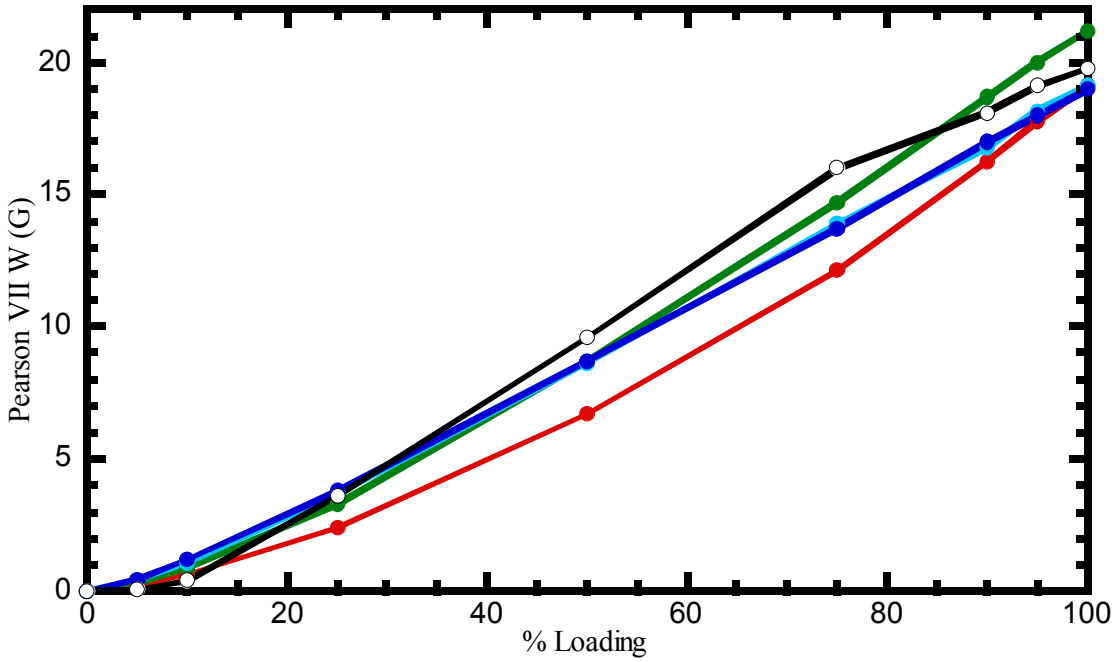


Figure 5-15. Comparison of calculations with different shell thicknesses with data for G(5). 5 Å (Red), 15 Å (Green), 25 Å (Lt. Blue), 35 Å (Dk. blue) and experiment (Black)

Simulations of Hemagutination Results

Results of the interaction of mannose loaded PAMAM dendrimers with the lectin Concanavalin A have been reported by our collaborators[5]. Hemaglutination assays were used to determine the number of Concanavalin A tetramers bound per dendrimer. It was undertaken to simulate the clustering of the lectins around the dendrimer: was the size and number of the lectins compatible with the size of the dendrimer? A very simple model was developed: the dendrimer was modeled as a sphere in the center, with diameter D_d, and the lectins were modeled as N_c spheres tangent to the inner sphere and not overlapping with each other, with a diameter of D_c. This scenario is equivalent to a sphere of diameter D_d+D_c with N_c sites that have an inter-site distance no less than D_c. There is a direct correspondence to the modeling of sites on dendrimer: D_d+D_c is equivalent to R_{max} and D_c is equivalent to R_{min}. The program that was previously used to construct models of dendrimers was easily modified for this purpose. Instead of the output being 20 files containing the coordinates for sites on the surface of a sphere, this program (Appendix R) steps through D_d+D_c and records the number of sites, N_c, successfully chosen with this inter-site distance >Dc. Approximately 40,000 spheres were tried per value of D_d+D_c, for each sphere, there was a total of 1,000,000 attempts at choosing acceptable sites. Because the sites are chosen randomly, we wanted to know if we were using enough configurations to be confident that we had found the maximum value.

Polyhedra

One non-random method of arranging the spheres is to place them at the vertices of a regular polyhedron. There are not regular polyhedra with every possible number of

vertices, but the existing ones allow us to check our computer simulations at various intervals. To calculate minimum D_d for a given D_c and N_c it is necessary to know the ratio (I) of the length of the edge of the polyhedron to the diameter of the circle that circumscribes it. Table 5-2 shows this ratio for the polyhedra used and Equation 5-1 was employed for the calculation of D_d.

$$D_d = (D_c * I) - D_c, \quad I = edge / circumscribed\ diameter \qquad [5\text{-}1]$$

Table 5-2. Ratio of edge length to circumscribed diameter for selected polyhedra.

Polyhedron	# of vertices (N_c)	Ratio of edge to circumscribed diameter (I)
Tetrahedron	4	$4/\sqrt{6}$
Octahedron	6	$4/\sqrt{8}$
Cube	8	$4/\sqrt{12}$
Icosahedron	12	$4/(\sqrt{10}+2\sqrt{5})$
Dodecahedron	20	$4/(\sqrt{3}+\sqrt{15})$
Truncated icosahedron (not a regular polyhedron)	60	$4/\sqrt{(58+18\sqrt{15})}$

Figure 5-18 shows the comparison of the two methods (random and polyhedra). Experimental values for the number of bound Concanavalin A molecules per dendrimer are plotted vs. the calculated diameters from the simulations of the EPR data (adjusted for the difference in the length between TEMPO and mannose). The diameters predicted by simulations of the EPR data are diameters larger than the theoretical diameters (Table 1-1), even allowing for the presence of TEMPO. However, the number of lectins bound per dendrimer match the simulated data with excellent agreement when the EPR predicted diameters of for the dendrimers are used.

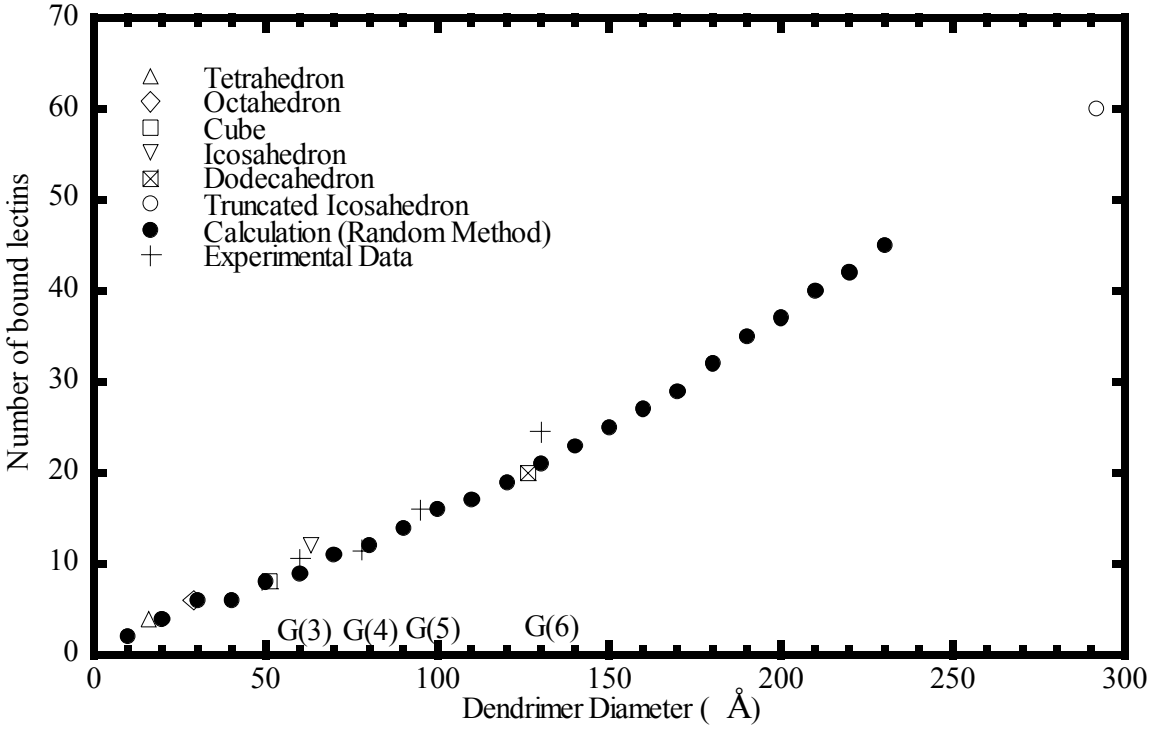


Figure 5-17. Calculation of the possible number of 70 Å spheres tangent to an inner sphere (repesenting lectins around a dendrimer). Experimental data for the number of ConA lectins bound to mannose loaded dendrimers plotted vs. diameter of the dendrimers.

CHAPTER 6

TIME-DOMAIN TECHNIQUES

Introduction

So far, determination of the spatial distribution has only been possible by comparing detailed simulations involving many distributions of distances with experiment spectra. The validity of our results therefore depend on there not existing a whole other set of distributions, consistent with the physical makeup of our system, that give equivalent results. While individual spectra could be the result of many different spatial distributions, our technique of matching the functional form of the broadening vs. loading makes it highly unlikely that there are other, very different, solutions.

It may be experimentally possible to determine what population of spins is at the various levels of dipolar broadening that make up the total lbf. Dipolar broadening affects not only the width of the spectrum, but the relaxation parameters of the system[54, 55]. Time-domain techniques can be tuned to accentuate the spectrum of spins that possess a certain value for relaxation and suppress the spectrum of the other spins. Even when suppression is not complete, sums and differences of spectra can be mathematically manipulated to yield the spectrum only of one type of spin, this is known as spectral editing.

Acquisition of Pulsed Operation at W-band

In order do this type of experiment, pulsed (as opposed to the more common continuous wave CW) techniques had to be employed. Our high frequency (95 GHz)

EPR spectrometer[56] is capable of pulsed operation; however, the electronics to signal it to create microwave pulses and receive data existed at only a very crude level. More complicated experiments require sequences of many pulses at many varying times, therefore computer control is advantageous.

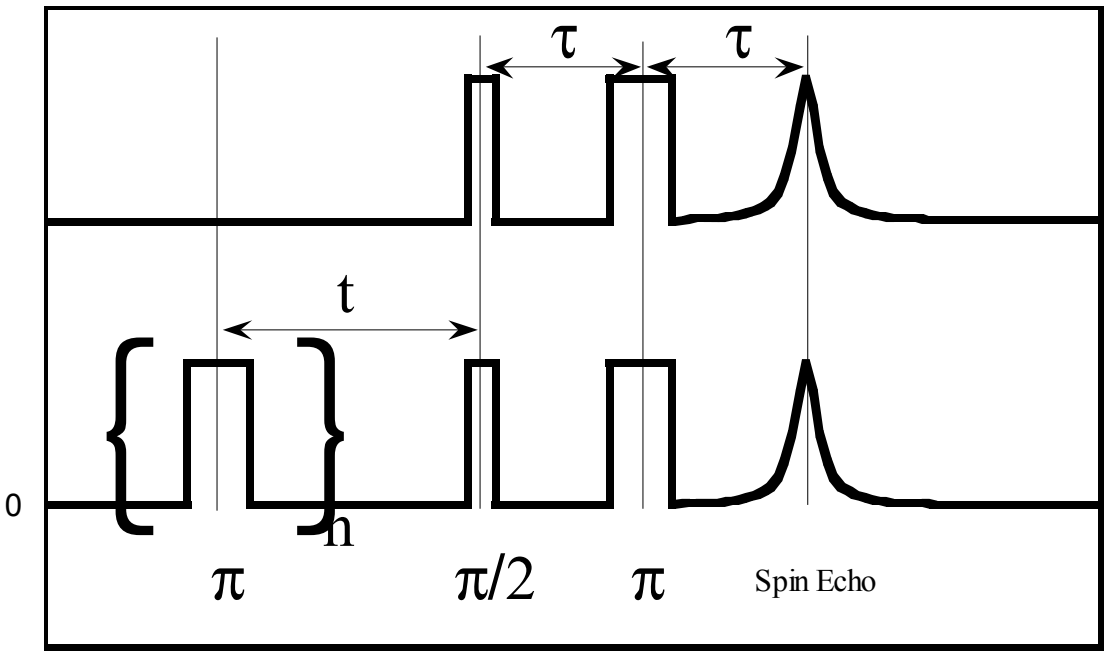


Figure 6-1. Pulse sequences. Top: pulse sequence used for T_2 experiments with t as a variable. Bottom: pulse sequence used for T_1 experiments. First a train of $\pi/2$ pulses (n~500) saturates the system, then a time t is allowed for recovery, after which the same π and $\pi/2$ pulses used in the T_2 experiment are used to probe the system.

The extension of our pulsed EPR capabilities started with the acquisition of a pulse generator, we chose the BNC 555, with 8 output channels and a 1ns time increment, it is controllable by computer through a GPIB interface. The pulse signals from several channels had to be combined with the electronic Or function and output at TTL levels. A circuit board was designed and etched with the help of Mark Reinholz and placed in an enclosure. BNC connectors were used for front panel connectivity.

Pulse sequences, data acquisition and magnetic field were all controlled via Labview programs written by Robert Usselman of which there are four basic types of experiments. There are separate pulse sequences for exploiting T_1 and T_2, shown in Figure 6-1, and there are two types of field variation. In some cases, the field is held at a single value and the pulse timing is varied. What is plotted is the height of the echo vs. the pulse timing (t or τ as used in Figure 6-1). The other mode involves scanning the entire spectrum (in field) at a single set of pulse timing parameters. The output here is a spectrum, whose magnitude and shape may be influenced by the pulse timing.

$$\frac{1}{T_{total}} = \frac{1}{T_{iso}} + \frac{1}{T_{dipole}} \qquad [6\text{-}1]$$

Relaxation Rates

Dipolar coupling has an additive interaction with relaxation rates, as represented in generic terms in Equation 6-1. The value for T_{iso}, the characteristic relaxation time for an isolated spin, is going to vary with temperature. Both T_1 and T_2 for several samples was measured as a function of temperature, the results shown in Figure 6-4. These data were collected at a single field value, the highest point in the spectrum. Figure 6-2 and 6-3 show examples exponential fits of echo decays (T_2) and recoveries (T_1) used to construct Figure 6-4. A double exponential was needed to fit the echo recovery of the highly broadened 75% loaded G(4) sample, therefore two relaxation times are reported for it.

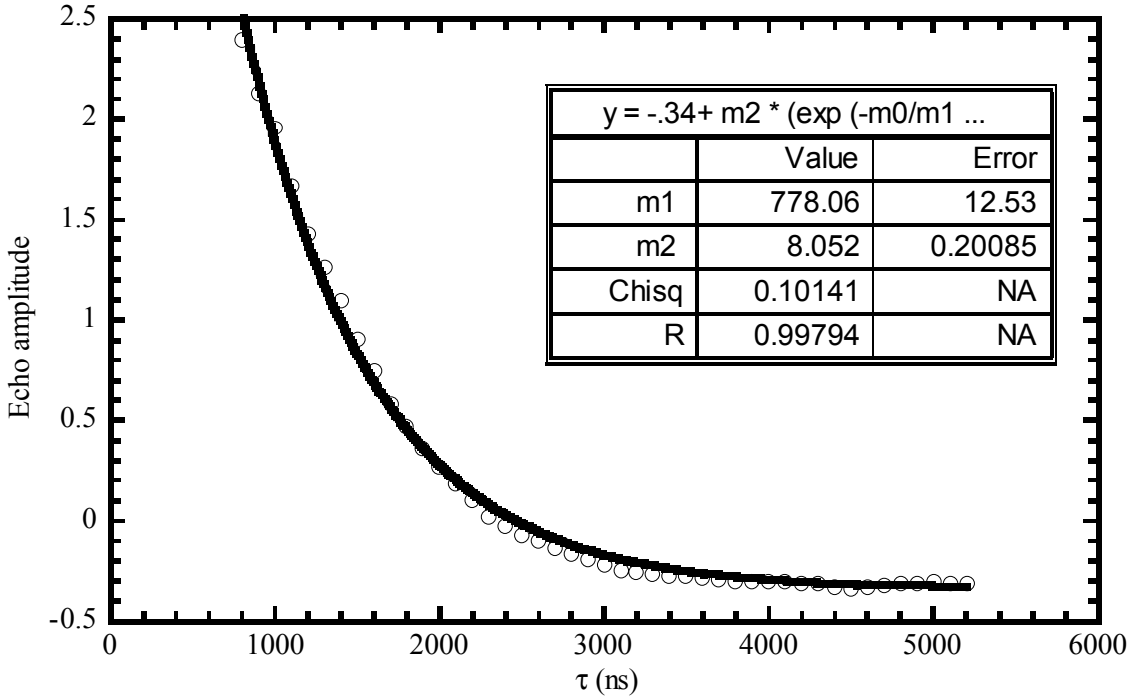


Figure 6-2. Echo decay (T_2). Fit with a single exponetial, sample is a 3mM TEMPO frozen solution (70K).

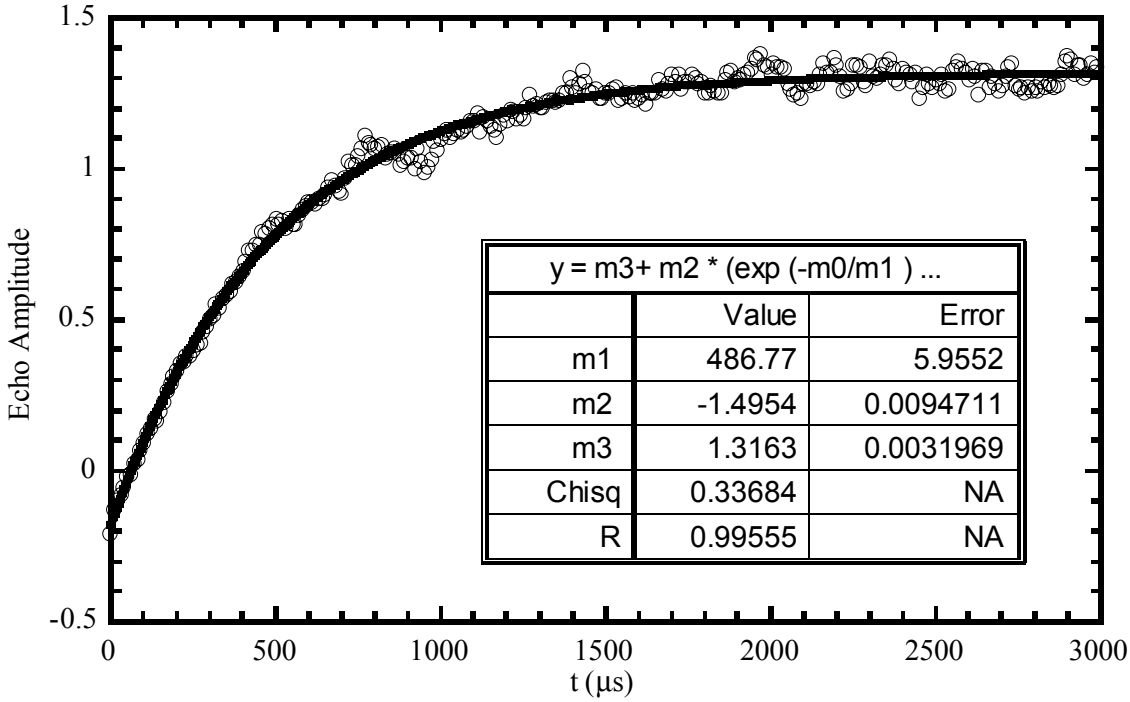


Figure 6-3. Fit of echo recovery (T_1) after a saturating pulse train. Sample is a 3mM TEMPO at 70 K.

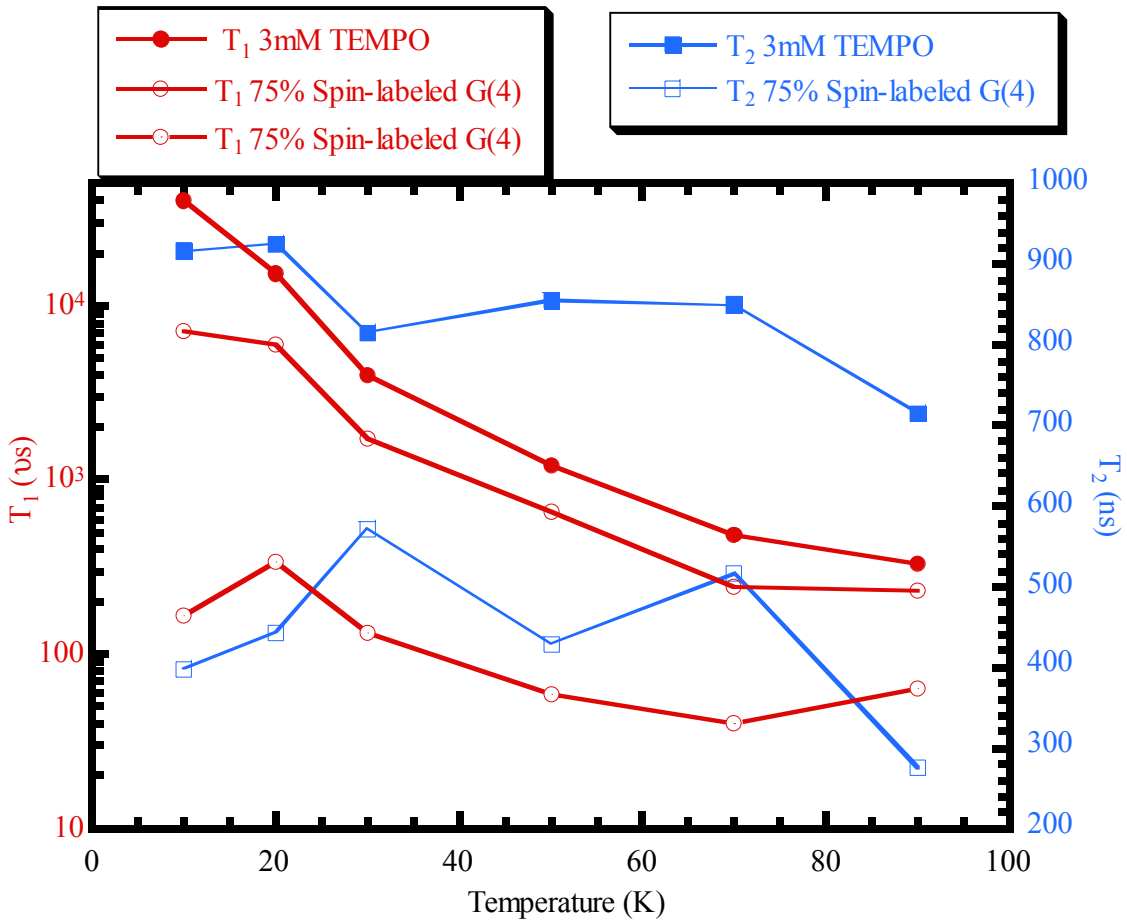


Figure 6-4. T_1 and T_2 variation with temperature. T_1 for 75% spin-labeled G(4) is bi-exponential, therefore two sets of values are shown.

As expected, dipolar broadening has a much greater effect[25] on T_1 than on T_2. This is because $1/T_2$ is much faster than $1/T_1$ and is less effected by the addition of $1/T_{dipole}$. T_1 is also effected more by temperature, from Table 6-1, it would seem that the best temperature to do the experiment would be the lowest. However, very long T_1 values lead to very long experiment times, data is collected as the average many pulse sequences with a delay between of several times T_1.

2-Dimensional Spectra

Analysis of echo decays or recoveries will probably not be an improvement over analysis of dipolar lbf's. A distribution of relaxation times, especially one that is nearly continuous, will also be able to be fit equally well with several distributions of times. It is conceivable that a combination of the two techniques will be an improvement. If the entire spectrum is recorded with each value of t or τ, then the line-broadening at each time point could be determined. Populations of spins with this line-broadening could then be calculated, thus giving a much more exact distribution.

Figure 6-5 shows the spectra taken at different values of t (time after saturating pulse train) for 75% spin-labeled G(4) at 70 K. The same spectra are shown in Figure 6-6, but peak normalized to make any changes in shape apparent. What was expected was a changes in broadening, namely that the spectra would be broadest at short times and gradually narrow. Very little change is seen on the low field side of the spectrum contrasted with the high field side. Close examination of the low field side reveals that the broadest spectrum occurs at the short time and the rest follow in order. The high field side is very different: the broadest spectrum is indeed at the shortest value of t, and width decreases with time at first. But at around 600 μs the high field side reverses direction and increases overall line width. In addition, the overall shape of the spectrum changes, relative intensities of peaks change.

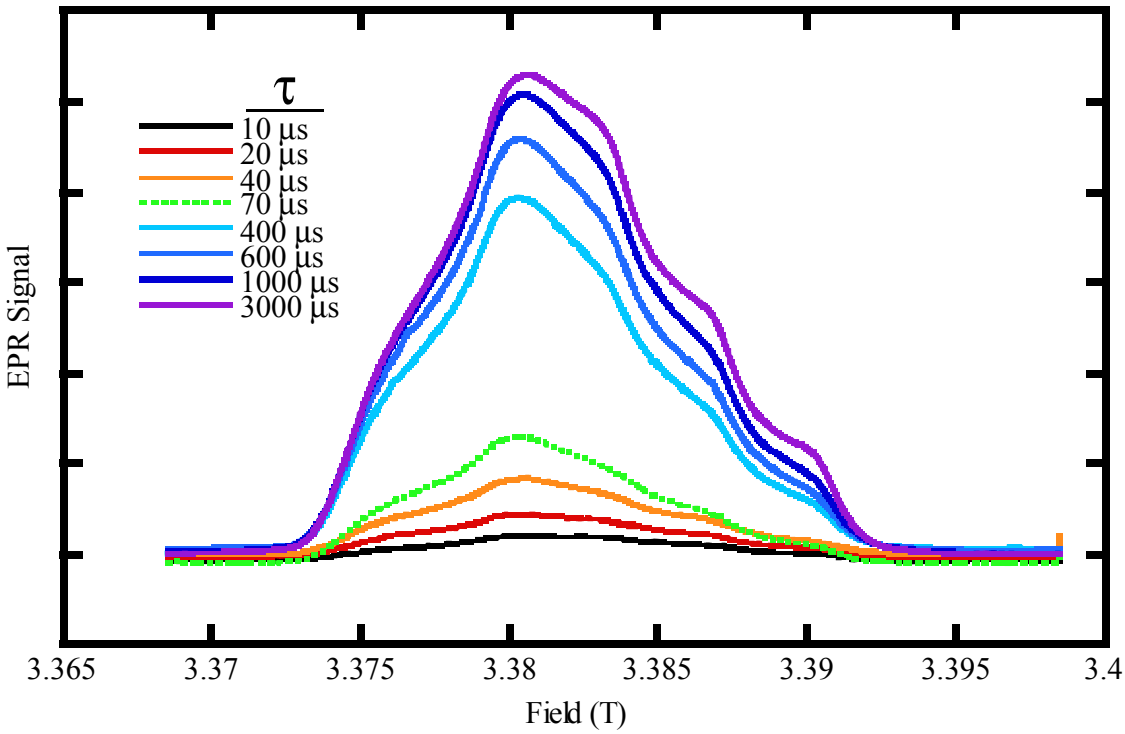


Figure 6-5. Field swept T_1 spectra. Sample is 75% spin-labeled G(4) at 70K.

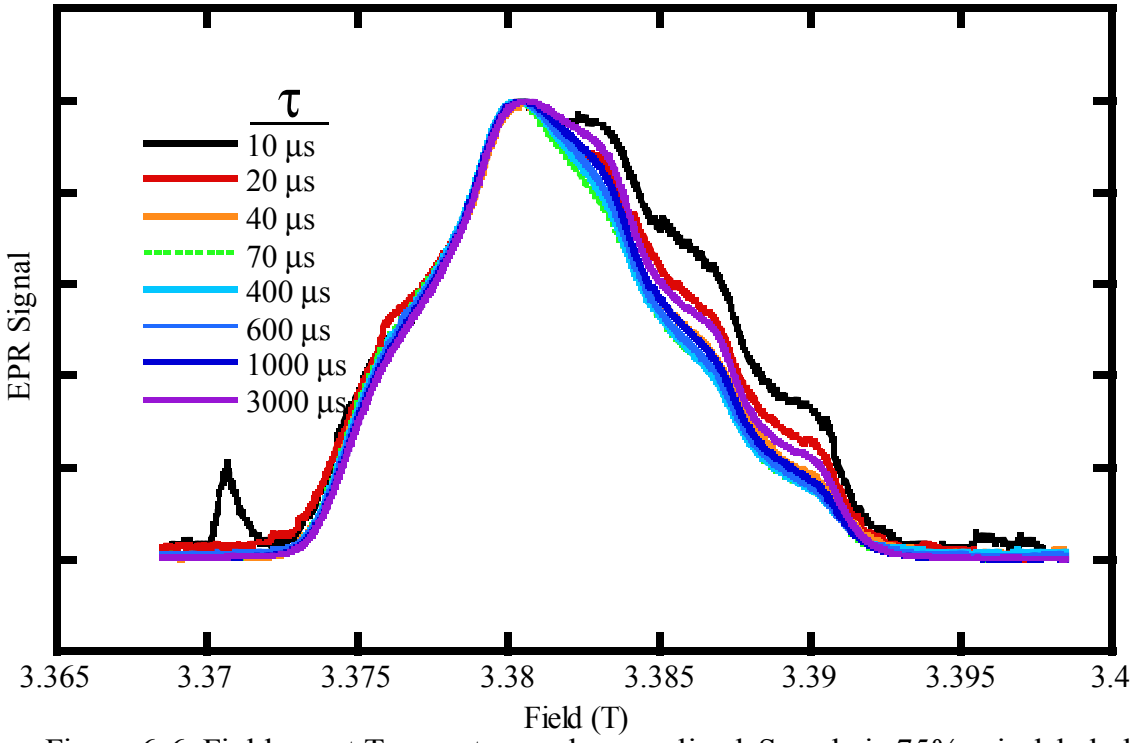


Figure 6-6. Field swept T_1 spectra, peak normalized. Sample is 75% spin-labeled G(4) at 70K.

This is very puzzling, but the solution is simple. For systems with a large amount of dipolar broadening, $T_1$is bi-exponential, this is known from our fitting the data taken at the center of the spectrum. However the values of T_1 vary with position in the spectrum, specifically, the short value for the high field side is shorter than for the rest of the spectrum, and the long value is longer than for the rest of the spectrum. What appear to be large changes in broadening are actually exaggerated by peak normalizing. This phenomenon has recently been reported at W-band for 2-D T_2 spectra[57].

This result means the broadening of the high field spectrum can not be modeled as a convolution. This was already suspected, as we were unable to deconvolve CW W-band spectra. While a change in broadening can be detected on the low field side, it will be difficult to abstract an lbf from only half a spectrum. The W-band spectra have this variation with field because the position in the spectrum is a largely a function of orientation of the molecular axis to the field. Specifically the left, center and right portions of the spectrum correspond to the x, y, and z orientation. At X-band, where all previous work has been done, gross field position (left, center or right) depends on the ^{14}N nuclear spin quantum number and much less on orientation. Because the spectra of spins at different orientations overlap, broadening at X-band can be modeled as a convolution. Therefore it may be possible to obtain the lbf at different time points if the pulsed experiment was done at lower frequency.

Time-domain experiments at W-band may also have some utility. The spectra may be simulated with an anisotropic broadening function. This sort of data analysis may yield information about orientation selectivity and possible alignment of spin-labels[58].

CHAPTER 7

CONCLUSION

Heterogeneous Functionalization of PAMAM Dendrimers

The technique of sequential addition of functional groups to PAMAM dendrimers by reaction of isothiocyanates to form thiourea linkages has proved to have several advantages. Firstly, the reactions go to completion at ambient conditions in DMSO. The ratio of substituents for heterogeneous functionalization can be easily controlled through the use of stoichiometric amounts of reactants. As a result, purification to remove excess reactants is necessary.

For all isothiocyanates and generations of dendrimers tested, the loading distribution that of a non-interacting (non-cooperative) system. The analytical technique developed to abstract the loading distributions from MALDI-TOF MS line-widths will allow the loading distribution to be determined when other (possibly interacting) groups are used to functionalize dendrimers.

The spatial distribution of all functionalities tested has also proved to be random. The spin-label EPR technique is sensitive to spatial distribution, and while not sensitive to distortions in shape (within reason), analysis of the dipolar line-broadening does yield information about other physical attributes. Values for the diameters of the loaded dendrimers as well as information about backfolding can be obtained from simulations. Changes in these characteristics as a function of generation are also made evident.

Virus Capsids

With this framework of spectral analysis and simulation in place, expansion to other systems has already started to take place. Virus capsids are another example of a large macromolecule whose physical characteristics and chemistry could be monitored by SL-EPR. Work has been started in our lab (by Robert Usselman) on the CCMV (chlorotic cowpea mottle virus) virus; its capsid consists of 180 identicle protein subunits arranged in icosahedral symmetry. One property of CCMV that can be probed by SL-EPR is its ability to swell (from 28 nm to 32 nm in diameter) and open pores with changes in Ca^{2+} and pH.

It is easy to conceptualize the experiment, use site-direct spin-labeling to label sites on the protein monomers, then monitor the spatial distribution by analyzing the dipolar interaction. The advantage of our experience extends beyond the analysis of the EPR spectra; we can use our simulation programs to pick which amino acid residues would make good candidates for labeling. The calculations are actually less complicated; unlike with dendrimers where the exact position of the terminal groups is not known and many random choices of site position are needed. Coordinates of all the residues can be abstracted from the pdb file (Protein Data Bank) and can be used to calculate the expected dipolar broadening for each site. This can be done with the swollen form also and the results compared. The difference in dipolar broadening with conformation change can be calculated by merely subtracting the results, Figure 7-1.

It is hoped that the assembly of CCMV from monomers into the complete capsid can be monitored by SL-EPR also. The exact progression of the assembly is not known, but it is suspected to start with dimers then progress to either pentamers or hexamers of

dimers. Again, intelligent experiment design is facilitated by initial calculations for all possible labeling sites. Assuming that the protein-protein interface in the precursors is the same as that in full assembly still leaves 12 possibilities for dimer structures. Labeling sites that distinguish among these possibilities can be chosen by simulation, the coordinates of the different possible pairs of monomers can be abstracted from the coordinates of the assembly.

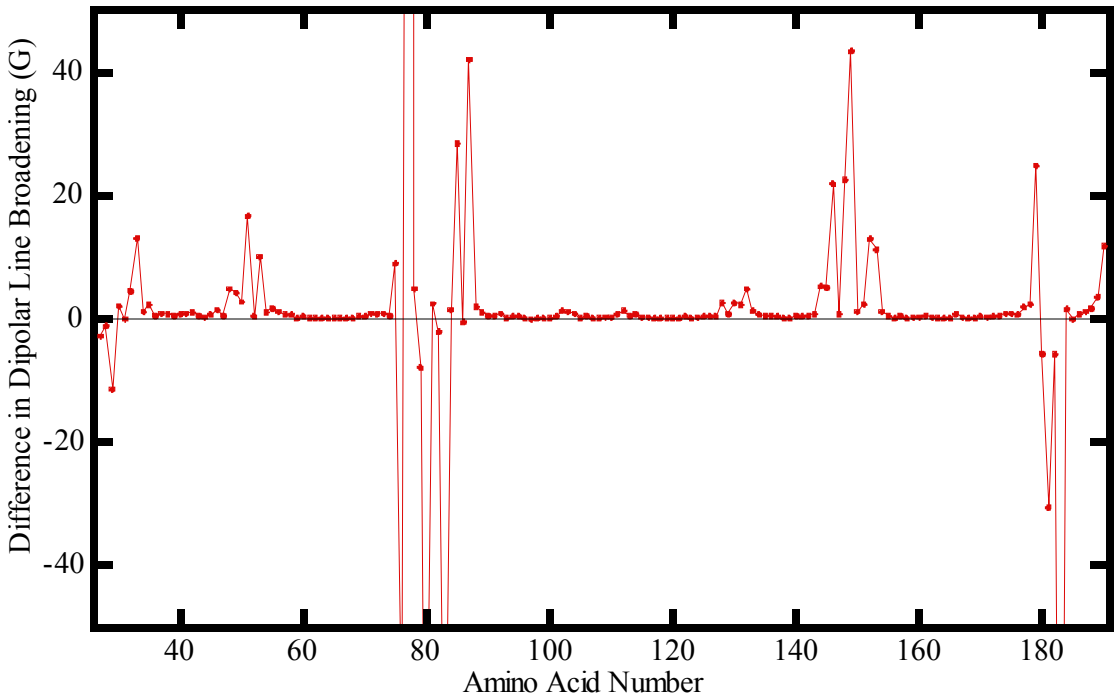


Figure 7-1. Difference in dipolar broadening between open and closed conformation of CCMV predicted for site-directed spin-labeling of different amino acids.

Nitrogenase Fe Protein

Another application for this technique was developed for work in collaboration with the John Peters group here at MSU. As part of the functionality of the nitrogenase Fe protein, a large hinging motion takes place. A homo-dimer, it is a great candidate for

SDSL-EPR to monitor these quaternary structure changes. One mutation will provide two total labels in the dimer needed for a dipolar interaction. While many successful SDSL-EPR experiment have been designed by merely viewing a rendering of the molecule on the computer and picking sites for labeling based on their distance of separation, we chose to look at all possible sites. Again, this was based on coordinates from pdb files. However, sites that were good from a dipolar coupling standpoint, were unacceptable for labeling: they tended to be at the interface between the monomers and therefore mutation and labeling would interfere with protein function.

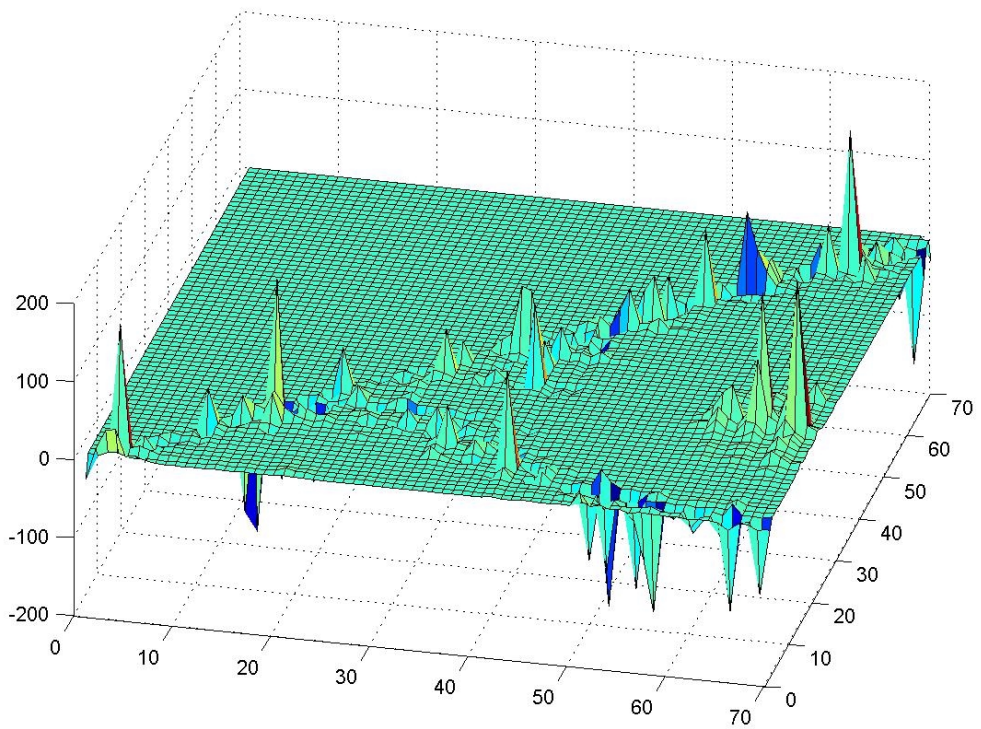


Figure 7-2. 2-D plot of the difference (between endpoints of its hinging motion) in dipolar broadening for 2 different spin-label sites in the nitrogenase Fe protein. z-axis is dipolar difference in gauss, x and y are amino acid number (-200, that is, 0 represents residue number 200).

Sites away from the interface had insufficient dipolar interaction; a site to the side of the interface has its analog not opposite (mirror symmetry) across the plane of the interface but rotated around the interface by 180°. What was needed was a double mutant (the norm for SDSL-EPR of monomeric proteins), so that the nearest neighboring label was opposite across the plane of the interface. With nearly 300 amino acids in the monomer, a total of over 35,000 (unique, $\sim 300^2/2$) double mutants (four total labels per dimer) are possible. This large amount of data can be displayed as a 2-D plot , a section of which is shown in Figure7-2. With a double mutant there are a large number of combinations to choose from and this makes it possible to choose a pair of sites without other compromises.

REFERENCES CITED

REFERENCES CITED

1. Tomalia, D.A. , Naylor, A., Goddard, W.I. *Starburst dendrimers:molecular control of size, shape, surface chemistry, and flexibility from atoms to macroscopic matter*. Angew. Chem. Int. Ed. Engl., 1990. **29** p. 138-175

2. Woller, E.K. and M.J. Cloninger, *The lectin-binding properties of six generations of mannose-functionalized dendrimers.* Org Lett, 2002. **4**(1): p. 7-10.

3. Cloninger, M.J., E. Walter, E.K. Woller, D.J. Singel, and K. Sebby, *Heterogeneous dendrimer surface functionalization.* Abstracts of Papers of the American Chemical Society, 2001. **222**: p. U131-U131.

4. Morgan, J.R. and M.J. Cloninger, *Heterogeneously functionalized dendrimers.* Curr Opin Drug Discov Devel, 2002. **5**(6): p. 966-73.

5. Woller, E.K., E.D. Walter, J.R. Morgan, D.J. Singel, and M.J. Cloninger, *Altering the strength of lectin binding interactions and controlling the amount of lectin clustering using mannose/hydroxyl-functionalized dendrimers.* J Am Chem Soc, 2003. **125**(29): p. 8820-6.

6. Woller, E.K. and M.J. Cloninger, *Mannose functionalization of a sixth generation dendrimer.* Biomacromolecules, 2001. **2**(3): p. 1052-4.

7. Cloninger, M., *Dendrimers and protein cages as nanoparticles in drug delivery.* Drug Discov Today, 2004. **9**(3): p. 111-2.

8. Cloninger, M.J., *Biological applications of dendrimers.* Curr Opin Chem Biol, 2002. **6**(6): p. 742-8.

9. Persson, M., J.R. Harbridge, P. Hammarstrom, R. Mitri, L.G. Martensson, U. Carlsson, G.R. Eaton, and S.S. Eaton, *Comparison of electron paramagnetic resonance methods to determine distances between spin labels on human carbonic anhydrase II.* Biophysical Journal, 2001. **80**(6): p. 2886-2897.

10. Zhou, Y., B.E. Bowler, K. Lynch, S.S. Eaton, and G.R. Eaton, *Interspin distances in spin-labeled metmyoglobin variants determined by saturation recovery EPR.* Biophysical Journal, 2000. **79**(2): p. 1039-1052.

11. Eaton, S.S. and G.R. Eaton, *Electron paramagnetic resonance techniques for measuring distances in proteins.* Structures and Mechanisms: From Ashes to Enzymes, 2002. **827**: p. 321-339.

12. Steinhoff, H.-J., N. Radzwill, W. Thevis, V. Lenz, D. Brandenburg, A. Antson, G. Dodson, and A. Wollmer, *Determination of interspin distances between spin labels attached to insulin: Comparison of electron paramagnetic resonance data with the x-ray structure.* Biophysical Journal, 1997. **73**: p. 3287-3298.

13. Rabenstein, M.D. and Y.-K. Shin, *Determination of the distance between two spin labels attached to a macromolecule.* Proc. Natl. Acad. Aci., 1995. **92**: p. 8239-3243.

14. Voss, J., L. Salwinski, H.R. Kaback, and W.L. Hubbell, *A method for distance determination in proteins using a designed metal ion binding site and site-directed spin labeling: evaluation with T4 lysozyme.* Proc Natl Acad Sci U S A, 1995. **92**(26): p. 12295-9.

15. Cornish, V.W., D.R. Benson, C.A. Altenbach, K. Hideg, W.L. Hubbell, and P.G. Schultz, *Site-specific incorporation of biophysical probes into proteins.* Proc Natl Acad Sci U S A, 1994. **91**(8): p. 2910.

16. Calciano, L.J., W.A. Escobar, G.L. Millhauser, S.M. Miick, J. Rubaloff, A.P. Todd, and A.L. Fink, *Side-chain mobility of the beta-lactamase A state probed by electron spin resonance spectroscopy.* Biochemistry, 1993. **32**(21): p. 5644-9.

17. Saxena, S. and J.H. Freed, *Theory of double quantum two-dimensional electron spin resonance with application to distance measurements.* J. Chem. Phys., 1997. **107**(5): p. 1317-1340.

18. Raitsimring, A., R.H. Crepeau, and J.H. Freed, *Nuclear modulation effects in "2 + 1" electron spin-echo correlation spectroscopy.* J. Chem. Phys., 1995. **102**(22): p. 8746-62.

19. Miick, S.M. and G.L. Millhauser, *Measuring Heisenberg spin exchange between spin-labeled peptides with 2D ELDOR.* J. Magn. Reson., Ser. B, 1994. **104**(1): p. 81-4.

20. Singel, R.G.L.a.D.J., *DEER Spin Echo Modulation: Spectroscopic Measurement of Electron Spin Pair Separations in Orientationally Disordered Solids.* Journ. Chem. Phys., 1993. **98**: p. 5134-5146.

21. Larsen, R.G. and D.J. Singel, *DEER Spin Echo Modulation: Spectroscopic Measurement of Electron Spin Pair Separations in Orientationally Disordered Solids.* Journal of Chemical Physics, 1993. **98**: p. **51**34-5146.

22. Zhao, M., K.C. Zen, J. Hernandez-Borrell, C. Altenbach, W.L. Hubbell, and H.R. Kaback, *Nitroxide scanning electron paramagnetic resonance of helices IV and V and the intervening loop in the lactose permease of Escherichia coli.* Biochemistry, 1999. **38**(48): p. 15970-7.

23. Dalton, L.R., *EPR and advanced EPR studies of biological systems.* 1985, Boca Raton: CRC Press.

24. Altenbach, C., T. Marti, H.G. Khorana, and W.L. Hubbell, *Transmembrane protein structure: spin labeling of bacteriorhodopsin mutants.* Science, 1990. **248**(4959): p. 1088-92.

25. Likhtenshtein, G.I., *Spin Labeling Methods in Molecular Biology*. 1974, Moscow: John Wiley & Sons. 258.

26. Poole, C.P.J., *Electron Spin Resonance*. 2nd ed. 1983, Mineola: Dover Publications.

27. Bielinska, A., J. Kukowskalatallo, L.T. Piehler, R. Yin, R. Spindler, D.A. Tomalia, and J.R. Baker, *Starburst(R) Pamam Dendrimers - a Novel Synthetic Vector for the Transfection of DNA into Mammalian-Cells.* Abstracts of Papers of the American Chemical Society, 1995. **210**: p. 145-Pmse.

28. Ottaviani, M.F., F. Furini, A. Casini, N.J. Turro, S. Jockusch, D.A. Tomalia, and L. Messori, *Formation of supramolecular structures between DNA and starburst*

dendrimers studied by EPR, CD, UV, and melting profiles. Macromolecules, 2000. **33**(21): p. 7842-7851.

29. Ottaviani, M.F., P. Matteini, M. Brustolon, N.J. Turro, S. Jockusch, and D.A. Tomalia, *Characterization of starburst dendrimers and vesicle solutions and their interactions by CW- and pulsed-EPR, TEM, and dynamic light scattering.* Journal of Physical Chemistry B, 1998. **102**(31): p. 6029-6039.

30. Ottaviani, M.F., R. Daddi, M. Brustolon, N.J. Turro, and D.A. Tomalia, *Structural modifications of DMPC vesicles upon interaction with poly(amidoamine) dendrimers studied by CW-electron paramagnetic resonance and electron spin-echo techniques.* Langmuir, 1999. **15**(6): p. 1973-1980.

31. Ottaviani, M.F., P. Favuzza, B. Sacchi, N.J. Turro, S. Jockusch, and D.A. Tomalia, *Interactions between starburst dendrimers and mixed DMPC/DMPA-Na vesicles studied by the spin label and the spin probe techniques, supported by transmission electron microscopy.* Langmuir, 2002. **18**(6): p. 2347-2357.

32. Ottaviani, M.F., N.J. Turro, S. Jockusch, and D.A. Tomalia, *Characterization of starburst dendrimers by EPR. Aggregational processes of a positively charged nitroxide surfactant.* Journal of Physical Chemistry, 1996. **100**(32): p. 13675-13686.

33. Ottaviani, M.F., P. Andechaga, N.J. Turro, and D.A. Tomalia, *Model for the interactions between anionic dendrimers and cationic surfactants by means of the spin probe method.* Journal of Physical Chemistry B, 1997. **101**(31): p. 6057-6065.

34. Ottaviani, M.F., E. Cossu, N.J. Turro, and D.A. Tomalia, *Characterization of Starburst Dendrimers by Electron-Paramagnetic-Resonance .2. Positively Charged Nitroxide Radicals of Variable Chain-Length Used as Spin Probes.* Journal of the American Chemical Society, 1995. **117**(15): p. 4387-4398.

35. Ottaviani, M.F., N.J. Turro, S. Jockusch, and D.A. Tomalia, *Aggregational process of the positively charged surfactants CTAC and CAT16 in the presence of starburst dendrimers: An electron paramagnetic resonance spectroscopic study.* Colloids and Surfaces a-Physicochemical and Engineering Aspects, 1996. **115**: p. 9-21.

36. Ottaviani, M.F., B. Sacchi, N.J. Turro, W. Chen, S. Jockusch, and D.A. Tomalia, *An EPR study of the interactions between starburst dendrimers and polynucleotides.* Macromolecules, 1999. **32**(7): p. 2275-2282.

37. Bielinska, A., J.F. KukowskaLatallo, J. Johnson, D.A. Tomalia, and J.R. Baker, *Regulation of in vitro gene expression using antisense oligonucleotides or antisense expression plasmids transfected using starburst PAMAM dendrimers.* Nucleic Acids Research, 1996. **24**(11): p. 2176-2182.

38. Bosman, A.W., R.A.J. Janssen, and E.W. Miejer, *Five generations of nitroxyl-functionalized dendrimers.* Macromolecules, 1997. **30**(12): p. 3606-3611.

39. Wiener, E.C., F.P. Auteri, J.W. Chen, M.W. Brechbiel, O.A. Gansow, D.S. Schneider, R.L. Belford, R.B. Clarkson, and P.C. Lauterbur, *Molecular Dynamics of Ion-Chelate Complexes Attached to Starburst Dendrimers.* J. Am. Chem. Soc., 1996. **118**(33): p. 7774-7782.

40. Yordanov, A.T., K. Yamada Ki, M.C. Krishna, J.B. Mitchell, E. Woller, M. Cloninger, and M.W. Brechbiel, *Spin-Labeled Dendrimers in EPR Imaging with Low Molecular Weight Nitroxides.* Angew Chem Int Ed Engl, 2001. **40**(14): p. 2690-2692.

41. Yordanov, A.T., A.L. Lodder, E.K. Woller, M.J. Cloninger, N. Patronas, D. Milenic, and M.W. Brechbiel, *Novel iodinated dendritic nanoparticles for computed tomography (CT) imaging.* Nano Letters, 2002. **2**(6): p. 595-599.

42. Krinichnyi, V.I., S.D. Chemerisov, and Y.S. Lebedev, *EPR and charge-transport studies of polyaniline.* Physical Review B, 1997. **55**(24): p. 16233-16244.

43. Magon, C.J., R.R. de Souza, A.J. Costa, E.A. Vidoto, R.M. Faria, and O.R. Nascimento, *Spin dynamics study in doped polyaniline by continuous wave and pulsed electron paramagnetic resonance.* Journal of Chemical Physics, 2000. **112**(6): p. 2958-2966.

44. Salikhov, K.M.a.T., Y. D., *Electron Spin-Echo Studies of Spin-Spin Interactions in Solids*, in *Ch. 7 in Time Domain Electron Spin Resonance*, L.K.a.R.N. Schwartz, Editor. 1979, John Wiley and Sons.

45. Peterson, J., V. Allikmaa, J. Subbi, T. Pehk, and M. Lopp, *Structural deviations in poly(amidoamine) dendrimers: a MALDI-TOF MS analysis.* European Polymer Journal, 2003. **39**(1): p. 33-42.

46. Zhou, L., D.H. Russell, M.Q. Zhao, and R.M. Crooks, *Characterization of poly(amidoamine) dendrimers and their complexes with Cu2+ by matrix-assisted laser desorption ionization mass spectrometry.* Macromolecules, 2001. **34**(11): p. 3567-3573.

47. Van Vleck, J.H., *The Dipolar Broadening of Magnetic Resonance Lines in Crystals.* Physical Review, 1948. **74**(9): p. 1168-83.

48. Li, J., L.T. Piehler, D. Qin, J.R. Baker, D.A. Tomalia, and D.J. Meier, *Visualization and characterization of poly(amidoamine) dendrimers by atomic force microscopy.* Langmuir, 2000. **16**(13): p. 5613-5616.

49. Slichter, C.P., *Principles of Magnetic Resonance*. Third ed. 1996: Springer Verlag. 669.

50. Hustedt, E.J., A.I. Smirnov, C.F. Laub, C.E. Cobb, and A.H. Beth, *Molecular distances from dipolar coupled spin-labels: the global analysis of multifrequency continuous wave electron paramagnetic resonance data.* Biophys. J., 1997. **72**(4): p. 1861-1877.

51. Uppuluri, S., S.E. Keinath, D.A. Tomalia, and P.R. Dvornic, *Rheology of dendrimers. I. Newtonian flow behavior of medium and highly concentrated solutions of polyamidoamine (PAMAM) dendrimers in ethylenediamine (EDA) solvent.* Macromolecules, 1998. **31**(14): p. 4498-4510.

52. Pake, G.E., *Nuclear Resonance Absorption in Hydrated Crystals: Fine Structure of the Proton Line.* The Journal of Chemical Physics, 1948. **16**(4): p. 327-36.

53. Pake, G.E. and T.L. Estle, *Physical Principles of Electron Paramagnetic Resonance*. Second ed. 1973, Reading, MA: W. A. Benjamin, Inc.

54. Lindgren, M., G.R. Eaton, S.S. Eaton, B.H. Jonsson, P. Hammarstrom, M. Svensson, and U. Carlsson, *Electron spin echo decay as a probe of aminoxyl*

environment in spin-labeled mutants of human carbonic anhydrase II. Journal of the Chemical Society-Perkin Transactions 2, 1997(12): p. 2549-2554.

55. Zhou, Y., B.E. Bowler, G.R. Eaton, and S.S. Eaton, *Electron spin lattice relaxation rates for S=1/2 molecular species in glassy matrices or magnetically dilute solids at temperatures between 10 and 300 K.* Journal of Magnetic Resonance, 1999. **139**(1): p. 165-174.

56. Schwartz, D.A., E.D. Walter, S.J. McIlwain, V.N. Krymov, and D.J. Singel, *High Frequency (94.9 GHz) EPR Spectroscopy of Paramagnetic Centers in a Neutron-Irradiated Sapphire Single Crystal Fiber.* Appl. Magn. Reson., 1999. **16**(2): p. 223-236.

57. Poluektov, O.G. and A.A. Doubinski, *Two-dimensional high-field electron spin echo spectroscopy of nitroxide spin probes.* Chemical Physics Letters, 1998. **288**(5-6): p. 841-846.

58. Smirnov, Belford, and Clarkson, *Comparative spin label spectra at X-band and W-band.* Bilogical magnetic resonance, 1998. **14**: p. 83-107.

APPENDICES

APPENDIX A: EPR SPECTRA

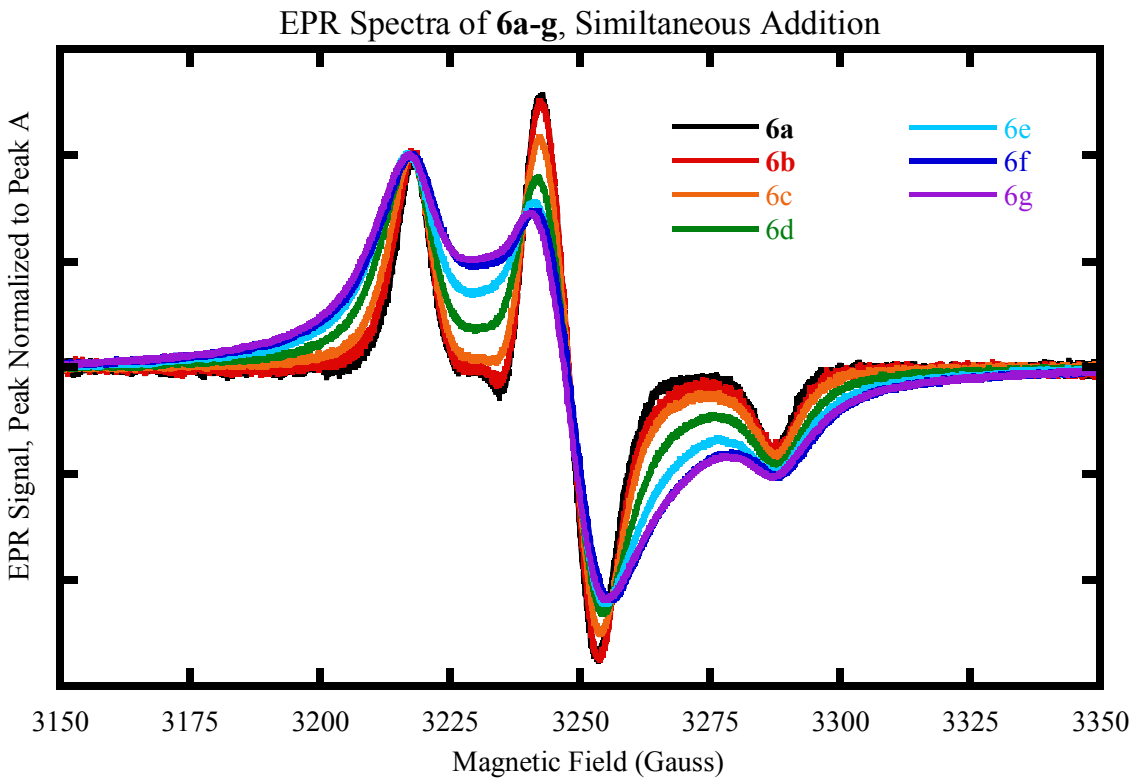
EPR spectra of heterogeneously loaded G(4) PAMAM dendrimers. Labeled according to the following legend:

Second compound

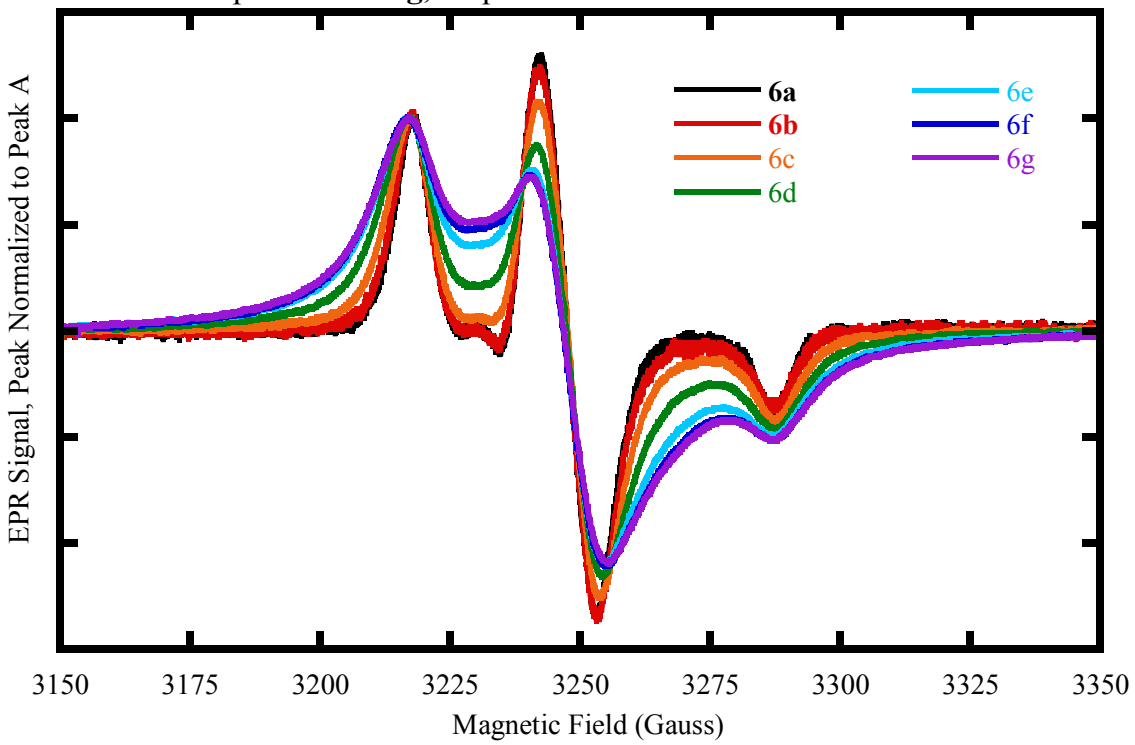
2,2,6,6 tetramethyl piperidine	**6a-g**
Propanol	**7a-g**
Phenol	**8a-g**
t-butyl	**9a-g**
TEMPO	**1**

% TEMPO Loadin

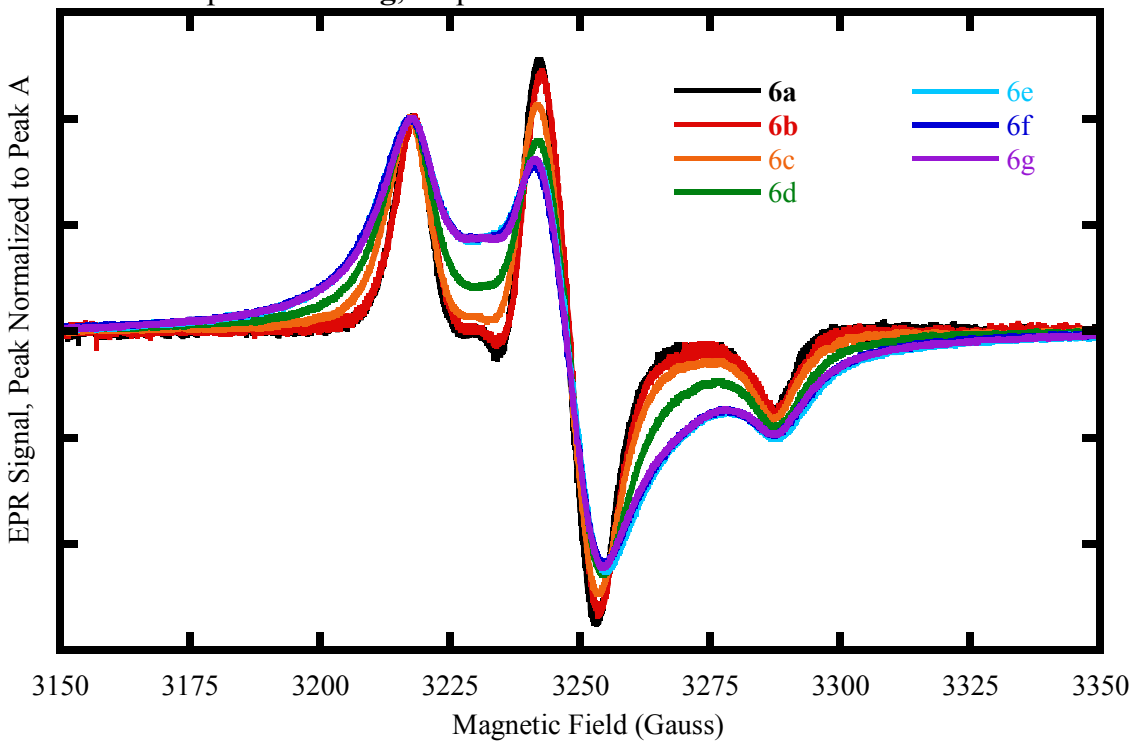
a	5%
b	10%
c	25%
d	50%
e	75%
f	90%
g	95%

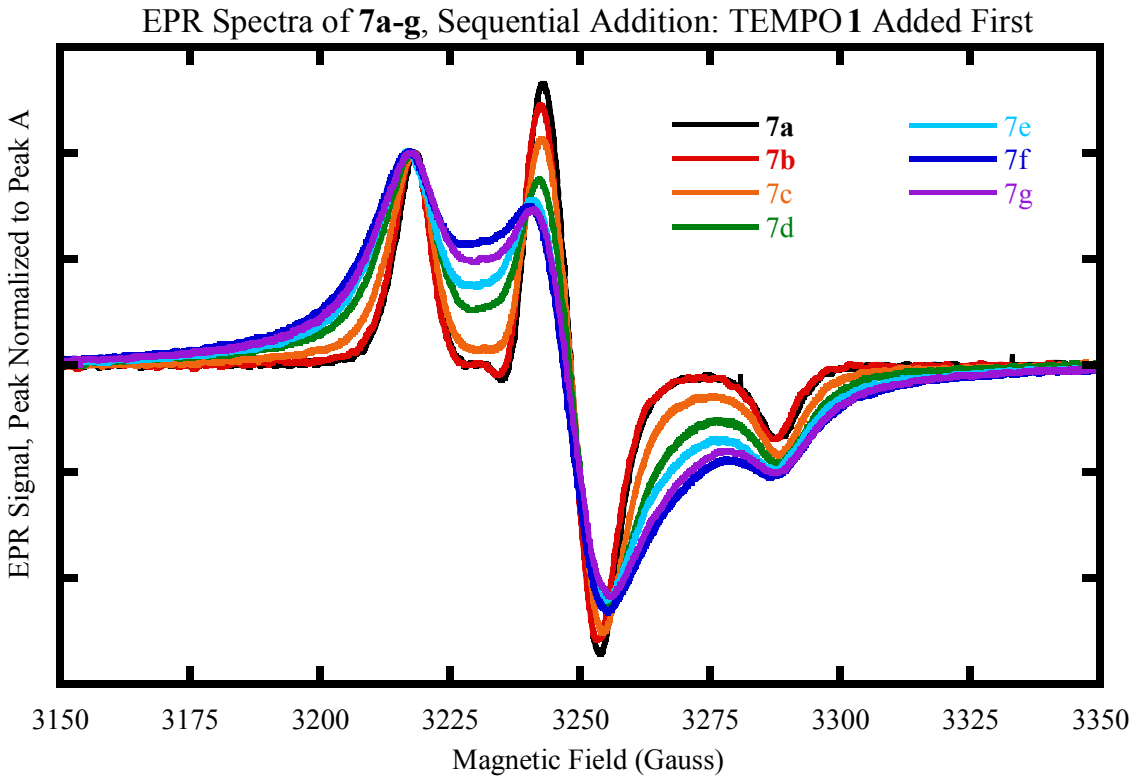


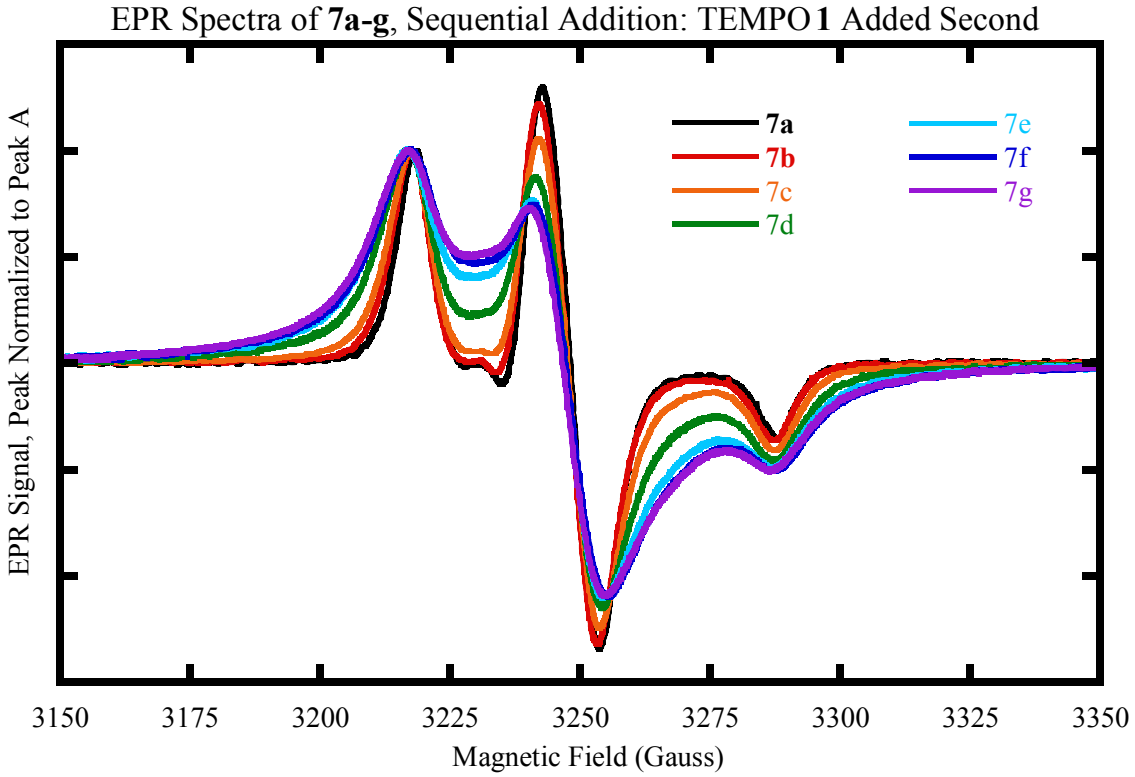
EPR Spectra of **6a-g**, Sequential Addition: TEMPO **1** Added First



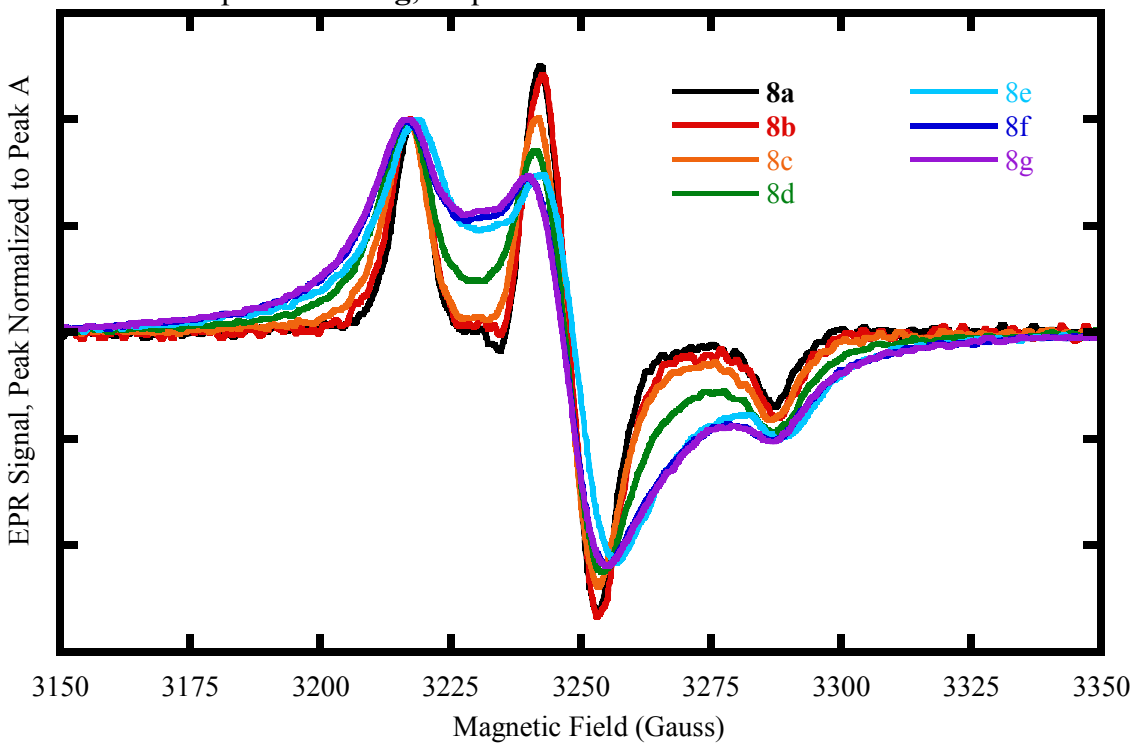
EPR Spectra of **6a-g**, Sequential Addition: TEMPO **1** Added Second



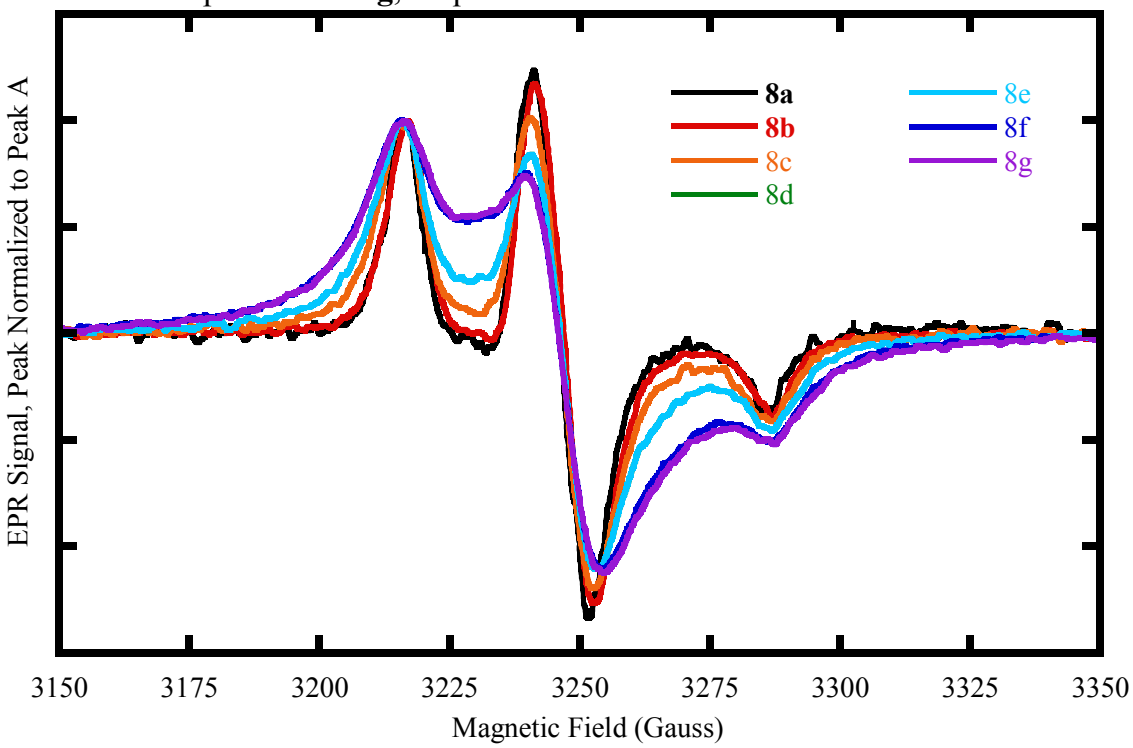

EPR Spectra of 7a-g, Sequential Addition: TEMPO 1 Added First
EPR Signal, Peak Normalized to Peak A
7a
7b
7c
7d
7e
7f
7g
3150
3175
3200
3225
3250
3275
3300
3325
3350
Magnetic Field (Gauss)


EPR Spectra of 7a-g, Sequential Addition: TEMPO 1 Added Second
EPR Signal, Peak Normalized to Peak A
7a
7b
7c
7d
7e
7f
7g
3150
3175
3200
3225
3250
3275
3300
3325
3350
Magnetic Field (Gauss)

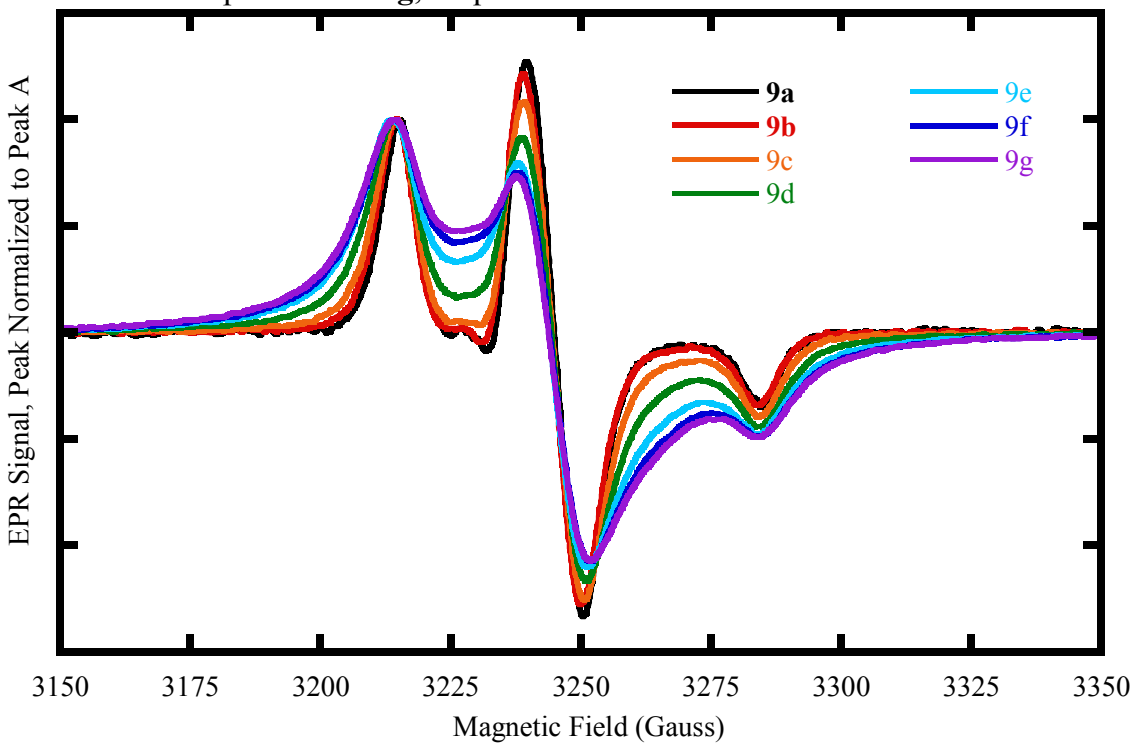
EPR Spectra of **8a-g**, Sequential Addition: TEMPO **1** Added First



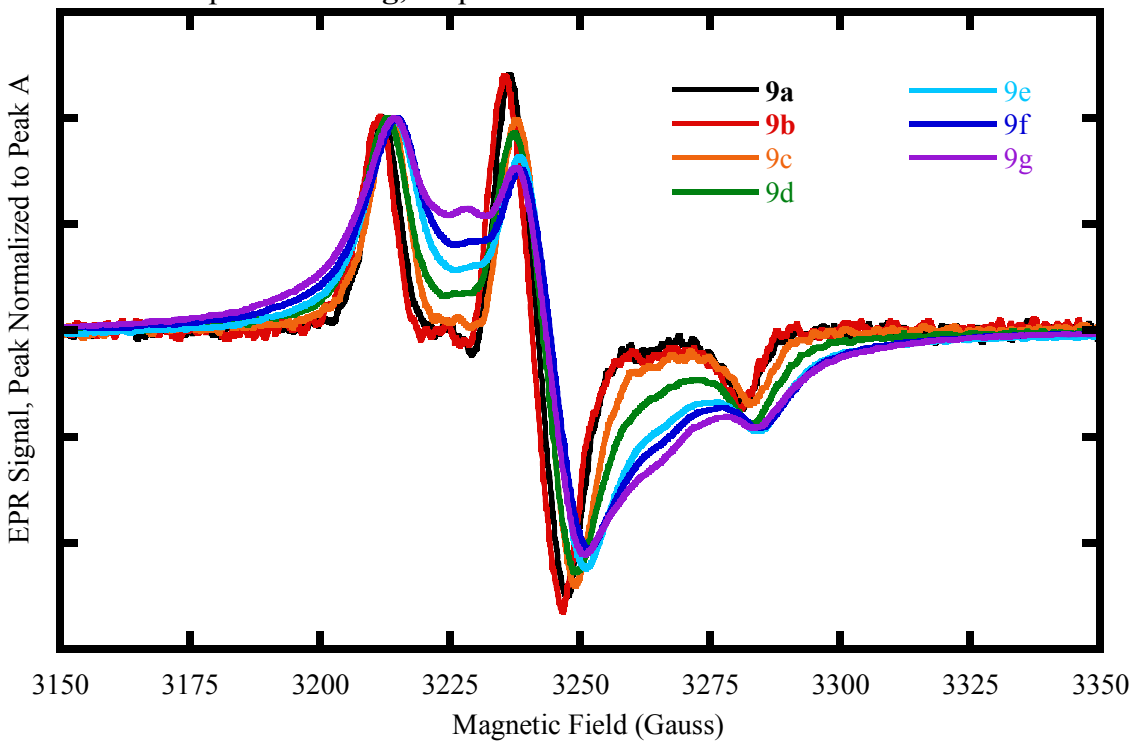
EPR Spectra of **8a-g**, Sequential Addition: TEMPO **1** Added Second



EPR Spectra of **9a-g**, Sequential Addition: TEMPO **1** Added First



EPR Spectra of **9a-g**, Sequential Addition: TEMPO **1** Added Second



Appendix B: RMS.C

A C program for choosing sites on a sphere (rms.c):

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

/*
double erand48 (unsigned short xsubi[3]);
*/


/* using malloc(3x) */
#include <sys/types.h>
#include <malloc.h>

#include <time.h>
 /*time usage:
 time_t time(time_t *tloc); */



int   debug = 0;

extern  double    str2num(char *temp);

#define maxlinelength 256
#define maxlinesinfile 1024

#define MAX 1000
#define pi 3.1415927


/* actual beginning of main program */
void main (args,argstr)

int args;
char *argstr[];

{ /* beginning of main program */

int   p,i,j,k,n,m,nlimit;
double      a,b,c;
double      x1, x2, y1, y2, z1, z2;
FILE  *infile, *outfile1;
char  filename[20];
int   nsamples;   /* number of samples user desires */
int   linesfound; /* number of lines found in input text stream */
char  textbuffer[maxlinesinfile][maxlinelength]; /* buffer of lines in
text */
```

```
char  **endptr;
unsigned short xsubi[4];
char  instring[10];
char  of1[256], of2[256];
double      r; /* radius of sphere */
double      min;
int         dia,rmin;
double      x[MAX];
double      y[MAX];
double      z[MAX];
double      d[56];
double      theta[56];
double      phi[56];
double      dist;
double      dx,dy;

/* trying a new type  */
time_t      presenttime;
time_t      tloc;

int   zero, one, two;



/* initializing random number generator */
      /* dumb initialization for the moment */
xsubi[0] = (unsigned short) 2;
xsubi[1] = (unsigned short) 1;
xsubi[2] = (unsigned short) 0;
      /* better initialization */
/* presenttime is of type time_t, which is equivalent to "long" */
presenttime = time(&tloc);

/* pointers to xsubi -- we can change these if we don't like something
about
                  the way the random numbers are coming out
                  setting all these to the same integer value will
                  prevent the routine from generating a different
                  series of numbers each time it is called numbers */
zero = 0;
one = 1;
two = 2;


/* randomly picking nsamples input lines from the textbuffer */

for (r = 31; r < 35; r = r + .5)
      {
      dia = r * 2;
      for (min = 4; min < 10; min = min + 1)
            {
            rmin = min;
            for (n = 0; n < 10; n = n + 1)
                  { /* beginning of n loop */
```

```
                    presenttime = time(&tloc);
                    j = 0xffff &  presenttime;
                    xsubi[zero] = (unsigned short ) j;

                    j = 0xffff0000 &  presenttime;
                    j = j / 0xffff;
                    xsubi[one] = (unsigned short ) j;

                    j = (n + (int) xsubi[zero]) * ((int) xsubi[one]);
                    xsubi[two] = (unsigned short ) j;

                    for (k = 0; k < 56; k = k + 1)
                          {
                          x[k]=0.0;
                          y[k]=0.0;
                          z[k]=0.0;
                          }
                    c = 0;
                    i = 0;
                    while ((i < 56))
                          { /* beginning of first i loop */
                          /* erand48 returns a number in the interval
[0.0,1.0] */
                          c=c+1;
                          i=i+1;
                          x[i] = (erand48(xsubi)-.5)*20;
                          y[i] = (erand48(xsubi) -.5)*20;
                          z[i] = (erand48(xsubi) -.5)*20;
                          a = r/(sqrt(x[i]*x[i] + y[i]*y[i] +
z[i]*z[i]));
                          x[i] = a * x[i];
                          y[i] = a * y[i];
                          z[i] = a * z[i];
                          k = i - 1;
                                for (p = 0; p < i; p=p+1)
                                      {
                                      dist =        (x[i]-x[1])*(x[i]-
x[1]);
                                      dist = dist + (y[i]-y[1])*(y[i]-
y[1]);
                                      dist = dist + (z[i]-z[1])*(z[i]-
z[1]);
                                      dist = sqrt(dist);
                                      if ( dist < min )
                                            {
                                            i = k;
                                            }
                                      if ( c > 1000000)
                                            {
                                            i = 56;
                                            }

                                      }
                          } /* end of first i loop */
```

```
                                sprintf(filename,"%d-55sphere%d-%d-
%d",k,dia,rmin,n);
                                outfile1 = fopen(filename,"w");

                          for (i = 1; i < k+1; i = i + 1)
                                {

      fprintf(outfile1,"%d\t%f\t%f\t%f\n",i,x[i],y[i],z[i]);

                                } /* end of second i loop */

                                c = 0;
                                fclose(outfile1);

                     } /* end of n loop */

            }/* end of min loop */

      }/* end of r loop */


} /* end of main program */
```

Appendix C: RANBROAD.C

A C program to calculate an lbf using the random model, Ranbroad.c:

```
extern int  debug = 0;
extern double str2num(char[99]);

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <time.h>

#define MAX 257
#define LINELENGTH 1024
#define     LIMIT 1000

main (args,argstr)
int args;
char *argstr[];

{ /* beginning of main program */

FILE *infile;
FILE *outfile1, *outfile2;
double      x[MAX],y[MAX],z[MAX];
char  lines[MAX][LINELENGTH];
int   nn[MAX];
int   i,j,k,l, m,n,p, line, max;
int   jj, ni, nj;
int   a,b,v,w;
int   nlimit;
char  *inpline;
char  inf[LINELENGTH];
char  of1[LINELENGTH];
char  of2[LINELENGTH];
double      broad[4001][MAX],field,btemp,d[MAX], dist;
double      sum[3100][MAX];
int   row,col;

/* for random number generation */
unsigned short xsubi[3];
int     zero = 0;
int   one = 1;
int   two = 2;

/* trying a new type  */
time_t  presenttime;
time_t  tloc;


/* initializing random number generator */
```

```
presenttime = time(&tloc);
jj = 0xffff &  presenttime;
xsubi[zero] = (unsigned short ) jj;

jj = 0xffff0000 &  presenttime;
jj = jj / 0xffff;
xsubi[one] = (unsigned short ) jj;

jj = ((int) xsubi[zero]) * ((int) xsubi[one]);
xsubi[two] = (unsigned short ) jj;




/* printf("arg 1 = %s\n",argstr[1]); */



if ( ( infile = fopen(argstr[1],"r") ) != (FILE *) NULL )
      {/* success */}
else
      {
      fprintf(stderr,"load: file '%s' not found.\n",argstr[1]);
      return(1);
      }

i = 0;
while ( (argstr[1][i] != '.' ) && ( argstr[1][i] != '\0') )
      {
      inf[i] = argstr[1][i];
      i = i + 1;
      }
inf[i] = '\0';
/* printf("output filename root = %s\n",inf); */



line = 1;
while ( !(feof(infile)) )
      { /*beginning of while loop to read file */

      /* reading line of input file  */
      fscanf(infile,"%[^\n]\n",&lines[line][0]);
      lines[line][-1] = '0';
/*    printf("'%s'\n",inpline);  */

      /* i = fscanf(infile,"%d %f %f
%f\n",&nn[line],&x[line],&y[line],&z[line]);
      printf("%d\t%f\t%f\t%f\n",nn[line],x[line],y[line],z[line]);
      */

      inpline = &lines[line][0];

      i = findtextfield(inpline,1);
      nn[line] = (int) str2num(&inpline[i]) ;
```

```
      j = findtextfield(inpline,2);
      x[line] = str2num(&inpline[j]) ;

      k = findtextfield(inpline,3);
      y[line] = str2num(&inpline[k]) ;

      l = findtextfield(inpline,4);
      z[line] = str2num(&inpline[l]) ;
      /* printf("number: '%s'\n",&inpline[l]); */

/*
      printf("%d\t%d\t%f\t%f\t%f\n",line,nn[line],x[line],y[line],z[lin
e]);
*/

      max = line;
      line = line + 1;

      } /*end of while loop to read file */

fclose(infile);


/* ================================ adapted  from random2.c */

/* randomly picking nsamples input lines from the textbuffer */

/*
printf("p loop \n");
*/

/* return(1); */

for (i = 0; i < 550 ; i = i + 1)
      {
      for (j = 0; j < max ; j = j + 1)
            {
            sum[i][j] = 0.0;
            }
      }

for (p = 2; p <= max; p = p + 1)
      { /* beginning of p loop */


      nlimit = LIMIT/p;
      for (n = 0; n < nlimit; n = n + 1)
            { /* beginning of n loop */

            for (a = 0; a < 600; a = a + 1) /* randomizing the list */
                  { /* beginning of "a" loop */
                  v = (int) ( erand48(xsubi) * (double) max + 1.0 ) ;
                        w = (int) ( erand48(xsubi) * (double) max + 1.0
) ;
```

```
                b = nn[v];
                nn[v] = nn[w];
                nn[w] = b;
                } /* end of "a" loop */

          for (i = 1; i <= p; i = i + 1)
                { /* beginning of second i loop */
                d[i] = 71.0; /* maximum difference */

                for (j = 1; j <= p; j = j + 1)
                      { /* beginning of j loop */
                      ni = nn[i];
                      nj = nn[j];
                      dist =        (x[ni]-x[nj])*(x[ni]-x[nj]);
                      dist = dist + (y[ni]-y[nj])*(y[ni]-y[nj]);
                      dist = dist + (z[ni]-z[nj])*(z[ni]-z[nj]);
                      dist = sqrt(dist);

                      if (i != j)
                            {
                                  sum[i+(n * p)][1] = sum[i+(n *
p)][1] + pow(dist,-6.);
                            }

                      } /* end of j loop */

                } /* end of second i loop */

          } /* end of n loop */

      } /* end of p loop */
/* ============================================ end of stuff from
random2.c */
for (i = 1; i <= (3025) ; i = i + 1)
      {
      for (j = 2; j <= max ; j = j + 1)
            {
            if (sum[i][j]> 1e-15)
                  {
                  sum[i][j] = (sqrt(sum[i][j])*19525.97);
                  }

            }
      }

/*calculates sum of gaussian broadening functions*/

for (k = 2; k <= max; k = k + 1)
      {
      for (j = 0; j <= 3999; j = j + 1)
            {
            field = j/20.0 - 99.975;

            for (i = 0; i <= 3025; i = i + 1)
                  {
```

```
                  if (sum[i][k] > 1e-15)
                        {
                        btemp = ( exp ( -0.693 * (field/(.5 *
sum[i][k]))* (field/(.5 * sum[i][k]))))/sum[i][k];
                        broad[j][k] = broad[j][k] + btemp;
                        }
                  }
            }
      }


/* this prints column titles */
printf("sum of r-6\t");
for (j = 2; j <= max ; j = j + 1)
      {
      printf("%d\t",j);
      }
printf("\n");

/* this prints rows of broadening amplitudes */
for (i = 0; i <= (3999) ; i = i + 1)
      {
      field = i/20.0 - 99.975;
      printf("%e\t",field);
      for (j = 2; j <= max ; j = j + 1)
            {
            printf("%e\t",broad[i][j]);
            }
      printf("\n");
      }


} /* end of main program */
```

Appendix D: CLUSTBROAD.C

A C program that calculates an lbf using the cluster model, clustbroad.c:

```
extern int  debug = 0;
extern double str2num(char[99]);

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <time.h>

#define MAX 257
#define LINELENGTH 1024
#define     LIMIT 1000

main (args,argstr)
int args;
char *argstr[];

{ /* beginning of main program */

FILE *infile;
FILE *outfile1, *outfile2;
double      x[MAX],y[MAX],z[MAX];
char  lines[MAX][LINELENGTH];
int   nn[MAX],order[MAX][MAX];
int   i,j,k,l, m,n,p, line, max,swap;
int   jj, ni, nj;
int   a,b,d,v,w;
int   nlimit;
char  *inpline;
char  inf[LINELENGTH];
char  of1[LINELENGTH];
char  of2[LINELENGTH];
double      btemp,field,broad[3999][MAX],distance[MAX][MAX], dist,h;
double      sum[3100][MAX];
int   row,col;

/* for random number generation */
unsigned short xsubi[3];
int     zero = 0;
int   one = 1;
int   two = 2;

/* trying a new type  */
time_t  presenttime;
time_t  tloc;
```

```
/* initializing random number generator */
presenttime = time(&tloc);
jj = 0xffff &  presenttime;
xsubi[zero] = (unsigned short ) jj;

jj = 0xffff0000 &  presenttime;
jj = jj / 0xffff;
xsubi[one] = (unsigned short ) jj;

jj = ((int) xsubi[zero]) * ((int) xsubi[one]);
xsubi[two] = (unsigned short ) jj;









if ( ( infile = fopen(argstr[1],"r") ) != (FILE *) NULL )
      {/* success */}
else
      {
      fprintf(stderr,"load: file '%s' not found.\n",argstr[1]);
      return(1);
      }

i = 0;
while ( (argstr[1][i] != '.' ) && ( argstr[1][i] != '\0') )
      {
      inf[i] = argstr[1][i];
      i = i + 1;
      }
inf[i] = '\0';
/* printf("output filename root = %s\n",inf); */



line = 1;
while ( !(feof(infile)) )
      { /*beginning of while loop to read file */

      /* reading line of input file  */
      fscanf(infile,"%[^\n]\n",&lines[line][0]);
      lines[line][-1] = '0';
/*    printf("'%s'\n",inpline);  */

      /* i = fscanf(infile,"%d %f %f
%f\n",&nn[line],&x[line],&y[line],&z[line]);
            */

      inpline = &lines[line][0];

      i = findtextfield(inpline,1);
```

```
      nn[line] = (int) str2num(&inpline[i]) ;

      j = findtextfield(inpline,2);
      x[line] = str2num(&inpline[j]) ;

      k = findtextfield(inpline,3);
      y[line] = str2num(&inpline[k]) ;

      l = findtextfield(inpline,4);
      z[line] = str2num(&inpline[l]) ;
      /* printf("number: '%s'\n",&inpline[l]); */

/*
      printf("%d\t%d\t%f\t%f\t%f\n",line,nn[line],x[line],y[line],z[lin
e]);
*/

      max = line;
      line = line + 1;

      } /*end of while loop to read file */

fclose(infile);


/* ================================ adapted  from random2.c */

/* randomly picking nsamples input lines from the textbuffer */

/*
printf("p loop \n");
*/

/* return(1); */
/* ================================ setting sum matrix to 0*/
for (i = 0; i < 3100 ; i = i + 1)
      {
      for (j = 0; j <= max ; j = j + 1)
            {
            sum[i][j] = 0.0;
            }
      }

/* ================================ setting distance matrix to 0*/
for (i = 0; i <= max ; i = i + 1)
      {
      for (j = 0; j <= max ; j = j + 1)
            {
            distance[i][j] = 0.0;
            }
      }
/* ================================ setting order matrix i,j=j*/
for (i = 0; i <= max ; i = i + 1)
      {
      for (j = 0; j <= max ; j = j + 1)
```

```
            {
            order[i][j] = j;
            }
      }

/* ================================ calculating distance matrix */
for (i = 1; i <= max ; i = i + 1)
      {
      for (j = 1; j <= max ; j = j + 1)
            {
            distance[i][j] =          pow(x[i]-x[j],2);
            distance[i][j] = distance[i][j] + pow(y[i]-y[j],2);
            distance[i][j] = distance[i][j] + pow(z[i]-z[j],2);
            distance[i][j] = sqrt(distance[i][j]);

            }

      }

/* ================================ calculating order matrix */
for (i = 1; i <= (max) ; i = i + 1)
      {
      swap = 1;
      while (swap == 1)
            {
            swap = 0;
            for (j = 2; j <= max; j = j + 1)
                  {
                  if (distance[i][order[i][j]] <
distance[i][order[i][j-1]])
                        {
                        d = order[i][j];
                        order[i][j] = order[i][j-1];
                        order[i][j-1] = d;
                        swap = 1;
                        }
                  }
            }
      }

/* ================================ calculating r6 matrix */
for (p = 2; p <= (max) ; p = p + 1)
      {
         for (i = 1; i <= max ; i = i + 1)
            {
                  for (k = 1; k <= p; k = k + 1)
                  {
                  for (j = 1; j <= p; j = j + 1)
                        {
                        if (k!=j)
                              {
                              sum[k+p*(i-1)][1] = sum[k+p*(i-1)][1] +
pow(distance[order[i][k]][order[i][j]],-6);
                              }
                        }
```

```
                }

            }
        }
for (i = 1; i <= (3025) ; i = i + 1)
        {
        for (j = 2; j <= max ; j = j + 1)
            {
            if (sum[i][j]> 1e-15)
                {
                sum[i][j] = (sqrt(sum[i][j])*19525.97);
                }

            }
        }


/*calculates sum of gaussian broadening functions*/

for (k = 2; k <= max; k = k + 1)
        {
        for (j = 0; j <= 3999; j = j + 1)
            {
            field = j/20.0 - 99.975;

            for (i = 0; i <= 3025; i = i + 1)
                {
                if (sum[i][k] > 1e-15)
                    {
                    btemp = ( exp ( -0.693 * (field/(.5 *
sum[i][k]))* (field/(.5 * sum[i][k]))))/sum[i][k];
                                broad[j][k] = broad[j][k] + btemp;
                    }
                }
            }
        }


/* this prints column titles */
printf("sum of r-6\t");
for (j = 2; j <= max ; j = j + 1)
        {
        printf("%d\t",j);
        }
printf("\n");

/* this prints rows of broadening amplitudes */
for (i = 0; i <= (3999) ; i = i + 1)
        {
        field = i/20.0 - 99.975;
        printf("%e\t",field);
        for (j = 2; j <= max ; j = j + 1)
            {
            printf("%e\t",broad[i][j]);
            }
```

```
        printf("\n");
        }

} /* end of main program */
```

Appendix E: AVOIDBROAD.C

A C program that calculates an lbf using the avoidance model, avoidbroad.c:

```
extern int  debug = 0;
extern double str2num(char[99]);

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <time.h>

#define MAX 257
#define LINELENGTH 1024
#define     LIMIT 1000

main (args,argstr)
int args;
char *argstr[];

{ /* beginning of main program */

FILE *infile;
FILE *outfile1, *outfile2;
double      x[MAX],y[MAX],z[MAX];
char  lines[MAX][LINELENGTH];
int   nn[MAX],order[MAX][MAX];
int   i,j,k,l, m,n,p, line,found, max,swap;
int   jj, ni, nj;
int   a,b,d,v,w;
int   nlimit;
char  *inpline;
char  inf[LINELENGTH];
char  of1[LINELENGTH];
char  of2[LINELENGTH];
double      btemp,field,broad[4001][MAX],distance[MAX][MAX],minimum,
dist,h;
double      sum[3100][MAX];
int   row,col;

/* for random number generation */
unsigned short xsubi[3];
int     zero = 0;
int   one = 1;
int   two = 2;

/* trying a new type  */
time_t  presenttime;
time_t  tloc;
```

```
/* initializing random number generator */
presenttime = time(&tloc);
jj = 0xffff &  presenttime;
xsubi[zero] = (unsigned short ) jj;

jj = 0xffff0000 &  presenttime;
jj = jj / 0xffff;
xsubi[one] = (unsigned short ) jj;

jj = ((int) xsubi[zero]) * ((int) xsubi[one]);
xsubi[two] = (unsigned short ) jj;









if ( ( infile = fopen(argstr[1],"r") ) != (FILE *) NULL )
      {/* success */}
else
      {
      fprintf(stderr,"load: file '%s' not found.\n",argstr[1]);
      return(1);
      }

i = 0;
while ( (argstr[1][i] != '.' ) && ( argstr[1][i] != '\0') )
      {
      inf[i] = argstr[1][i];
      i = i + 1;
      }
inf[i] = '\0';
/* printf("output filename root = %s\n",inf); */



line = 1;
while ( !(feof(infile)) )
      { /*beginning of while loop to read file */

      /* reading line of input file  */
      fscanf(infile,"%[^\n]\n",&lines[line][0]);
      lines[line][-1] = '0';
/*    printf("'%s'\n",inpline);  */

      /* i = fscanf(infile,"%d %f %f
%f\n",&nn[line],&x[line],&y[line],&z[line]);
            */

      inpline = &lines[line][0];
```

```
      i = findtextfield(inpline,1);
      nn[line] = (int) str2num(&inpline[i]) ;

      j = findtextfield(inpline,2);
      x[line] = str2num(&inpline[j]) ;

      k = findtextfield(inpline,3);
      y[line] = str2num(&inpline[k]) ;

      l = findtextfield(inpline,4);
      z[line] = str2num(&inpline[l]) ;
      /* printf("number: '%s'\n",&inpline[l]); */

/*
      printf("%d\t%d\t%f\t%f\t%f\n",line,nn[line],x[line],y[line],z[lin
e]);
*/

      max = line;
      line = line + 1;

      } /*end of while loop to read file */

fclose(infile);


/* ================================ adapted  from random2.c */

/* randomly picking nsamples input lines from the textbuffer */

/*
printf("p loop \n");
*/

/* return(1); */
/* ================================ setting sum matrix to 0*/
for (i = 0; i < 3100 ; i = i + 1)
      {
      for (j = 0; j <= max ; j = j + 1)
            {
            sum[i][j] = 0.0;
            }
      }

/* ================================ setting distance matrix to 0*/
for (i = 0; i <= max ; i = i + 1)
      {
      for (j = 0; j <= max ; j = j + 1)
            {
            distance[i][j] = 0.0;
            }
      }
/* ================================ setting order matrix i,j=j*/
for (i = 0; i <= max ; i = i + 1)
      {
```

```
      for (j = 0; j <= max ; j = j + 1)
            {
            order[i][j] = j;
            }
      }

/* ================================ calculating distance matrix */
for (i = 1; i <= max ; i = i + 1)
      {
      for (j = 1; j <= max ; j = j + 1)
            {
            distance[i][j] =          pow(x[i]-x[j],2);
            distance[i][j] = distance[i][j] + pow(y[i]-y[j],2);
            distance[i][j] = distance[i][j] + pow(z[i]-z[j],2);
            distance[i][j] = sqrt(distance[i][j]);

            }

      }

/* ================================ calculating order matrix */
minimum = 100;
for (i=0; i < max; i=i+1)
      {
      order[i][0] = i;
      minimum = 70;
      l = 1;
      swap = 0;
      while (l <= max)
            {
            j = l-1;
            found = 0;
            while (j <=max && found == 0)
                  {
                  j = j + 1;
                  swap = 0;
                  for (k = 0; k <= l; k = k + 1)
                        {
                        if (distance[order[i][k]][order[i][j]] <
minimum)
                              {
                              swap = 1;
                              }
                        }
                  if (swap == 0)
                        {
                        found = 1;
                        d = order[i][j];
                        order[i][j] = order[i][l];
                        order[i][l] = d;
                        l = l + 1;
                        }

                  }
            if (swap == 1)
```

```
                  {
                  minimum = minimum - 1;
                  swap = 0;
                  }
            }
      }
/* ================================ calculating r6 matrix */
for (p = 2; p <= (max) ; p = p + 1)
      {
         for (i = 1; i <= max ; i = i + 1)
            {
                  for (k = 1; k <= p; k = k + 1)
                  {
                  for (j = 1; j <= p; j = j + 1)
                        {
                        if (k!=j)
                              {
                              sum[k+p*(i-1)][1] = sum[k+p*(i-1)][1] +
pow(distance[order[i][k]][order[i][j]],-6);
                              }
                        }
                  }

            }
      }
for (i = 1; i <= (3025) ; i = i + 1)
      {
      for (j = 2; j <= max ; j = j + 1)
            {
            if (sum[i][j]> 1e-15)
                  {
                  sum[i][j] = (sqrt(sum[i][j])*19525.97);
                  }

            }
      }


/*calculates sum of gaussian broadening functions*/

for (k = 2; k <= max; k = k + 1)
      {
      for (j = 0; j <= 3999; j = j + 1)
            {
            field = j/20.0 - 99.975;

            for (i = 0; i <= 3025; i = i + 1)
                  {
                  if (sum[i][k] > 1e-15)
                        {
                        btemp = ( exp ( -0.693 * (field/(.5 *
sum[i][k]))* (field/(.5 * sum[i][k]))))/sum[i][k];
                        broad[j][k] = broad[j][k] + btemp;
                        }
                  }
```

```
            }
      }


/* this prints column titles */
printf("sum of r-6\t");
for (j = 2; j <= max ; j = j + 1)
      {
      printf("%d\t",j);
      }
printf("\n");

/* this prints rows of broadening amplitudes */
for (i = 0; i <= (3999) ; i = i + 1)
      {
      field = i/20.0 - 99.975;
      printf("%e\t",field);
      for (j = 2; j <= max ; j = j + 1)
            {
            printf("%e\t",broad[i][j]);
            }
      printf("\n");
      }


} /* end of main program */
```

Appendix F :KINSUM.AWK

An awk script that sums together lbf functions to simulate a normal distribution, kinsum.awk:

```
#
BEGIN {
nrow=4000;


for (i=0;i<nrow+1;i=i+1)
      {

      for (k=1;k<10;k=k+1)
            {
            b[i,k] = 0.0
            }
      }




}
{
if (NR == 2)
      {
      ncol = NF
      }
if (NR > 0)
      {
      S[NR,1] = $1;
      for (x=1;x<ncol+1;x=x+1)
            {
            S[NR,x] =$x;
            csum[x] = csum[x] + $x
            }
      }

}
END {
nrow = NR
for (i=1;i<nrow+1;i=i+1)
      {
      for (x=2;x<ncol+1;x=x+1)
            {
            S[i,x] = S[i,x]/csum[x]
            }
      }

l[1]=.05;
l[2]=.1;
```

```
l[3]=.25;
l[4]=.5;
l[5]=.75;
l[6]=.90;
l[7]=.95;
for (i=0;i<9;i=i+1)
      {
      c[i]= ncol * l[i]
      }


for (i=1;i<nrow+1;i=i+1)
      {
      b[i,0]=S[i,1];
      b[i,8]=S[i,ncol];
      }
w=1
for (m=1;m<ncol+1;m=m+1)
      {
      w = w * m
      }


for (i=1;i<nrow+1;i=i+1)
      {
      for (j=2;j<ncol+1;j=j+1)
            {
            for (k=1;k<8;k=k+1)
                  {
                  y=1
                  z=1
                  w=1
                  for (m=1;m<ncol+1;m=m+1)
                        {
                        w = w * m
                        }
                  for (m=1;m<j+1;m=m+1)
                        {
                        y = y * m
                        }

                  for (n=1;n<ncol+1-j;n=n+1)
                        {
                        z = z * n
                        }
                  if (z<1)
                        {
                        z=1
                        }
                  if (y<1)
                        {
                        y=1
                        }
                  x  = (l[k]^j) *((1-l[k])^(ncol-j))*w/(z*y)
```

```
                    if (x < 1.1 && x > .0001)
                          {
                          b[i,k] = b[i,k] + x* S[i,j];
                          }
                    }

              }
        }


for (i=1;i<nrow+1;i=i+1)
        {
        for (j=0;j<9;j=j+1)
              {
              printf("%f\t",b[i,j])
              }
        printf("\n")
        }

}
```

Appendix G: XCONVOLUTE.AWK

An awk script that convolutes a spectrum with a variable width function, xconvolute.awk:

```
#
BEGIN {
n=0;
W=15;
}
{
n=n+1;
s[NR]=$2;
f[NR]=$1;
sum=sum+$2;
if (NR == 1)
      {
      startfield=$1;
      }
}
END {
endfield=f[n];
step=(startfield-endfield);
for (i=1;i<n+1;i=i+1)
      {
      for (j=1;j<n+1;j=j+1)
            {
            delta=((f[i]-f[j])^2)^.5;
            factor[i]=factor[i]+((.5+(1.1753*(delta/W)+1.5499)^(-
3.2484))*s[j])/sum;
            }
      }
for (i=1;i<n+1;i=i+1)
      {
      w=W*2*factor[i];
      for (j=1;j<n+1;j=j+1)
            {
            S[j] = S[j] + (s[i] /((4*((f[j]-f[i])/w)*((f[j]-
f[i])/w))+1))/w;
            }
      }
for (i=1;i<n+1;i=i+1)
      {
      printf("%f\t%f\t",f[i],S[i]);
      printf("\n");
      }

    }
```

Appendix H : OBLATE.C

A C program that chooses random sites on the surface of an oblate ellipsoid, oblate.c:

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

/*
double erand48 (unsigned short xsubi[3]);
*/


/* using malloc(3x) */
#include <sys/types.h>
#include <malloc.h>

#include <time.h>
 /*time usage:
 time_t time(time_t *tloc); */



int   debug = 0;

extern  double    str2num(char *temp);

#define maxlinelength 256
#define maxlinesinfile 1024

#define MAX 1000
#define pi 3.1415927


/* actual beginning of main program */
void main (args,argstr)

int args;
char *argstr[];

{ /* beginning of main program */

int   p,i,j,k,n,nlimit,a1,b1,t;
double      a,b,c,e,s,at,bt,m;
double      x1, x2, y1, y2, z1, z2;
FILE  *infile, *outfile1;
char  filename[20];
int   nsamples;   /* number of samples user desires */
int   linesfound; /* number of lines found in input text stream */
char  textbuffer[maxlinesinfile][maxlinelength]; /* buffer of lines in
text */
char  **endptr;
```

```
unsigned short xsubi[4];
char  instring[10];
char  of1[256], of2[256];
double      r; /* radius of sphere */
double      min;
int         dia,rmin;
double      x[MAX];
double      y[MAX];
double      z[MAX];
double      d[56];
double      theta[56];
double      phi[56];
double      dist;
double      dx,dy;

/* trying a new type  */
time_t      presenttime;
time_t      tloc;

int   zero, one, two;



/* initializing random number generator */
      /* dumb initialization for the moment */
xsubi[0] = (unsigned short) 2;
xsubi[1] = (unsigned short) 1;
xsubi[2] = (unsigned short) 0;
      /* better initialization */
/* presenttime is of type time_t, which is equivalent to "long" */
presenttime = time(&tloc);

/* pointers to xsubi -- we can change these if we don't like something
about
                  the way the random numbers are coming out
                  setting all these to the same integer value will
                  prevent the routine from generating a different
                  series of numbers each time it is called numbers */
zero = 0;
one = 1;
two = 2;


/* randomly picking nsamples input lines from the textbuffer */



      r=34;
      for (m = 1.5; m < 10; m = m + 1)
            {
            for (n = 0; n < 10; n = n + 1)
                  { /* beginning of n loop */


                  presenttime = time(&tloc);
```

```
                    j = 0xffff &  presenttime;
                    xsubi[zero] = (unsigned short ) j;

                    j = 0xffff0000 &  presenttime;
                    j = j / 0xffff;
                    xsubi[one] = (unsigned short ) j;

                    j = (n + (int) xsubi[zero]) * ((int) xsubi[one]);
                    xsubi[two] = (unsigned short ) j;

                    for (k = 0; k < 56; k = k + 1)
                          {
                          x[k]=0.0;
                          y[k]=0.0;
                          z[k]=0.0;
                          }
                    c = 0;
                    i = 0;
                    e = log10(10*log10(m));
                    a = sqrt((2*r*r)/( 1 + ((1-e*e)/(2*e)) * log(
(1+e)/(1-e))));
                    b = a * sqrt(1-e*e);
                    s = a/b;
                    min = 7.0;

                    printf("a = %f, b = %f,e = %f \n ",a,b,e);




                    while ((i < 56))
                          { /* beginning of first i loop */
                          /* erand48 returns a number in the interval
[0.0,1.0] */
                          c=c+1;
                          i=i+1;
                          x[i] = (erand48(xsubi)-.5)*s;
                          y[i] = (erand48(xsubi) -.5)*s;
                          z[i] = (erand48(xsubi) -.5);
                          t = (sqrt((x[i]*x[i])/(s*s) + (y[i]*y[i])/(s*s)
+ (z[i]*z[i]) ));

                          if (t < 1.0)
                                {
                                bt = sqrt( (x[i]*x[i])/(s*s) +
(y[i]*y[i])/(s*s) + z[i]*z[i]);
                                at = s * bt;
                                x[i] = (a/at) * x[i];
                                y[i] = (a/at) * y[i];
                                z[i] = (b/bt) * z[i];
                                k = i - 1;
                                      for (p = 0; p < i; p=p+1)
                                            {
                                            dist =        (x[i]-
x[1])*(x[i]-x[1]);
```

```
                                        dist = dist + (y[i]-
y[1])*(y[i]-y[1]);
                                        dist = dist + (z[i]-
z[1])*(z[i]-z[1]);
                                        dist = sqrt(dist);
                                        if ( dist < min )
                                              {
                                              i = k;
                                              }
                                        if ( c > 1000000)
                                              {
                                              i = 56;
                                              }

                                        }
                              }
                        if (t > 1.)
                              {
                              i=i-1;
                              }

                        } /* end of first i loop */


                        a1=a;
                        b1=b;
                              sprintf(filename,"%d-55oblate%d-%d-
%d",k,a1,b1,n);
                              outfile1 = fopen(filename,"w");


                        for (i = 1; i < k+1; i = i + 1)
                              {

      fprintf(outfile1,"%d\t%f\t%f\t%f\n",i,x[i],y[i],z[i]);

                              } /* end of second i loop */

                              c = 0;
                              fclose(outfile1);

                  } /* end of n loop */
      }/* end of m loop */


} /* end of main program */
```

Appendix I: PROLATE.C

C program that chooses random sites on the surface of an prolate ellipsoid, prolate.c:

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

/*
double erand48 (unsigned short xsubi[3]);
*/


/* using malloc(3x) */
#include <sys/types.h>
#include <malloc.h>

#include <time.h>
 /*time usage:
 time_t time(time_t *tloc); */



int   debug = 0;

extern  double    str2num(char *temp);

#define maxlinelength 256
#define maxlinesinfile 1024

#define MAX 1000
#define pi 3.1415927


/* actual beginning of main program */
void main (args,argstr)

int args;
char *argstr[];

{ /* beginning of main program */

int   p,i,j,k,n,nlimit,a1,b1,t;
double      a,b,c,e,s,at,bt,m;
double      x1, x2, y1, y2, z1, z2;
FILE  *infile, *outfile1;
char  filename[20];
int   nsamples;   /* number of samples user desires */
int   linesfound; /* number of lines found in input text stream */
```

```
char  textbuffer[maxlinesinfile][maxlinelength]; /* buffer of lines in
text */
char  **endptr;
unsigned short xsubi[4];
char  instring[10];
char  of1[256], of2[256];
double      r; /* radius of sphere */
double      min;
int         dia,rmin;
double      x[MAX];
double      y[MAX];
double      z[MAX];
double      d[56];
double      theta[56];
double      phi[56];
double      dist;
double      dx,dy;

/* trying a new type  */
time_t      presenttime;
time_t      tloc;

int   zero, one, two;



/* initializing random number generator */
      /* dumb initialization for the moment */
xsubi[0] = (unsigned short) 2;
xsubi[1] = (unsigned short) 1;
xsubi[2] = (unsigned short) 0;
      /* better initialization */
/* presenttime is of type time_t, which is equivalent to "long" */
presenttime = time(&tloc);

/* pointers to xsubi -- we can change these if we don't like something
about
                  the way the random numbers are coming out
                  setting all these to the same integer value will
                  prevent the routine from generating a different
                  series of numbers each time it is called numbers */
zero = 0;
one = 1;
two = 2;


/* randomly picking nsamples input lines from the textbuffer */



      r=34;
      for (m = 1.5; m < 10; m = m + 1)
            {
            for (n = 0; n < 10; n = n + 1)
                  { /* beginning of n loop */
```

```
                    presenttime = time(&tloc);
                    j = 0xffff &  presenttime;
                    xsubi[zero] = (unsigned short ) j;

                    j = 0xffff0000 &  presenttime;
                    j = j / 0xffff;
                    xsubi[one] = (unsigned short ) j;

                    j = (n + (int) xsubi[zero]) * ((int) xsubi[one]);
                    xsubi[two] = (unsigned short ) j;

                    for (k = 0; k < 56; k = k + 1)
                          {
                          x[k]=0.0;
                          y[k]=0.0;
                          z[k]=0.0;
                          }
                    c = 0;
                    i = 0;
                    e = log10(10*log10(m));
                    a = sqrt((2*r*r)/( (1-e*e) + sqrt(1-e*e) *  asin(e)/e
));
                    b = a * sqrt(1-e*e);
                    s = a/b;
                    min = 7.0;

                    printf("a = %f, b = %f,e = %f \n ",a,b,e);




                    while ((i < 56))
                          { /* beginning of first i loop */
                          /* erand48 returns a number in the interval
[0.0,1.0] */
                          c=c+1;
                          i=i+1;
                          x[i] = (erand48(xsubi)-.5);
                          y[i] = (erand48(xsubi) -.5);
                          z[i] = (erand48(xsubi) -.5)*s;
                          t = (sqrt((x[i]*x[i]) + (y[i]*y[i]) +
(z[i]*z[i])/(s*s) ));

                          if (t < 1.0)
                                {
                                bt = sqrt( (x[i]*x[i]) + (y[i]*y[i]) +
z[i]*z[i]/(s*s));
                                at = s * bt;
                                x[i] = (b/bt) * x[i];
                                y[i] = (b/bt) * y[i];
                                z[i] = (a/at) * z[i];
                                k = i - 1;
                                      for (p = 0; p < i; p=p+1)
```

```
                                        {
                                        dist =        (x[i]-
x[1])*(x[i]-x[1]);
                                        dist = dist + (y[i]-
y[1])*(y[i]-y[1]);
                                        dist = dist + (z[i]-
z[1])*(z[i]-z[1]);
                                        dist = sqrt(dist);
                                        if ( dist < min )
                                              {
                                              i = k;
                                              }
                                        if ( c > 1000000)
                                              {
                                              i = 56;
                                              }

                                        }
                                  }
                            if (t > 1.)
                                  {
                                  i=i-1;
                                  }

                            } /* end of first i loop */


                            a1=a;
                            b1=b;
                                  sprintf(filename,"%d-55prolate%d-%d-
%d",k,a1,b1,n);
                                  outfile1 = fopen(filename,"w");


                            for (i = 1; i < k+1; i = i + 1)
                                  {

      fprintf(outfile1,"%d\t%f\t%f\t%f\n",i,x[i],y[i],z[i]);

                                  } /* end of second i loop */

                                  c = 0;
                                  fclose(outfile1);

                      } /* end of n loop */
      }/* end of m loop */


} /* end of main program */
```

Appendix J: RANSOL.C

A C program that chooses random sites in a 3-D volume, ransol.c

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

/*
double erand48 (unsigned short xsubi[3]);
*/


/* using malloc(3x) */
#include <sys/types.h>
#include <malloc.h>

#include <time.h>
 /*time usage:
 time_t time(time_t *tloc); */



int   debug = 0;

extern  double    str2num(char *temp);

#define maxlinelength 256
#define maxlinesinfile 1024

#define MAX 1000
#define pi 3.1415927
#define min 6

/* actual beginning of main program */
void main (args,argstr)

int args;
char *argstr[];

{ /* beginning of main program */

int   p,i,j,k,n,nlimit,mMolar;
double      a,b,c;
double      x1, x2, y1, y2, z1, z2;
FILE  *infile, *outfile1;
char  filename[20];
int   nsamples;   /* number of samples user desires */
int   linesfound; /* number of lines found in input text stream */
char  textbuffer[maxlinesinfile][maxlinelength]; /* buffer of lines in
text */
char  **endptr;
```

```
unsigned short xsubi[4];
char  instring[10];
char  of1[256], of2[256];
double      r = 25; /* radius of sphere */
double      x[MAX];
double      y[MAX];
double      z[MAX];
double      d[56];
double      theta[56];
double      phi[56];
double      dist, size, mM;
double      dx,dy;

/* trying a new type  */
time_t      presenttime;
time_t      tloc;

int   zero, one, two;



/* initializing random number generator */
      /* dumb initialization for the moment */
xsubi[0] = (unsigned short) 2;
xsubi[1] = (unsigned short) 1;
xsubi[2] = (unsigned short) 0;
      /* better initialization */
/* presenttime is of type time_t, which is equivalent to "long" */
presenttime = time(&tloc);

/* pointers to xsubi -- we can change these if we don't like something
about
                  the way the random numbers are coming out
                  setting all these to the same integer value will
                  prevent the routine from generating a different
                  series of numbers each time it is called numbers */
zero = 0;
one = 1;
two = 2;


/* randomly picking nsamples input lines from the textbuffer */



      for (n = 0; n < 11; n = n +1)
            { /* beginning of n loop */
            mM = pow(2,n);
            mMolar = pow(2,n);
            sprintf(filename,"200solution%d-%d",min,mMolar);
            outfile1 = fopen(filename,"w");

            presenttime = time(&tloc);
            j = 0xffff &  presenttime;
            xsubi[zero] = (unsigned short ) j;
```

```
            j = 0xffff0000 &  presenttime;
            j = j / 0xffff;
            xsubi[one] = (unsigned short ) j;

            j = (n + (int) xsubi[zero]) * ((int) xsubi[one]);
            xsubi[two] = (unsigned short ) j;


            i = 0;

            size = pow( ((200*1660)/(mM/1000)),.33333333);

            while ((i < 200))
                  { /* beginning of first i loop */
                  /* erand48 returns a number in the interval [0.0,1.0]
*/
                  c=c+1;
                  i=i+1;
                  x[i] = (erand48(xsubi)-.5)*size;
                  y[i] = (erand48(xsubi) -.5)*size;
                  z[i] = (erand48(xsubi) -.5)*size;

                  k = i - 1;
                        for (p = 0; p < i; p=p+1)
                              {
                              dist =        (x[i]-x[1])*(x[i]-x[1]);
                              dist = dist + (y[i]-y[1])*(y[i]-y[1]);
                              dist = dist + (z[i]-z[1])*(z[i]-z[1]);
                              dist = sqrt(dist);
                              if ( dist < min )
                                    {
                                    i = k;
                                    }
                              if ( c > 2000000)
                                    {
                                    i = 200;
                                    }

                              }
                  } /* end of first i loop */

            for (i = 1; i < 200; i = i + 1)
                  {

      fprintf(outfile1,"%d\t%f\t%f\t%f\n",i,x[i],y[i],z[i]);

                  } /* end of second i loop */

                  c = 0;
                  fclose(outfile1);
            } /* end of n loop */

} /* end of main program */
```

Appendix K: SOLBROAD.C

A C program that calculates an lbf for a 3-D solution, solbroad.c:

```
extern int  debug = 0;
extern double str2num(char[99]);

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <time.h>

#define MAX 257
#define LINELENGTH 1024
#define     LIMIT 1000

main (args,argstr)
int args;
char *argstr[];

{ /* beginning of main program */

FILE *infile;
FILE *outfile1, *outfile2;
double      x[MAX],y[MAX],z[MAX];
char  lines[MAX][LINELENGTH];
int   nn[MAX],order[MAX];
int   i,j,k,l, m,n,p, line, max,maxused,swap;
int   jj, ni, nj;
int   a,b,d,v,w;
int   nlimit;
char  *inpline;
char  inf[LINELENGTH];
char  of1[LINELENGTH];
char  of2[LINELENGTH];
double      btemp,field,broad[3999],distance[MAX][MAX], dist,h;
double      sum[3100];
int   row,col;

/* for random number generation */
unsigned short xsubi[3];
int     zero = 0;
int   one = 1;
int   two = 2;

/* trying a new type  */
time_t  presenttime;
time_t  tloc;



/* initializing random number generator */
```

```
presenttime = time(&tloc);
jj = 0xffff &  presenttime;
xsubi[zero] = (unsigned short ) jj;

jj = 0xffff0000 &  presenttime;
jj = jj / 0xffff;
xsubi[one] = (unsigned short ) jj;

jj = ((int) xsubi[zero]) * ((int) xsubi[one]);
xsubi[two] = (unsigned short ) jj;









if ( ( infile = fopen(argstr[1],"r") ) != (FILE *) NULL )
      {/* success */}
else
      {
      fprintf(stderr,"load: file '%s' not found.\n",argstr[1]);
      return(1);
      }

i = 0;
while ( (argstr[1][i] != '.' ) && ( argstr[1][i] != '\0') )
      {
      inf[i] = argstr[1][i];
      i = i + 1;
      }
inf[i] = '\0';
/* printf("output filename root = %s\n",inf); */



line = 1;
while ( !(feof(infile)) )
      { /*beginning of while loop to read file */

      /* reading line of input file  */
      fscanf(infile,"%[^\n]\n",&lines[line][0]);
      lines[line][-1] = '0';
/*    printf("'%s'\n",inpline);  */

      /* i = fscanf(infile,"%d %f %f
%f\n",&nn[line],&x[line],&y[line],&z[line]);
            */

      inpline = &lines[line][0];

      i = findtextfield(inpline,1);
      nn[line] = (int) str2num(&inpline[i]) ;
```

```
      j = findtextfield(inpline,2);
      x[line] = str2num(&inpline[j]) ;

      k = findtextfield(inpline,3);
      y[line] = str2num(&inpline[k]) ;

      l = findtextfield(inpline,4);
      z[line] = str2num(&inpline[l]) ;
      /* printf("number: '%s'\n",&inpline[l]); */

/*
      printf("%d\t%d\t%f\t%f\t%f\n",line,nn[line],x[line],y[line],z[lin
e]);
*/

      max = line;
      line = line + 1;

      } /*end of while loop to read file */

fclose(infile);


maxused = .3 * max;


/* ================================ adapted  from random2.c */

/* randomly picking nsamples input lines from the textbuffer */

/*
printf("p loop \n");
*/

/* return(1); */
/* ================================ setting sum matrix to 0*/
for (i = 0; i < maxused ; i = i + 1)
      {
      sum[i] = 0.0;
      }

/* ================================ setting distance matrix to 0*/
for (i = 0; i <= max ; i = i + 1)
      {
      for (j = 0; j <= max ; j = j + 1)
            {
            distance[i][j] = 0.0;
            }
      }
/* ================================ setting order matrix i=i*/
for (i = 0; i <= max ; i = i + 1)
      {
      order[i] = i;
      }
```

```
/* ================================ calculating distance matrix */


      for (j = 1; j <= max ; j = j + 1)  /* 1st column dist from 0,0,0
*/
            {
            distance[0][j] =          pow(0.0-x[j],2);
            distance[0][j] = distance[0][j] + pow(0.0-y[j],2);
            distance[0][j] = distance[0][j] + pow(0.0-z[j],2);
            distance[0][j] = sqrt(distance[0][j]);

            }


for (i = 1; i <= max ; i = i + 1)
      {
      for (j = 1; j <= max ; j = j + 1)
            {
            distance[i][j] =          pow(x[i]-x[j],2);
            distance[i][j] = distance[i][j] + pow(y[i]-y[j],2);
            distance[i][j] = distance[i][j] + pow(z[i]-z[j],2);
            distance[i][j] = sqrt(distance[i][j]);

            }

      }
/*
for (i = 1; i <= max ; i = i + 1)
      {
      for (j = 1; j <= max ; j = j + 1)
            {
            printf("%e\t",distance[i][j]);

            }
      printf("\n");

      }*/

/* ================================ calculating order vector */


      swap = 1;
      while (swap == 1)
            {
            swap = 0;
            for (j = 2; j <= max; j = j + 1)
                  {
                  if (distance[0][order[j]] < distance[0][order[j-1]])
                        {
                        d = order[j];
                        order[j] = order[j-1];
                        order[j-1] = d;
                        swap = 1;
                        }
                  }
```

```
            }

/* ================================ calculating r6 matrix */


for (i = 1; i <= maxused ; i = i + 1)
      {

      for (j = 1; j <= max; j = j + 1)
            {
            if (i!=j)
                  {
                  sum[i] = sum[i] + pow(distance[order[i]][order[j]],-
6);
                  }
            }
      }

for (i = 1; i <= maxused ; i = i + 1)
      {
            {
            if (sum[i]> 1e-15)
                  {
                  sum[i] = (sqrt(sum[i])*19525.97);
                  }

            }
      }

/*calculates sum of gaussian broadening functions*/


      for (j = 0; j <= 3999; j = j + 1)
            {
            field = j/20.0 - 99.975;

            for (i = 0; i <= 3025; i = i + 1)
                  {
                  if (sum[i] > 1e-15)
                        {
                        btemp = ( exp ( -0.693 * (field/(.5 * sum[i]))*
(field/(.5 * sum[i]))))/sum[i];

                        broad[j] = broad[j] + btemp;
                        }
                  }
            }




/* this prints broadening  */
for (i = 0; i <= (3999) ; i = i + 1)
      {
```

```
      field = i/20.0 - 99.975;
      printf("%e\t",field);
      printf("%e\t",broad[i]);
      printf("\n");
      }

} /* end of main program */
```

Appendix L: RANSURF.C

A C program that chooses random sites on a 2-D surface, ransurf.c:

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

/*
double erand48 (unsigned short xsubi[3]);
*/


/* using malloc(3x) */
#include <sys/types.h>
#include <malloc.h>

#include <time.h>
 /*time usage:
 time_t time(time_t *tloc); */



int   debug = 0;

extern  double    str2num(char *temp);

#define maxlinelength 256
#define maxlinesinfile 1024

#define MAX 2001
#define pi 3.1415927
#define min 6

/* actual beginning of main program */
void main (args,argstr)

int args;
char *argstr[];

{ /* beginning of main program */

int   p,i,j,k,n,nlimit,Areaperspin;
double      a,b,c;
double      x1, x2, y1, y2, z1, z2;
FILE  *infile, *outfile1;
char  filename[20];
int   nsamples;   /* number of samples user desires */
int   linesfound; /* number of lines found in input text stream */
char  textbuffer[maxlinesinfile][maxlinelength]; /* buffer of lines in
text */
char  **endptr;
```

```
unsigned short xsubi[4];
char  instring[10];
char  of1[256], of2[256];
double      r = 25; /* radius of sphere */
double      x[MAX];
double      y[MAX];
double      z[MAX];
double      d[MAX];
double      theta[MAX];
double      phi[MAX];
double      dist, size, areaperspin;
double      dx,dy;

/* trying a new type  */
time_t      presenttime;
time_t      tloc;

int   zero, one, two;



/* initializing random number generator */
      /* dumb initialization for the moment */
xsubi[0] = (unsigned short) 2;
xsubi[1] = (unsigned short) 1;
xsubi[2] = (unsigned short) 0;
      /* better initialization */
/* presenttime is of type time_t, which is equivalent to "long" */
presenttime = time(&tloc);

/* pointers to xsubi -- we can change these if we don't like something
about
                  the way the random numbers are coming out
                  setting all these to the same integer value will
                  prevent the routine from generating a different
                  series of numbers each time it is called numbers */
zero = 0;
one = 1;
two = 2;


/* randomly picking nsamples input lines from the textbuffer */
      for (n = 0; n < 7; n = n +1)
            { /* beginning of n loop */
            areaperspin = pow(2,n+7);
            Areaperspin = pow(2,n+7);
            sprintf(filename,"2000surface%d-%d",min,Areaperspin);
            outfile1 = fopen(filename,"w");

            presenttime = time(&tloc);
            j = 0xffff &  presenttime;
            xsubi[zero] = (unsigned short ) j;

            j = 0xffff0000 &  presenttime;
            j = j / 0xffff;
```

```
            xsubi[one] = (unsigned short ) j;

            j = (n + (int) xsubi[zero]) * ((int) xsubi[one]);
            xsubi[two] = (unsigned short ) j;


            i = 0;

            size = pow( ((2000*areaperspin)),.5);

            while ((i < 2000))
                  { /* beginning of first i loop */
                  /* erand48 returns a number in the interval [0.0,1.0]
*/
                  c=c+1;
                  i=i+1;
                  x[i] = (erand48(xsubi)-.5)*size;
                  y[i] = (erand48(xsubi) -.5)*size;
                  z[i] = 0.0;

                  k = i - 1;
                        for (p = 0; p < i; p=p+1)
                              {
                              dist =        (x[i]-x[1])*(x[i]-x[1]);
                              dist = dist + (y[i]-y[1])*(y[i]-y[1]);
                              dist = dist + (z[i]-z[1])*(z[i]-z[1]);
                              dist = sqrt(dist);
                              if ( dist < min )
                                    {
                                    i = k;
                                    }
                              if ( c > 2000000)
                                    {
                                    i = 200;
                                    }

                              }
                  } /* end of first i loop */

            for (i = 1; i < 2000; i = i + 1)
                  {

      fprintf(outfile1,"%d\t%f\t%f\t%f\n",i,x[i],y[i],z[i]);

                  } /* end of second i loop */

                  c = 0;
                  fclose(outfile1);
            } /* end of n loop */



} /* end of main program */
```

Appendix M: RANLINE.C

A C program that chooses random sites on a 1-D line, ranline.c

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

/*
double erand48 (unsigned short xsubi[3]);
*/


/* using malloc(3x) */
#include <sys/types.h>
#include <malloc.h>

#include <time.h>
 /*time usage:
 time_t time(time_t *tloc); */



int   debug = 0;

extern  double    str2num(char *temp);

#define maxlinelength 256
#define maxlinesinfile 1024

#define MAX 2001
#define pi 3.1415927
#define min 6

/* actual beginning of main program */
void main (args,argstr)

int args;
char *argstr[];

{ /* beginning of main program */

int   p,i,j,k,n,nlimit,Lengthperspin;
double      a,b,c;
double      x1, x2, y1, y2, z1, z2;
FILE  *infile, *outfile1;
char  filename[20];
int   nsamples;   /* number of samples user desires */
int   linesfound; /* number of lines found in input text stream */
```

```
char  textbuffer[maxlinesinfile][maxlinelength]; /* buffer of lines in
text */
char  **endptr;
unsigned short xsubi[4];
char  instring[10];
char  of1[256], of2[256];
double      r = 25; /* radius of sphere */
double      x[MAX];
double      y[MAX];
double      z[MAX];
double      d[MAX];
double      theta[MAX];
double      phi[MAX];
double      dist, size, lengthperspin;
double      dx,dy;

/* trying a new type  */
time_t      presenttime;
time_t      tloc;

int   zero, one, two;



/* initializing random number generator */
      /* dumb initialization for the moment */
xsubi[0] = (unsigned short) 2;
xsubi[1] = (unsigned short) 1;
xsubi[2] = (unsigned short) 0;
      /* better initialization */
/* presenttime is of type time_t, which is equivalent to "long" */
presenttime = time(&tloc);

/* pointers to xsubi -- we can change these if we don't like something
about
                  the way the random numbers are coming out
                  setting all these to the same integer value will
                  prevent the routine from generating a different
                  series of numbers each time it is called numbers */
zero = 0;
one = 1;
two = 2;


/* randomly picking nsamples input lines from the textbuffer */
      for (n =3; n < 11; n = n +1)
            { /* beginning of n loop */
            lengthperspin = n * n;
            Lengthperspin = n * n;
            sprintf(filename,"2000line%d-%d",min,Lengthperspin);
            outfile1 = fopen(filename,"w");

            presenttime = time(&tloc);
            j = 0xffff &  presenttime;
            xsubi[zero] = (unsigned short ) j;
```

```
            j = 0xffff0000 &  presenttime;
            j = j / 0xffff;
            xsubi[one] = (unsigned short ) j;

            j = (n + (int) xsubi[zero]) * ((int) xsubi[one]);
            xsubi[two] = (unsigned short ) j;


            i = 0;

            size = 2000 * lengthperspin;

            while ((i < 2000))
                  { /* beginning of first i loop */
                  /* erand48 returns a number in the interval [0.0,1.0]
*/
                  c=c+1;
                  i=i+1;
                  x[i] = (erand48(xsubi)-.5)*size;
                  y[i] = 0.0;
                  z[i] = 0.0;

                  k = i - 1;
                        for (p = 0; p < i; p=p+1)
                              {
                              dist =        (x[i]-x[1])*(x[i]-x[1]);
                              dist = dist + (y[i]-y[1])*(y[i]-y[1]);
                              dist = dist + (z[i]-z[1])*(z[i]-z[1]);
                              dist = sqrt(dist);
                              if ( dist < min )
                                    {
                                    i = k;
                                    }
                              if ( c > 2000000)
                                    {
                                    i = 200;
                                    }

                              }
                  } /* end of first i loop */

            for (i = 1; i < 2000; i = i + 1)
                  {

      fprintf(outfile1,"%d\t%f\t%f\t%f\n",i,x[i],y[i],z[i]);

                  } /* end of second i loop */

                  c = 0;
                  fclose(outfile1);
            } /* end of n loop */
```

```
} /* end of main program */
```

Appendix N: SPSH.C

A C program that chooses sites in spherical shell, spsh.c:

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

/*
double erand48 (unsigned short xsubi[3]);
*/


/* using malloc(3x) */
#include <sys/types.h>
#include <malloc.h>

#include <time.h>
 /*time usage:
 time_t time(time_t *tloc); */



int   debug = 0;

extern  double    str2num(char *temp);

#define maxlinelength 256
#define maxlinesinfile 1024

#define MAX 1000
#define pi 3.1415927


/* actual beginning of main program */
void main (args,argstr)

int args;
char *argstr[];

{ /* beginning of main program */

int   p,i,j,k,n,m,nlimit;
double      a,b,c;
double      x1, x2, y1, y2, z1, z2;
FILE  *infile, *outfile1;
char  filename[20];
int   nsamples;   /* number of samples user desires */
int   linesfound; /* number of lines found in input text stream */
```

```
char  textbuffer[maxlinesinfile][maxlinelength]; /* buffer of lines in
text */
char  **endptr;
unsigned short xsubi[4];
char  instring[10];
char  of1[256], of2[256];
double      r; /* radius of sphere */
double      min;
int         dia,rmin;
double      x[MAX];
double      y[MAX];
double      z[MAX];
double      d[56];
double      theta[56];
double      phi[56];
double      dist;
double      dx,dy;

/* trying a new type  */
time_t      presenttime;
time_t      tloc;

int   zero, one, two;



/* initializing random number generator */
      /* dumb initialization for the moment */
xsubi[0] = (unsigned short) 2;
xsubi[1] = (unsigned short) 1;
xsubi[2] = (unsigned short) 0;
      /* better initialization */
/* presenttime is of type time_t, which is equivalent to "long" */
presenttime = time(&tloc);

/* pointers to xsubi -- we can change these if we don't like something
about
                  the way the random numbers are coming out
                  setting all these to the same integer value will
                  prevent the routine from generating a different
                  series of numbers each time it is called numbers */
zero = 0;
one = 1;
two = 2;


/* randomly picking nsamples input lines from the textbuffer */

for (r = 35; r < 39; r = r + 5)
      {
      dia = r * 2;
      for (min = 6; min < 7; min = min + 2)
            {
            rmin = min;
            for (n = 0; n < 20; n = n + 1)
```

```
{ /* beginning of n loop */


presenttime = time(&tloc);
j = 0xffff &  presenttime;
xsubi[zero] = (unsigned short ) j;

j = 0xffff0000 &  presenttime;
j = j / 0xffff;
xsubi[one] = (unsigned short ) j;

j = (n + (int) xsubi[zero]) * ((int) xsubi[one]);
xsubi[two] = (unsigned short ) j;

for (k = 0; k < 56; k = k + 1)
      {
      x[k]=0.0;
      y[k]=0.0;
      z[k]=0.0;
      }
c = 0;
i = 0;
while ((i < 56))
      { /* beginning of first i loop */
      /* erand48 returns a number in the interval
[0.0,1.0] */
      c=c+1;
      i=i+1;
      a = r + 1.0 ;
      while ((a > r) || (a < r-20.))
            {
            x[i] = (erand48(xsubi)-.5)*r*2;
            y[i] = (erand48(xsubi) -.5)*r*2;
            z[i] = (erand48(xsubi) -.5)*r*2;
            a = (sqrt(x[i]*x[i] + y[i]*y[i] +
z[i]*z[i]));
            }
      k = i - 1;
            for (p = 0; p < i; p=p+1)
                  {
                  dist =        (x[i]-x[1])*(x[i]-
x[1]);
                  dist = dist + (y[i]-y[1])*(y[i]-
y[1]);
                  dist = dist + (z[i]-z[1])*(z[i]-
z[1]);
                  dist = sqrt(dist);
                  if ( dist < min )
                        {
                        i = k;
                        }
                  if ( c > 1000000)
                        {
                        i = 56;
                        }
```

```
                                    }
                              } /* end of first i loop */


                                    sprintf(filename,"%d-55sphereshell%d-%d-
%d",k,dia,rmin,n);
                                    outfile1 = fopen(filename,"w");

                              for (i = 1; i < k+1; i = i + 1)
                                    {

      fprintf(outfile1,"%d\t%f\t%f\t%f\n",i,x[i],y[i],z[i]);

                                    } /* end of second i loop */

                                    c = 0;
                                    fclose(outfile1);

                        } /* end of n loop */

                  }/* end of min loop */

      }/* end of r loop */


} /* end of main program */
```

Appendix O: RANDOM.CPP

A C++ program that calculates an lbf for a random model using the repeated splittings method, random.cpp

```
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <string>
#include <ctime>
#include <time.h>

using namespace std;

ofstream outfile ("outfileData.txt");

ofstream outCoord ("outfileCoord.txt");

ofstream outError ("outfileError.txt");

ofstream outTime ("outfileTime.txt");

const int num_of_spins = 55;
const int MAX = 2000;
const int MAXangle = 360;
const int AXIS = 1000;
const int CONFIGURATIONS = 10;
const double INCREMENTS = 0.1;
const float DISTANCE_SPINS_MINIMUM = 10.0;
const float DISTANCE_SPINS_MAXIMUM = 70.0;
const int CIRCLE_RADIUS = 35;
double increments = 0.0;
int INCREMENT = 6;

/*********************************************************************
*****************************************************
This function will generate random numbers (I got it from the web)
*********************************************************************
*****************************************************/

void init_mm( );
int number_range( int from, int to );
int number_mm( void );
static  int     rgiState[2+55]; // leave this alone

float random(int TO)
{
      init_mm(); //seed the number generator
      int random = number_range( 0, TO);
      return random;
}
```

```
int number_mm( void )
{
      int *piState;
      int iState1;
      int iState2;
      int iRand;
      piState = &rgiState[2];
      iState1 = piState[-2];
      iState2 = piState[-1];
      iRand = ( piState[iState1] + piState[iState2] )
                  & ( ( 1 << 30 ) - 1 );
      piState[iState1]    = iRand;
      if ( ++iState1 == 55 )
            iState1 = 0;
      if ( ++iState2 == 55 )
            iState2 = 0;
      piState[-2] = iState1;
      piState[-1] = iState2;
      return iRand >> 6;
}

/*
* Generate a random number.
*/
int number_range( int from, int to )
{
      int power;
      int number;
      if ( ( to = to - from + 1 ) <= 1 )
            return from;
      for ( power = 2; power < to; power <<= 1 )
            ;
      while ( ( number = number_mm( ) & ( power - 1 ) ) >= to )
            ;
      return from + number;
}

/*
* this is the Mitchell-Moore algorithm from Knuth Volume II.
*/
void init_mm( )
{
      int *piState;
      int iState;
      piState     = &rgiState[2];
      piState[-2] = 55 - 55;
      piState[-1] = 55 - 24;
      piState[0]  = ( (int) time( NULL ) ) & ( ( 1 << 30 ) - 1 );
      piState[1]  = 1;
      for ( iState = 2; iState < 55; iState++ )
      {
            piState[iState] = ( piState[iState-1] + piState[iState-2] )
            & ( ( 1 << 30 ) - 1 );
      }
      return;
```

```
}
/*********************************************************************
*****************************************************/

inline float square(float x)
{
      return x*x;
}

inline float cube(float x)
{
      return x*x*x;
}

inline float ROOT_SQUARE_SUM(float x, float y, float z)
{
      return sqrt(square(x) + square(y) + square(z));
}

inline void swap(int x, int y)
{
      int temp;

      temp = x;
      x = y;
      y = temp;
}

struct Coordinates
{
      float X;
      float Y;
      float Z;
};

Coordinates SpinCoordinates[num_of_spins];

double tempRunningArray[num_of_spins][AXIS];

double output[num_of_spins][AXIS];

double distanceArray[num_of_spins][num_of_spins];

int orderArray[num_of_spins][num_of_spins];

class Spectrum
{
private:
      double runningArray[num_of_spins][AXIS];
      Coordinates coordinates;
public:
      Coordinates createCoordinates();
      void PutSpinsOnSphere();
      void initializeArray();
      Coordinates createField(int, int);
```

```
      float CalculateSplitValue(Coordinates, Coordinates, Coordinates);
      void Splitting(float, int);
      void RunProgram();
      void initializeDistanceArray();
      void initializeOrderArray();
      void initializeTempRunningArray();
      void initializeOutput();
      void outputData();
      void getTime(int, int);
};

Coordinates Spectrum::createCoordinates()
{
      Coordinates temporary;
      float theta, phi;

      temporary.X = static_cast<float>(random(MAXangle));
      temporary.Z = ((static_cast<float>(random(MAX*CIRCLE_RADIUS*2)))
/ MAX) - CIRCLE_RADIUS;

      theta = asin(temporary.Z /CIRCLE_RADIUS);
      phi = temporary.X;

      temporary.X = CIRCLE_RADIUS * cos(theta) * cos(phi);
      temporary.Y = CIRCLE_RADIUS * cos(theta) * sin(phi);

      return temporary;
}

void Spectrum::PutSpinsOnSphere()
{
      string answer;
      double distanceBetweenSpins;
      int counter = 0;

            for (int i=0; i<num_of_spins;i++)
            {
                  SpinCoordinates[i] = createCoordinates();

                  for (int j=0; j<i; j++)
                  {
                        distanceBetweenSpins =
ROOT_SQUARE_SUM(SpinCoordinates[i].X - SpinCoordinates[j].X,

      SpinCoordinates[i].Y - SpinCoordinates[j].Y,

      SpinCoordinates[i].Z - SpinCoordinates[j].Z);

                        if (distanceBetweenSpins <
DISTANCE_SPINS_MINIMUM || distanceBetweenSpins >
DISTANCE_SPINS_MAXIMUM)
                              i--;
                  }
            }
      for (int i=0; i<num_of_spins; i++)
```

```
                  outCoord << SpinCoordinates[i].X << "      " <<
SpinCoordinates[i].Y << "      " << SpinCoordinates[i].Z << endl;
}

void Spectrum::initializeDistanceArray()
{
      int i, j;

      for (i=0; i<num_of_spins; i++)
            for (j=0; j<num_of_spins; j++)
                  distanceArray[i][j] = 0;
}

void Spectrum::initializeOrderArray()
{
      int i,j;

      for (i=0; i<num_of_spins; i++)
            for (j=0; j<num_of_spins; j++)
                  orderArray[i][j] = j;
}

void Spectrum::initializeArray()
{
      int i, j;

      for (i=0; i<num_of_spins; i++)
            for(j=0; j<AXIS; j++)
                  runningArray[i][j] = 0.0;
      for (i=0; i<num_of_spins; i++)
            runningArray[i][AXIS/2] = 0.01;
}

void Spectrum::initializeTempRunningArray()
{
      int i, j;

      for (i=0; i<num_of_spins; i++)
            for (j=0; j<AXIS; j++)
                  tempRunningArray[i][j] = 0.0;
}

void Spectrum::initializeOutput()
{
      int i, j;

      for (i=0; i<num_of_spins; i++)
            for (j=0; j<AXIS; j++)
                  output[i][j] = 0.0;
}

Coordinates Spectrum::createField(int theta, int phi)
{
      Coordinates temporary;
```

```
	temporary.X = CIRCLE_RADIUS * sin(theta) * cos(phi);
	temporary.Y = CIRCLE_RADIUS * sin(theta) * sin(phi);
	temporary.Z = CIRCLE_RADIUS * cos(theta);

	return temporary;
}

float Spectrum::CalculateSplitValue(Coordinates firstSpin, Coordinates
secondSpin, Coordinates Field)
{
	float FieldRadius;
	float SpinsRadius;
	Coordinates Spins;
	float cosine;
	float splitValue;

	Spins.X = secondSpin.X - firstSpin.X;
	Spins.Y = secondSpin.Y - firstSpin.Y;
	Spins.Z = secondSpin.Z - firstSpin.Z;

	SpinsRadius = ROOT_SQUARE_SUM(Spins.X, Spins.Y, Spins.Z);
	FieldRadius = ROOT_SQUARE_SUM(Field.X, Field.Y, Field.Z);

	cosine = ((Spins.X * Field.X) + (Spins.Y * Field.Y) + (Spins.Z *
Field.Z)) / (FieldRadius * SpinsRadius);

	splitValue = 52041 * (3 * square(cosine) - 1) /
(cube(SpinsRadius) * 2.80494488);

	return splitValue;
}

void Spectrum::Splitting(float split, int spinNumber)
{
	int move;
	int i, j;
	int minus = 0;
	int plus = 0;

	split *= 10;                        //10 is 0.1G step and 100 is
0.01 G
	split /= 2;
	split += 0.5;
	move = static_cast<int>(split);

	if (move != 0)
	{
		initializeTempRunningArray();

		for (i=0; i<AXIS; i++)
		{
			if(runningArray[spinNumber][i] != 0)
			{
				minus = i - move;
				plus = i + move;
```

```
                        if (plus > AXIS || minus < 0)
                        {
                              outError << "plus: " << plus << endl;
                              outError << "minus: " << minus << endl;
                              outError << "split: " << split << endl;
                              outError << "move: " << move << endl;
                              outError << "i: " << i << endl;
                              outError << "split problem" << endl;
                              outError << "Need to increase the size of
the array" << endl;
                              cout << "error" << endl;
                        }
                        else
                        {
                              tempRunningArray[spinNumber][minus] +=
runningArray[spinNumber][i] / 2;
                              tempRunningArray[spinNumber][plus] +=
runningArray[spinNumber][i] / 2;
                        }
                  }
            }
            for (j=0; j<AXIS; j++)
                        runningArray[spinNumber][j] =
tempRunningArray[spinNumber][j];
      }
}

void Spectrum::RunProgram()
{
      int i,j;
      int configuration, theta, phi;
      Coordinates Field;
      float split;
      int first, second;
      int SpinNumberTemp;
      int initialTime, finalTime;


      initialTime = clock();

      initializeOutput();

      for (configuration=0; configuration<CONFIGURATIONS;
configuration++)
      {
            PutSpinsOnSphere();

            for (theta=0; theta<180; theta += INCREMENT)
            {
                  for (phi=0; phi<360*sin(theta); phi += INCREMENT)
                  {
                        initializeArray();

                        Field = createField(theta, phi);
```

```
                        split = CalculateSplitValue(SpinCoordinates[0],
SpinCoordinates[1], Field);
                        Splitting(split, 0);
                        split = CalculateSplitValue(SpinCoordinates[1],
SpinCoordinates[0], Field);
                        Splitting(split, 1);

                        for (i=0; i<AXIS; i++)
                              output[1][i] += runningArray[0][i] +
runningArray[1][i];

                        for (SpinNumberTemp = 2;
SpinNumberTemp<num_of_spins; SpinNumberTemp++)
                        {
                              for (first=0; first<=SpinNumberTemp;
first++)
                              {
                                    if (first == SpinNumberTemp)
                                    {
                                          for (second= 0; second<first;
second++)
                                          {
                                                split =
CalculateSplitValue(SpinCoordinates[first], SpinCoordinates[second],
Field);
                                                Splitting(split,
first);
                                          }
                                    }
                                    else
                                    {
                                          split =
CalculateSplitValue(SpinCoordinates[first],
SpinCoordinates[SpinNumberTemp], Field);
                                          Splitting(split, first);
                                    }
                                    for (i=0; i<=SpinNumberTemp; i++)
                                          for(j=0; j<AXIS; j++)

      output[SpinNumberTemp][j] += runningArray[i][j];
                              }
                        }
                  }
            }
      }

      finalTime = clock();
      getTime(initialTime, finalTime);
}

void Spectrum::outputData()
{
      int i, j;
```

```
	for (i=0; i < AXIS; i++)
	{
		outfile << increments;
		increments += INCREMENTS;
		for(j=1; j<num_of_spins; j++)
			outfile << "	" << output[j][i];
		outfile << endl;
	}
}

void Spectrum::getTime(int initialTime, int finalTime)
{
	int time;
	int seconds, minutes, hours, days;

	time = (finalTime - initialTime) / CLOCKS_PER_SEC;

	days = time /86400;
	time %= 86400;
	hours = time / 3600;
	time %= 3600;
	minutes = time / 60;
	time %= 60;
	seconds = time;

	outTime << "spin# " << num_of_spins << ", configurations: " <<
CONFIGURATIONS << "and increments of: " << INCREMENT <<  endl;
	outTime << "the program ran in: " << days << " days, " << hours
<< " hours, " << minutes << " minutes, " << seconds << " seconds" <<
endl;
	outTime << "initial time was: " <<  initialTime << " and final
time was: " << finalTime << endl;
}

int main()
{
	Spectrum test;

	test.RunProgram();
	test.outputData();

	return 0;
}
```

Appendix P: AVOID.CPP

A C++ program that calculates an lbf for a avoidance model using the repeated splittings method, avoid.cpp:

```
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <string>
#include <ctime>
#include <time.h>

using namespace std;

ofstream outfile ("outfileData.txt");

ofstream outCoord ("outfileCoord.txt");

ofstream outCoordClust ("outfileCoordClust.txt");

ofstream outCoordAvoid ("outfileCoordAvoid.txt");

ofstream outError ("outfileError.txt");

ofstream outTime ("outfileTime.txt");

ofstream outDistance ("outfileDist.txt");

ifstream inputfile ("inputfileData.txt");

const int num_of_spins = 55;
const int MAX = 1000;
const int MAXangle = 360;
const int CIRCLE_RADIUS = 35;
const int AXIS = 1000;
const int CONFIGURATIONS = 50;
const double INCREMENTS = 0.1;
const float DISTANCE_SPINS_MINIMUM = 10.0;
const float DISTANCE_SPINS_MAXIMUM = 70.0;
double increments = 0.0;
int INCREMENT = 6;

/*********************************************************************
********************
This function will generate random numbers (I got it from the web)
*********************************************************************
********************/

void init_mm( );
int number_range( int from, int to );
int number_mm( void );
```

```
static  int     rgiState[2+55]; // leave this alone

float random(int TO)
{
      init_mm(); //seed the number generator
      int random = number_range( 0, TO);
      return random;
}

int number_mm( void )
{
      int *piState;
      int iState1;
      int iState2;
      int iRand;
      piState = &rgiState[2];
      iState1 = piState[-2];
      iState2 = piState[-1];
      iRand = ( piState[iState1] + piState[iState2] )
                  & ( ( 1 << 30 ) - 1 );
      piState[iState1]    = iRand;
      if ( ++iState1 == 55 )
            iState1 = 0;
      if ( ++iState2 == 55 )
            iState2 = 0;
      piState[-2] = iState1;
      piState[-1] = iState2;
      return iRand >> 6;
}

/*
* Generate a random number.
*/
int number_range( int from, int to )
{
      int power;
      int number;
      if ( ( to = to - from + 1 ) <= 1 )
            return from;
      for ( power = 2; power < to; power <<= 1 )
            ;
      while ( ( number = number_mm( ) & ( power - 1 ) ) >= to )
            ;
      return from + number;
}

/*
* this is the Mitchell-Moore algorithm from Knuth Volume II.
*/
void init_mm( )
{
      int *piState;
      int iState;
      piState     = &rgiState[2];
      piState[-2] = 55 - 55;
```

```
      piState[-1] = 55 - 24;
      piState[0]  = ( (int) time( NULL ) ) & ( ( 1 << 30 ) - 1 );
      piState[1]  = 1;
      for ( iState = 2; iState < 55; iState++ )
      {
            piState[iState] = ( piState[iState-1] + piState[iState-2] )
            & ( ( 1 << 30 ) - 1 );
      }
      return;
}
/*********************************************************************
********************/

inline float square(float x)
{
      return x*x;
}

inline float cube(float x)
{
      return x*x*x;
}

inline float ROOT_SQUARE_SUM(float x, float y, float z)
{
      return sqrt(square(x) + square(y) + square(z));
}

inline void swap(int x, int y)
{
      int temp;

      temp = x;
      x = y;
      y = temp;
}

struct Coordinates
{
      float X;
      float Y;
      float Z;
};

Coordinates SpinCoordinates[num_of_spins];

double tempRunningArray[num_of_spins][AXIS];

double output[num_of_spins][AXIS];

float distanceArray[num_of_spins][num_of_spins];

int orderArray[num_of_spins];

class Dendrimer
```

```
{
private:
      double runningArray[num_of_spins][AXIS];
      Coordinates coordinates;
public:
      Coordinates createCoordinates();
      void PutSpinsOnSphere();
      void initializeArray();
      Coordinates createField(int, int);
      float CalculateSplitValue(Coordinates, Coordinates, Coordinates);
      void Splitting(float, int);
      void RunProgram();
      void initializeOrderArray();
      void initializeDistanceArray();
      void initializeTempRunningArray();
      void initializeOutput();
      void outputData();
      void getTime(int, int);
};

Coordinates Dendrimer::createCoordinates()
{
      Coordinates temporary;
      float theta, phi;

      temporary.X = static_cast<float>(random(MAXangle));
      temporary.Z = ((static_cast<float>(random(MAX*CIRCLE_RADIUS*2)))
/ MAX) - CIRCLE_RADIUS;

      theta = asin(temporary.Z /CIRCLE_RADIUS);
      phi = temporary.X;

      temporary.X = CIRCLE_RADIUS * cos(theta) * cos(phi);
      temporary.Y = CIRCLE_RADIUS * cos(theta) * sin(phi);

      return temporary;
}

void Dendrimer::PutSpinsOnSphere()
{
      double distanceBetweenSpins;
      Coordinates temp[num_of_spins];
      float minimum2;
      bool swap;

      initializeOrderArray();

      for (int i=0; i<num_of_spins;i++)
      {
            SpinCoordinates[i] = createCoordinates();

            for (int j=0; j<i; j++)
            {
                  distanceBetweenSpins =
ROOT_SQUARE_SUM(SpinCoordinates[i].X - SpinCoordinates[j].X,
```

```
      SpinCoordinates[i].Y - SpinCoordinates[j].Y,

      SpinCoordinates[i].Z - SpinCoordinates[j].Z);

                  if (distanceBetweenSpins < DISTANCE_SPINS_MINIMUM ||
distanceBetweenSpins > DISTANCE_SPINS_MAXIMUM)
                        i--;
            }
      }
      for (int i=0; i<num_of_spins; i++)
                        outCoord << SpinCoordinates[i].X << "     " <<
SpinCoordinates[i].Y << "     "
                                    << SpinCoordinates[i].Z << endl;

      for (int i=0; i<num_of_spins;i++)
      {
            for (int j=0; j<num_of_spins; j++)
            {
                  distanceArray[i][j] =
ROOT_SQUARE_SUM(SpinCoordinates[i].X - SpinCoordinates[j].X,

      SpinCoordinates[i].Y - SpinCoordinates[j].Y,

      SpinCoordinates[i].Z - SpinCoordinates[j].Z);
                  outDistance << distanceArray[i][j] << "   ";
            }
            outDistance << endl;
      }

      minimum2 = DISTANCE_SPINS_MAXIMUM;
      swap = true;
      int l;
      int j;
      int k;

      for (l=1; l<num_of_spins; l++)
      {
            for (j=l+1; j<num_of_spins; j++)
            {
                  swap = true;
                  for (k=0; k<l; k++)
                  {
                        if (distanceArray[orderArray[k]][orderArray[j]]
< minimum2)
                              swap = false;
                  }
                  if (swap == true)
                  {
                        int d = orderArray[j];
                        orderArray[j] = orderArray[l];
                        orderArray[l] = d;
                        j = num_of_spins;
                  }
            }
```

```
            if (swap == false)
            {
                  swap = true;
                  l--;
                  minimum2--;
            }
      }

      for (int i=0; i<num_of_spins; i++)
            temp[i] = SpinCoordinates[orderArray[i]];

      for (int i=0; i<num_of_spins; i++)
      {
            SpinCoordinates[i] = temp[i];
            outCoordAvoid << SpinCoordinates[i].X << "      " <<
SpinCoordinates[i].Y << "     "
                                << SpinCoordinates[i].Z << endl;
      }
}

void Dendrimer::initializeOrderArray()
{
      int i;

      for (i=0; i<num_of_spins; i++)
            orderArray[i] = i;
}

void Dendrimer::initializeDistanceArray()
{
      int i, j;

      for (i=0; i<num_of_spins; i++)
            for (j=0; j<num_of_spins; j++)
                  distanceArray[i][j] = 0;
}

void Dendrimer::initializeArray()
{
      int i, j;

      for (i=0; i<num_of_spins; i++)
            for(j=0; j<AXIS; j++)
                  runningArray[i][j] = 0.0;
      for (i=0; i<num_of_spins; i++)
            runningArray[i][AXIS/2] = 0.01;
}

void Dendrimer::initializeTempRunningArray()
{
      int i, j;

      for (i=0; i<num_of_spins; i++)
            for (j=0; j<AXIS; j++)
```

```
                  tempRunningArray[i][j] = 0.0;
}

void Dendrimer::initializeOutput()
{
      int i, j;

      for (i=0; i<num_of_spins; i++)
            for (j=0; j<AXIS; j++)
                  output[i][j] = 0.0;
}

Coordinates Dendrimer::createField(int theta, int phi)
{
      Coordinates temporary;

      temporary.X = CIRCLE_RADIUS * sin(theta) * cos(phi);
      temporary.Y = CIRCLE_RADIUS * sin(theta) * sin(phi);
      temporary.Z = CIRCLE_RADIUS * cos(theta);

      return temporary;
}

float Dendrimer::CalculateSplitValue(Coordinates firstSpin, Coordinates
secondSpin, Coordinates Field)
{
      float FieldRadius;
      float SpinsRadius;
      Coordinates Spins;
      float cosine;
      float splitValue;

      Spins.X = secondSpin.X - firstSpin.X;
      Spins.Y = secondSpin.Y - firstSpin.Y;
      Spins.Z = secondSpin.Z - firstSpin.Z;

      SpinsRadius = ROOT_SQUARE_SUM(Spins.X, Spins.Y, Spins.Z);
      FieldRadius = ROOT_SQUARE_SUM(Field.X, Field.Y, Field.Z);

      cosine = ((Spins.X * Field.X) + (Spins.Y * Field.Y)
                  + (Spins.Z * Field.Z)) / (FieldRadius * SpinsRadius);

      splitValue = 52041 * (3 * square(cosine) - 1) /
(cube(SpinsRadius) * 2.80494488);

      return splitValue;
}

void Dendrimer::Splitting(float split, int spinNumber)
{
      int move;
      int i, j;
      int minus = 0;
      int plus = 0;
```

```
	split *= 10;                        //10 is 0.1G step and 100 is
0.01 G
	split /= 2;
	split += 0.5;
	move = static_cast<int>(split);

	if (move != 0)
	{
		initializeTempRunningArray();

		for (i=0; i<AXIS; i++)
		{
			if(runningArray[spinNumber][i] != 0)
			{
				minus = i - move;
				plus = i + move;

				if (plus > AXIS || minus < 0)
				{
					outError << "plus: " << plus << endl;
					outError << "minus: " << minus << endl;
					outError << "split: " << split << endl;
					outError << "move: " << move << endl;
					outError << "i: " << i << endl;
					outError << "split problem" << endl;
					outError << "Need to increase the size of
the array" << endl;
					cout << "error" << endl;
				}
				else
				{
					tempRunningArray[spinNumber][minus] +=
runningArray[spinNumber][i] / 2;
					tempRunningArray[spinNumber][plus] +=
runningArray[spinNumber][i] / 2;
				}
			}
		}
		for (j=0; j<AXIS; j++)
				runningArray[spinNumber][j] =
tempRunningArray[spinNumber][j];
	}
}

void Dendrimer::RunProgram()
{
	int i,j;
	int configuration, theta, phi;
	Coordinates Field;
	float split;
	int first, second;
	int SpinNumberTemp;
	int initialTime, finalTime;

	initialTime = clock();
```

```
      initializeOutput();

      for (configuration=0; configuration<CONFIGURATIONS;
configuration++)
      {
            PutSpinsOnSphere();

            for (theta=0; theta<180; theta += INCREMENT)
            {
                  for (phi=0; phi<360*sin(theta); phi += INCREMENT)
                  {
                        initializeArray();

                        Field = createField(theta, phi);

                        split = CalculateSplitValue(SpinCoordinates[0],
SpinCoordinates[1], Field);
                        Splitting(split, 0);
                        split = CalculateSplitValue(SpinCoordinates[1],
SpinCoordinates[0], Field);
                        Splitting(split, 1);

                        for (i=0; i<AXIS; i++)
                              output[1][i] += runningArray[0][i] +
runningArray[1][i];

                        for (SpinNumberTemp = 2;
SpinNumberTemp<num_of_spins; SpinNumberTemp++)
                        {
                              for (first=0; first<=SpinNumberTemp;
first++)
                              {
                                    if (first == SpinNumberTemp)
                                    {
                                          for (second= 0; second<first;
second++)
                                          {
                                                split =
CalculateSplitValue(SpinCoordinates[first],

                  SpinCoordinates[second], Field);
                                                Splitting(split,
first);
                                          }
                                    }
                                    else
                                    {
                                          split =
CalculateSplitValue(SpinCoordinates[first],

            SpinCoordinates[SpinNumberTemp], Field);
                                          Splitting(split, first);
                                    }
                                    for (i=0; i<=SpinNumberTemp; i++)
```

```
                                        for(j=0; j<AXIS; j++)

      output[SpinNumberTemp][j] += runningArray[i][j];
                                }
                        }
                }
        }
      }

      finalTime = clock();
      getTime(initialTime, finalTime);
}

void Dendrimer::outputData()
{
      int i, j;

      for (i=0; i < AXIS; i++)
      {
            outfile << increments;
            increments += INCREMENTS;
            for(j=1; j<num_of_spins; j++)
                  outfile << "      " << output[j][i];
            outfile << endl;
      }
}

void Dendrimer::getTime(int initialTime, int finalTime)
{
      int time;
      int seconds, minutes, hours, days;

      time = (finalTime - initialTime) / CLOCKS_PER_SEC;

      days = time /86400;
      time %= 86400;
      hours = time / 3600;
      time %= 3600;
      minutes = time / 60;
      time %= 60;
      seconds = time;

      outTime << "spin# " << num_of_spins << ", configurations: " <<
CONFIGURATIONS
                  << " increments of: " << INCREMENT << endl;
      outTime << "the program ran in: " << days << " days, " << hours
<< " hours, "
                  << minutes << " minutes, " << seconds << " seconds"
<< endl;
}

int main()
{
      Dendrimer dendrimer;
```

```
        dendrimer.RunProgram();
        dendrimer.outputData();

        return 0;
}
```

Appendix Q: CLUSTER.CPP

A C++ program that calculates an lbf for a cluster model using the repeated splittings method, cluster.cpp:

```
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <string>
#include <ctime>
#include <time.h>

using namespace std;

ofstream outfile ("outfileData.txt");

ofstream outCoord ("outfileCoord.txt");

ofstream outCoordClust ("outfileCoordClust.txt");

ofstream outError ("outfileError.txt");

ofstream outTime ("outfileTime.txt");

ofstream outDistance ("outfileDist.txt");

const int num_of_spins = 15;
const int MAX = 1000;
const int MAXangle = 360;
const int CIRCLE_RADIUS = 35;
const int AXIS = 1000;
const int CONFIGURATIONS = 80;
const double INCREMENTS = 0.1;
const float DISTANCE_SPINS_MINIMUM = 10.0;
const float DISTANCE_SPINS_MAXIMUM = 70.0;
double increments = 0.0;
int INCREMENT = 6;


/*********************************************************************
*****************************************************
This function will generate random numbers (I got it from the web)
*********************************************************************
*****************************************************/

void init_mm( );
int number_range( int from, int to );
int number_mm( void );
static  int     rgiState[2+55]; // leave this alone

float random(int TO)
{
```

```
        init_mm(); //seed the number generator
        int random = number_range( 0, TO);
        return random;
}

int number_mm( void )
{
        int *piState;
        int iState1;
        int iState2;
        int iRand;
        piState = &rgiState[2];
        iState1 = piState[-2];
        iState2 = piState[-1];
        iRand = ( piState[iState1] + piState[iState2] )
                    & ( ( 1 << 30 ) - 1 );
        piState[iState1]    = iRand;
        if ( ++iState1 == 55 )
              iState1 = 0;
        if ( ++iState2 == 55 )
              iState2 = 0;
        piState[-2] = iState1;
        piState[-1] = iState2;
        return iRand >> 6;
}

/*
* Generate a random number.
*/
int number_range( int from, int to )
{
        int power;
        int number;
        if ( ( to = to - from + 1 ) <= 1 )
              return from;
        for ( power = 2; power < to; power <<= 1 )
              ;
        while ( ( number = number_mm( ) & ( power - 1 ) ) >= to )
              ;
        return from + number;
}

/*
* this is the Mitchell-Moore algorithm from Knuth Volume II.
*/
void init_mm( )
{
        int *piState;
        int iState;
        piState     = &rgiState[2];
        piState[-2] = 55 - 55;
        piState[-1] = 55 - 24;
        piState[0]  = ( (int) time( NULL ) ) & ( ( 1 << 30 ) - 1 );
        piState[1]  = 1;
        for ( iState = 2; iState < 55; iState++ )
```

```
	{
		piState[iState] = ( piState[iState-1] + piState[iState-2] )
		& ( ( 1 << 30 ) - 1 );
	}
	return;
}
/*********************************************************************
*****************************************************/

inline float square(float x)
{
	return x*x;
}

inline float cube(float x)
{
	return x*x*x;
}

inline float ROOT_SQUARE_SUM(float x, float y, float z)
{
	return sqrt(square(x) + square(y) + square(z));
}

inline void swap(int x, int y)
{
	int temp;

	temp = x;
	x = y;
	y = temp;
}

struct Coordinates
{
	float X;
	float Y;
	float Z;
};

Coordinates SpinCoordinates[num_of_spins];

double tempRunningArray[num_of_spins][AXIS];

double output[num_of_spins][AXIS];

float distanceArray[num_of_spins][num_of_spins];

int orderArray[num_of_spins][num_of_spins];

class Spectrum
{
private:
	double runningArray[num_of_spins][AXIS];
	Coordinates coordinates;
```

```
public:
      Coordinates createCoordinates();
      void PutSpinsOnSphere();
      void initializeArray();
      Coordinates createField(int, int);
      float CalculateSplitValue(Coordinates, Coordinates, Coordinates);
      void Splitting(float, int);
      void RunProgram();
      void initializeDistanceArray();
      void initializeOrderArray();
      void initializeTempRunningArray();
      void initializeOutput();
      void outputData();
      void getTime(int, int);
};

Coordinates Spectrum::createCoordinates()
{
      Coordinates temporary;
      float theta, phi;

      temporary.X = static_cast<float>(random(MAXangle));
      temporary.Z = ((static_cast<float>(random(MAX*CIRCLE_RADIUS*2)))
/ MAX) - CIRCLE_RADIUS;

      theta = asin(temporary.Z /CIRCLE_RADIUS);
      phi = temporary.X;

      temporary.X = CIRCLE_RADIUS * cos(theta) * cos(phi);
      temporary.Y = CIRCLE_RADIUS * cos(theta) * sin(phi);

      return temporary;
}

void Spectrum::PutSpinsOnSphere()
{
      string answer;
      double distanceBetweenSpins;
      float minimum, maximum;
      Coordinates temp[num_of_spins];

      for (int i=0; i<num_of_spins;i++)
      {
            SpinCoordinates[i] = createCoordinates();

            for (int j=0; j<i; j++)
            {
                  distanceBetweenSpins =
ROOT_SQUARE_SUM(SpinCoordinates[i].X - SpinCoordinates[j].X,

      SpinCoordinates[i].Y - SpinCoordinates[j].Y,

      SpinCoordinates[i].Z - SpinCoordinates[j].Z);
```

```
                  if (distanceBetweenSpins < DISTANCE_SPINS_MINIMUM ||
distanceBetweenSpins > DISTANCE_SPINS_MAXIMUM)
                        i--;
            }
      }

      for (int i=0; i<num_of_spins; i++)
                  outCoord << SpinCoordinates[i].X << "     " <<
SpinCoordinates[i].Y << "     " << SpinCoordinates[i].Z << endl;

      for (int i=0; i<num_of_spins;i++)
      {
            for (int j=0; j<num_of_spins; j++)
            {
                  distanceArray[i][j] =
ROOT_SQUARE_SUM(SpinCoordinates[i].X - SpinCoordinates[j].X,

      SpinCoordinates[i].Y - SpinCoordinates[j].Y,

      SpinCoordinates[i].Z - SpinCoordinates[j].Z);
                  outDistance << distanceArray[i][j] << "   ";
            }
            outDistance << endl;
      }

      for (int i=0; i<num_of_spins; i++)
      {
            orderArray[0][0] = 0;
            minimum = 0;
            maximum = DISTANCE_SPINS_MAXIMUM;

            for (int scan=1; scan<num_of_spins; scan++)
            {
                  for (int spin=0; spin<num_of_spins; spin++)
                  {
                        if (distanceArray[0][spin] > minimum &&
distanceArray[0][spin] < maximum)
                        {
                              maximum = distanceArray[0][spin];
                              orderArray[0][scan] = spin;
                        }
                  }

                  minimum = maximum;
                  maximum = DISTANCE_SPINS_MAXIMUM;
            }
      }

      for (int i=0; i<num_of_spins; i++)
            temp[i] = SpinCoordinates[orderArray[0][i]];

      for (int i=0; i<num_of_spins; i++)
      {
            SpinCoordinates[i] = temp[i];
```

```
            outCoordClust << SpinCoordinates[i].X << "      " <<
SpinCoordinates[i].Y << "     " << SpinCoordinates[i].Z << endl;
      }
}

void Spectrum::initializeDistanceArray()
{
      int i, j;

      for (i=0; i<num_of_spins; i++)
            for (j=0; j<num_of_spins; j++)
                  distanceArray[i][j] = 0;
}

void Spectrum::initializeOrderArray()
{
      int i,j;

      for (i=0; i<num_of_spins; i++)
            for (j=0; j<num_of_spins; j++)
                  orderArray[i][j] = j;
}

void Spectrum::initializeArray()
{
      int i, j;

      for (i=0; i<num_of_spins; i++)
            for(j=0; j<AXIS; j++)
                  runningArray[i][j] = 0.0;
      for (i=0; i<num_of_spins; i++)
            runningArray[i][AXIS/2] = 0.01;
}

void Spectrum::initializeTempRunningArray()
{
      int i, j;

      for (i=0; i<num_of_spins; i++)
            for (j=0; j<AXIS; j++)
                  tempRunningArray[i][j] = 0.0;
}

void Spectrum::initializeOutput()
{
      int i, j;

      for (i=0; i<num_of_spins; i++)
            for (j=0; j<AXIS; j++)
                  output[i][j] = 0.0;
}

Coordinates Spectrum::createField(int theta, int phi)
{
      Coordinates temporary;
```

```
	temporary.X = CIRCLE_RADIUS * sin(theta) * cos(phi);
	temporary.Y = CIRCLE_RADIUS * sin(theta) * sin(phi);
	temporary.Z = CIRCLE_RADIUS * cos(theta);

	return temporary;
}

float Spectrum::CalculateSplitValue(Coordinates firstSpin, Coordinates
secondSpin, Coordinates Field)
{
	float FieldRadius;
	float SpinsRadius;
	Coordinates Spins;
	float cosine;
	float splitValue;

	Spins.X = secondSpin.X - firstSpin.X;
	Spins.Y = secondSpin.Y - firstSpin.Y;
	Spins.Z = secondSpin.Z - firstSpin.Z;

	SpinsRadius = ROOT_SQUARE_SUM(Spins.X, Spins.Y, Spins.Z);
	FieldRadius = ROOT_SQUARE_SUM(Field.X, Field.Y, Field.Z);

	cosine = ((Spins.X * Field.X) + (Spins.Y * Field.Y) + (Spins.Z *
Field.Z)) / (FieldRadius * SpinsRadius);

	splitValue = 52041 * (3 * square(cosine) - 1) /
(cube(SpinsRadius) * 2.80494488);

	return splitValue;
}

void Spectrum::Splitting(float split, int spinNumber)
{
	int move;
	int i, j;
	int minus = 0;
	int plus = 0;

	split *= 10;                        //10 is 0.1G step and 100 is
0.01 G
	split /= 2;
	split += 0.5;
	move = static_cast<int>(split);

	if (move != 0)
	{
		initializeTempRunningArray();

		for (i=0; i<AXIS; i++)
		{
			if(runningArray[spinNumber][i] != 0)
			{
				minus = i - move;
```

```
                        plus = i + move;

                        if (plus > AXIS || minus < 0)
                        {
                              outError << "plus: " << plus << endl;
                              outError << "minus: " << minus << endl;
                              outError << "split: " << split << endl;
                              outError << "move: " << move << endl;
                              outError << "i: " << i << endl;
                              outError << "split problem" << endl;
                              outError << "Need to increase the size of
the array" << endl;
                              cout << "error" << endl;
                        }
                        else
                        {
                              tempRunningArray[spinNumber][minus] +=
runningArray[spinNumber][i] / 2;
                              tempRunningArray[spinNumber][plus] +=
runningArray[spinNumber][i] / 2;
                        }
                  }
            }
            for (j=0; j<AXIS; j++)
                        runningArray[spinNumber][j] =
tempRunningArray[spinNumber][j];
      }
}

void Spectrum::RunProgram()
{
      int i,j;
      int configuration, theta, phi;
      Coordinates Field;
      float split;
      int first, second;
      int SpinNumberTemp;
      int initialTime, finalTime;


      initialTime = clock();

      initializeOutput();

      for (configuration=0; configuration<CONFIGURATIONS;
configuration++)
      {
            PutSpinsOnSphere();

            for (theta=0; theta<180; theta += INCREMENT)
            {
                  for (phi=0; phi<360*sin(theta); phi += INCREMENT)
                  {
                        initializeArray();
```

```
                        Field = createField(0, 0);

                        split = CalculateSplitValue(SpinCoordinates[0],
SpinCoordinates[1], Field);
                        Splitting(split, 0);
                        split = CalculateSplitValue(SpinCoordinates[1],
SpinCoordinates[0], Field);
                        Splitting(split, 1);

                        for (i=0; i<AXIS; i++)
                              output[1][i] += runningArray[0][i] +
runningArray[1][i];

                        for (SpinNumberTemp = 2;
SpinNumberTemp<num_of_spins; SpinNumberTemp++)
                        {
                              for (first=0; first<=SpinNumberTemp;
first++)
                              {
                                    if (first == SpinNumberTemp)
                                    {
                                          for (second= 0; second<first;
second++)
                                          {
                                                split =
CalculateSplitValue(SpinCoordinates[first], SpinCoordinates[second],
Field);
                                                Splitting(split,
first);
                                          }
                                    }
                                    else
                                    {
                                          split =
CalculateSplitValue(SpinCoordinates[first],
SpinCoordinates[SpinNumberTemp], Field);
                                          Splitting(split, first);
                                    }
                                    for (i=0; i<=SpinNumberTemp; i++)
                                          for(j=0; j<AXIS; j++)

      output[SpinNumberTemp][j] += runningArray[i][j];
                              }
                        }
                  }
            }
      }

      finalTime = clock();
      getTime(initialTime, finalTime);
}

void Spectrum::outputData()
{
      int i, j;
```

```
	for (i=0; i < AXIS; i++)
	{
		outfile << increments;
		increments += INCREMENTS;
		for(j=1; j<num_of_spins; j++)
			outfile << "	" << output[j][i];
		outfile << endl;
	}
}

void Spectrum::getTime(int initialTime, int finalTime)
{
	int time;
	int seconds, minutes, hours, days;

	time = (finalTime - initialTime) / CLOCKS_PER_SEC;

	days = time /86400;
	time %= 86400;
	hours = time / 3600;
	time %= 3600;
	minutes = time / 60;
	time %= 60;
	seconds = time;

	outTime << "spin# " << num_of_spins << ", configurations: " <<
CONFIGURATIONS << " increments of: " << INCREMENT << endl;
	outTime << "the program ran in: " << days << " days, " << hours
<< " hours, " << minutes << " minutes, "
			<< seconds << " seconds" << endl;
}

int main()
{
	Spectrum test;

	test.RunProgram();
	test.outputData();

	return 0;
}
```

Appendix R: RMS70-10000.C

A program that calculated the number of 70 spheres that will tightly pack around an inner sphere, rms70-10000.c:

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

/*
double erand48 (unsigned short xsubi[3]);
*/


/* using malloc(3x) */
#include <sys/types.h>
#include <malloc.h>

#include <time.h>
 /*time usage:
 time_t time(time_t *tloc); */



int   debug = 0;

extern  double    str2num(char *temp);

#define maxlinelength 256
#define maxlinesinfile 1024

#define MAX 1000
#define pi 3.1415927


/* actual beginning of main program */
void main (args,argstr)

int args;
char *argstr[];

{ /* beginning of main program */

int   p,i,j,k,n,m,nlimit,v;
double      a,b,c;
double      x1, x2, y1, y2, z1, z2;
FILE  *infile, *outfile1;
char  filename[20];
int   nsamples;   /* number of samples user desires */
int   linesfound; /* number of lines found in input text stream */
```

```
char  textbuffer[maxlinesinfile][maxlinelength]; /* buffer of lines in
text */
char  **endptr;
unsigned short xsubi[4];
char  instring[10];
char  of1[256], of2[256];
double      r; /* radius of sphere */
double      min;
int   K[30][10002],     dia,rmin;
double      x[MAX];
double      y[MAX];
double      z[MAX];
double      d[56];
double      theta[56];
double      phi[56];
double      dist;
double      dx,dy;

/* trying a new type  */
time_t      presenttime;
time_t      tloc;

int   zero, one, two;



/* initializing random number generator */
      /* dumb initialization for the moment */
xsubi[0] = (unsigned short) 2;
xsubi[1] = (unsigned short) 1;
xsubi[2] = (unsigned short) 0;
      /* better initialization */
/* presenttime is of type time_t, which is equivalent to "long" */
presenttime = time(&tloc);

/* pointers to xsubi -- we can change these if we don't like something
about
                  the way the random numbers are coming out
                  setting all these to the same integer value will
                  prevent the routine from generating a different
                  series of numbers each time it is called numbers */
zero = 0;
one = 1;
two = 2;


/* randomly picking nsamples input lines from the textbuffer */
v=0;

for (r = 40; r < 155; r = r + 5)
      {
      v=(r-40)/5;
      dia = r * 2;
      /* for (min = 4; min < 10; min = min + 1)
            { */
```

```
min = 70;
rmin = 70;
for (n = 0; n < 10000; n = n + 1)
      { /* beginning of n loop */


      presenttime = time(&tloc);
      j = 0xffff &  presenttime;
      xsubi[zero] = (unsigned short ) j;

      j = 0xffff0000 &  presenttime;
      j = j / 0xffff;
      xsubi[one] = (unsigned short ) j;

      j = (n + (int) xsubi[zero]) * ((int) xsubi[one]);
      xsubi[two] = (unsigned short ) j;

      for (k = 0; k < 56; k = k + 1)
            {
            x[k]=0.0;
            y[k]=0.0;
            z[k]=0.0;
            }
      c = 0;
      i = 0;
      while ((i < 56))
            { /* beginning of first i loop */
            /* erand48 returns a number in the interval
[0.0,1.0] */
            c=c+1;
            i=i+1;
            x[i] = (erand48(xsubi)-.5)*20;
            y[i] = (erand48(xsubi) -.5)*20;
            z[i] = (erand48(xsubi) -.5)*20;
            a = r/(sqrt(x[i]*x[i] + y[i]*y[i] +
z[i]*z[i]));
            x[i] = a * x[i];
            y[i] = a * y[i];
            z[i] = a * z[i];
            k = i - 1;
                  for (p = 1; p < i; p=p+1)
                        {
                        dist =        (x[i]-x[1])*(x[i]-
x[1]);
                        dist = dist + (y[i]-y[1])*(y[i]-
y[1]);
                        dist = dist + (z[i]-z[1])*(z[i]-
z[1]);
                        dist = sqrt(dist);
                        if ( dist < min )
                              {
                              i = k;
                              }
                        if ( c > 100000)
                              {
```

```
                                        i = 56;
                                        }

                                  }
                        } /* end of first i loop */


                        K[v][n]= k;
                        c = 0;


                  } /* end of n loop */

            /* } end of min loop */

      }/* end of r loop */


      for (r = 40; r < 155; r = r + 5)
            {
            dia = r * 2;
            printf("%d\t",dia);
            }
       printf("\n");

      for (j = 0; j < 10000; j = j + 1)
            {

            for (i = 0; i < v+1; i = i + 1)
                  {

                  printf("%d\t",K[i][j]);

                  }
            printf("\n");

            }
} /* end of main program */
```

Appendix S: PDB.AWK

An awk script that extracts x,y,z coordinates for the terminal atom from a pdb file, pdb.awk:

```
Begin {
end = ".coord"
}
{

if ($2=="LYS" &&($3 == "NZ"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="ALA" &&($3 == "CB"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="ILE" &&($3 == "CG2"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="TRP" &&($3 == "NE1"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="THR" &&($3 == "CG2"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="TYR" &&($3 == "OH"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="SER" &&($3 == "OG"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="CYS" &&($3 == "SG"))
```

```
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="GLU" &&($3 == "OE1"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="LEU" &&($3 == "CD2"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="PRO" &&($3 == "CG"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="ASN" &&($3 == "ND2"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="ARG" &&($3 == "NH1"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="GLN" &&($3 == "NE2"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="PHE" &&($3 == "CE2"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="MET" &&($3 == "CG"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="HIS" &&($3 == "ND1"))
      {
```

```
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="ASP" &&($3 == "OD2"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="GLY" &&($3 == "CA"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }

if ($2=="VAL" &&($3 == "CG1"))
      {
      name = $1 $2 $3;
      printf("%s\t%s\t%s\n",$4,$5,$6)> name;
      }
}
end
{
}
```

Appendix T: PROBROAD.C

A C program that calculates an lbf from the coordinate files generated by resendatom.awk, probroad.c:

```
/*
This file

by DAS
Fri Jul 21 15:30:33 MDT 2000
 */


extern int  debug = 0;
extern double str2num(char[99]);

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <time.h>

#define MAX 257
#define LINELENGTH 1024
#define     LIMIT 1000

main (args,argstr)
int args;
char *argstr[];

{ /* beginning of main program */

FILE *infile;
FILE *outfile1, *outfile2;
double      x[MAX],y[MAX],z[MAX];
char  lines[MAX][LINELENGTH];
int   nn[MAX],order[MAX];
int   i,j,k,l, m,n,p, line, max,maxused,swap;
int   jj, ni, nj;
int   a,b,d,v,w;
int   nlimit;
char  *inpline;
char  inf[LINELENGTH];
char  of1[LINELENGTH];
char  of2[LINELENGTH];
double      btemp,field,broad[3999],distance[MAX][MAX], dist,h;
double      sum[3100];
int   row,col;

/* for random number generation */
unsigned short xsubi[3];
int     zero = 0;
int   one = 1;
```

```
int   two = 2;

/* trying a new type  */
time_t  presenttime;
time_t  tloc;



/* initializing random number generator */
presenttime = time(&tloc);
jj = 0xffff &  presenttime;
xsubi[zero] = (unsigned short ) jj;

jj = 0xffff0000 &  presenttime;
jj = jj / 0xffff;
xsubi[one] = (unsigned short ) jj;

jj = ((int) xsubi[zero]) * ((int) xsubi[one]);
xsubi[two] = (unsigned short ) jj;









if ( ( infile = fopen(argstr[1],"r") ) != (FILE *) NULL )
      {/* success */}
else
      {
      fprintf(stderr,"load: file '%s' not found.\n",argstr[1]);
      return(1);
      }

i = 0;
while ( (argstr[1][i] != '.' ) && ( argstr[1][i] != '\0') )
      {
      inf[i] = argstr[1][i];
      i = i + 1;
      }
inf[i] = '\0';
/* printf("output filename root = %s\n",inf); */



line = 1;
while ( !(feof(infile)) )
      { /*beginning of while loop to read file */

      /* reading line of input file  */
      fscanf(infile,"%[^\n]\n",&lines[line][0]);
      lines[line][-1] = '0';
/*    printf("'%s'\n",inpline);  */
```

```
      /* i = fscanf(infile," %f %f %f\n",&x[line],&y[line],&z[line]);
            */

      inpline = &lines[line][0];


      j = findtextfield(inpline,1);
      x[line] = str2num(&inpline[j]) ;

      k = findtextfield(inpline,2);
      y[line] = str2num(&inpline[k]) ;

      l = findtextfield(inpline,3);
      z[line] = str2num(&inpline[l]) ;
      /* printf("number: '%s'\n",&inpline[l]); */


      /*printf("%d\t%f\t%f\t%f\n",line,x[line],y[line],z[line]);*/


      max = line;
      line = line + 1;

      } /*end of while loop to read file */

fclose(infile);


maxused = .3 * max;


/* ================================ adapted  from random2.c */

/* randomly picking nsamples input lines from the textbuffer */

/*
printf("p loop \n");
*/

/* return(1); */
/* ================================ setting sum matrix to 0*/
for (i = 0; i < max ; i = i + 1)
      {
      sum[i] = 0.0;
      }

/* ================================ setting distance matrix to 0*/
for (i = 0; i <= max ; i = i + 1)
      {
      for (j = 0; j <= max ; j = j + 1)
            {
            distance[i][j] = 0.0;
            }
      }
/* ================================ setting order matrix i=i*/
```

```
for (i = 0; i <= max ; i = i + 1)
      {
      order[i] = i;
      }
/* ================================ setting broad matrix = 0*/
for (i = 0; i <= 3999 ; i = i + 1)
      {
      broad[i] = 0.0;
      }

/* ================================ calculating distance matrix */




for (i = 1; i <= max ; i = i + 1)
      {
      for (j = 1; j <= max ; j = j + 1)
            {
            distance[i][j] =          pow(x[i]-x[j],2);
            distance[i][j] = distance[i][j] + pow(y[i]-y[j],2);
            distance[i][j] = distance[i][j] + pow(z[i]-z[j],2);
            distance[i][j] = sqrt(distance[i][j]);

            }

      }

/*for (i = 1; i <= max ; i = i + 1)
      {
      for (j = 1; j <= max ; j = j + 1)
            {
            printf("%e\t",distance[i][j]);

            }
      printf("\n");

      }
*/

/* ================================ calculating r6 matrix */


for (i = 1; i <= max ; i = i + 1)
      {

      for (j = 1; j <= max; j = j + 1)
            {
            if (i!=j)
                  {
                  sum[i] = sum[i] + pow(distance[order[i]][order[j]],-
6);
                  }
            }
      }
```

```
for (i = 1; i <= max ; i = i + 1)
      {
            {
            if (sum[i]> 1e-15)
                  {
                  sum[i] = (sqrt(sum[i])*19525.97);
                  }

            }
      }

/*calculates sum of gaussian broadening functions*/


      for (j = 0; j <= 3999; j = j + 1)
            {
            field = ((double)j)/20.0 - 100.0;

            for (i = 0; i <= max; i = i + 1)
                  {
                  if (sum[i] > 1e-15)
                        {
                        btemp = ( exp ( -0.693 * (field/(.5 * sum[i]))*
(field/(.5 * sum[i]))))/sum[i];

                        broad[j] = broad[j] + btemp;
                        }
                  }
            }




/* this prints broadening */
  for (i = 0; i <= (3999) ; i = i + 1)
      {
      field = ((double)i)/20.0 - 100.0;
      printf("%e\t",field);
      printf("%e\t",broad[i]);
      printf("\n");
      }


} /* end of main program */
```