LBL-37588
UC-401

# Lawrence Berkeley Laboratory

UNIVERSITY OF CALIFORNIA

## CHEMICAL SCIENCES DIVISION

**Excited State Carrier Dynamics in $CdS_xSe_{1-x}$ Semiconductor Alloys as Studied by Ultrafast Fluorescence Spectroscopy**

S.E. Gadd
(Ph.D. Thesis)

August 1995

REFERENCE COPY
Does Not
Circulate
Bldg. 50Ø Library.
LBL-37588
Copy 1

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

## DISCLAIMER

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

LBL-37588

# Excited State Carrier Dynamics in $CdS_xSe_{1-x}$ Semiconductor Alloys as Studied by Ultrafast Fluorescence Spectroscopy

by

Steven Edward Gadd

Ph.D. Thesis

Department of Chemistry
University of California

and

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

August 1995

This work was supported by the Laboratory Directed Research and Development Program of Lawrence Berkeley National Laboratory under the U. S. Department of Energy, Contract No. DE-AC03-76SF00098.

Excited State Carrier Dynamics in $CdS_xSe_{1-x}$ Semiconductor Alloys as Studied by Ultrafast Fluorescence Spectroscopy

© Copyright 1995

by

Steven Edward Gadd

The U.S. Department of Energy has the right to use this dissertation for any purpose whatsoever, including the right to reproduce all or any part thereof.

# Abstract

Excited State Carrier Dynamics in $CdS_xSe_{1-x}$ Semiconductor Alloys as Studied by Ultrafast Fluorescence Spectroscopy

by

Steven Edward Gadd

Doctor of Philosophy in Chemistry

University of California, Berkeley

Professor Charles B. Harris, Chair

This dissertation discusses studies of the electron–hole pair dynamics of $CdS_xSe_{1-x}$ semiconductor alloys for the entire compositional range from $x = 1$ to $x = 0$ as examined by the ultrafast fluorescence techniques of time correlated single photon counting and fluorescence upconversion. Specifically, samples with $x = 1$, .75, .5, .25, and 0 were studied each at a spread of wavelengths about its respective emission maximum which varies according to $\lambda = 718\text{nm} - 210x$ nm. The decays of these samples were found to obey a Kohlrausch distribution, $\exp\left[(t/\tau)^{\beta}\right]$, with the exponent $\beta$ in the range .5–.7 for the alloys. These results are in agreement with those expected for localization due to local potential variations resulting from the random distribution of sulfur and selenium atoms on the element VI A sub-lattice. This localization can be understood in terms of Anderson localization of the holes in states whose energy distribution tails into the forbidden energy band-gap. Because these states have energy dependent lifetimes, the carriers can decay via many parallel channels. This distribution of channels is the ultimate source of the Kohlrausch form of the fluorescence decays.

The lifetimes observed in these experiments depended on the wavelength of excitation. In the time correlated single photon counting experiments where ultraviolet excitation was employed, lifetimes of about 30 ps were observed independent of composition, indicative of a strong domination of the decay by nonradiative mechanisms resulting from surface trapping. By comparison, the decays in the fluorescence upconversion experiments displayed longer decays which varied as a function of composition.

In addition to displaying a Kohlrausch form, the decays of all of the samples also included an exponential rise of 1–2 ps consistent with results for other direct band-gap semiconductors. On the other hand, none of the samples exhibited a signigicant variation in either their decays or their risetimes as a function of wavelength.

# Contents

**List of Figures** v

**List of Tables** vii

**1 Introduction** 1
- 1.1 General . . . 1
- 1.2 Graded Alloys . . . 5

**2 Localization and the Density of States** 9
- 2.1 Introduction . . . 9
- 2.2 Anderson Localization Theory . . . 13
- 2.3 Density of States . . . 19
- 2.4 Determining the Type of Localization . . . 26
- 2.5 Summary . . . 32

**3 Ultrafast Fluorescence Techniques** 34
- 3.1 Time Correlated Single Photon Counting . . . 34
- 3.2 Fluorescence Upconversion . . . 40
  - 3.2.1 Introduction . . . 40
  - 3.2.2 Phase-matching . . . 43
  - 3.2.3 Dispersion . . . 47
  - 3.2.4 Upconversion Systems . . . 49
  - 3.2.5 Summary . . . 54
- 3.3 A Comparison: Upconversion vs. TCSPC . . . 55

**4 Experimental** 58
- 4.1 Samples . . . 58
- 4.2 The Laser System . . . 60
- 4.3 Time Correlated Single Photon Counting . . . 61

4.4 Fluorescence Upconversion . . . . . 64
4.5 Numerical Fitting . . . . . 69

**5 Results and Discussion** **74**
5.1 Results . . . . . 74
5.1.1 Time Correlated Single Photon Counting . . . . . 74
5.1.2 Fluorescence Upconversion . . . . . 79
5.2 Discussion . . . . . 84

**6 Summary** **91**

**Bibliography** **94**

**A Data Collection Program** **104**
A.1 smain.c . . . . . 104
A.2 sdata.c . . . . . 130
A.3 snice.c . . . . . 137
A.4 splot.c . . . . . 160
A.5 ssave.c . . . . . 167
A.6 slook.c . . . . . 171
A.7 sconst.c . . . . . 187

**B Data Fitting Program** **193**
B.1 fitX.c . . . . . 193
B.2 fitaux.c . . . . . 221
B.3 fitgraph.c . . . . . 242
B.4 calc.c . . . . . 254

# List of Figures


1.1 CdS and CdSe band structures . . . . . 4

2.1 Fluctuations in the potential and band edge states . . . . . 11
2.2 Expansion of self-energy in propagator series and locator series . . . . 16
2.3 Spherical potential wells for single and double particle localization . . 24
2.4 Localization type as a function of $-\alpha_n/\alpha_p$ and $m_n/m_p$ . . . . . 31

3.1 Typical time correlated single photon counting setup . . . . . 38
3.2 Schematic of laser pulse acting as a light gate . . . . . 42
3.3 Typical fluorescence upconversion setup with collinear phase matching 52

4.1 Spectra for $CdS_xSe_{1-x}$, $x = 1$, .75, .50, .25, and 0 . . . . . 59
4.2 TCSPC instrument function . . . . . 63
4.3 Fluorescence upconversion instrument function . . . . . 70

5.1 Fluorescence decays of all five samples by TCSPC . . . . . 75
5.2 Fluorescence decays of $CdS_{.50}Se_{.50}$ by TCSPC . . . . . 78
5.3 Fluorescence decays of CdSe by upconversion . . . . . 80
5.4 Fluorescence decays of $CdS_{.25}Se_{.75}$ by upconversion . . . . . 81
5.5 Fluorescence decays of $CdS_{.50}Se_{.50}$ by upconversion . . . . . 82
5.6 Rising edge fit with and without exponential rise . . . . . 85

# List of Tables

4.1 Experimental conditions for upconversion experiments . . . . . 67

5.1 Fitting results of decays measured by TCSPC . . . . . 79
5.2 Fitting results of decays measured by upconversion . . . . . 83

# Acknowledgments

The execution of such work as represented by the present dissertation cannot in all honesty be claimed as the work of a single individual. As John Donne rightly noted, "No man is an island." In this light I am honor bound to acknowledge those who have made this work possible. Unfortunately the list of names of those whom I would like to acknowledge is prohibitively long, and only the most notable will be recognized here.

Firstly I would like to extend to Professor Charles B. Harris the thanks due for his support and guidance. He has allowed me the freedom to proceed in this work and to continue to grow. His encouragement and exemplary leadership have been crucial to this continued development and to the course of life on which I now embark.

Jennifer Hane, who initiated this line of inquiry in the Harris group along with Dr. Michael Prisant, has been my collaborator in the lab for this work. Working with her has been both educational and enjoyable, and my deepest appreciation is extended to her for her contributions. In addition to her aid, much of what I have learned of the laser systems with which I have worked has been due to the ever-helpful and knowledgable assistance of Robert Jordan. My forays into computer programming were aided by the work of Dan Russell, whose data collection program provided the basis for my own collection program. Later work was enhanced by Jason McNeill and Matt Asplund who provided systems support and helpful discussions on computing science in general. I would also like to thank Walter Merry, Dr. Robert Lingle, and especially Dr. David Padowitz for helpful discussions over the years.

I would also be remiss were I not to extend my gratitude to Vijaya Narasimhan, whose oversight of the group finances and purchasing ensured that work could

always continue. Her ability to maneuver through the bureaucracy of purchasing greatly simplified such matters, and her friendly greetings made mornings a little brighter. Also I would like to recognize Lawrence Berkeley Laboratories for financial support of this work through an LDRD grant. Additional support was provided by the National Science Foundation and the Office of Naval Research, the latter through a National Defense Science and Engineering Graduate Fellowship.

Man is a social animal, and in this I am no exception. The Harris group both past and present has been of inestimable importance in fulfilling this aspect of my being. From coffee breaks to baseball games to Thanksgiving dinners, the members of the group have provided many hours of relief from the stresses of graduate school. Most important of these has undoubtedly been Jason McNeill, whose friendship both at work and beyond has made my tenure here much more enjoyable than it would have otherwise been. I would also like to thank all of the members of the many softball teams on which I was a member. We may not have won many games, but we always had a good time. In addition, the GSO should also be recognized as a source of friendship and revelry. I hope its social traditions may continue for many years to come.

Finally, I must thank my parents, Roland and Janice Gadd, for always being supportive and encouraging. From the frequent trips of my youth to musesus and zoos to my yet ongoing discussions of academic topics, my parents have continually assisted my continued intellectual development. I could surely never have attained this level of achievement without their loving support. For these reasons this work is dedicated to them.

# Chapter 1

# Introduction

## 1.1 General

Semiconductors and semiconductor technology are fundamental to modern existence. From their uses in computers and communications technology in the everyday world to their use in the laboratory, semiconductors surround and enrich human life. Ideally one would desire to have complete control over the optical and electronic properties of these substances. While complete control is undoubtedly unattainable, semiconductor alloys do provide an additional means for tailoring such materials. For example, substituting selenium atoms for sulfur atoms in cadmium sulfide allows one to engineer a spread of properties from those of pure cadmium sulfide to those of pure cadmium selenide. An example of the results of such control can be observed in the emission edge, which varies smoothly and monotonically from 2.4 eV for cadmium sulfide to 1.8 eV for cadmium selenide as a greater fraction of the VI A element sites are filled by selenium atoms.

Furthermore, substitutional alloys of semiconductors play an important role

in the continuing evolution of the understanding of the fundamental physics of the solid state. They occupy an interesting position between well-ordered pure crystals and highly disordered amorphous semiconductors. The underlying structure of these alloys is regular and well-ordered; the alloy retains a crystalline structure. However, due to the random substitution of one type of atom for another, there is a loss of symmetry in the local potential even when the crystal structure is relatively unchanged. The question that immediately arises is what effects this loss of symmetry will have on the optical and electronic properties of the material and to what degree these effects will extend. Under what circumstances will the properties be dependent upon the microscopic details of the local potential, and when will the properties represent an averaging over many local variations? An example of such an averaging is the aforementioned emission edge shift. Nevertheless, situations do arise in which a consideration of the microscopic details are necessary, as this dissertation will demonstrate.

Among the many properties of these alloys, the relaxation of a carrier following an excitation as by a laser pulse is of particular interest. Immediately after such an excitation, a rapid thermalization via electron–electron scattering in the conduction band (or hole–hole scattering in the valence band) occurs; then the carriers relax to their respective band edges by means of phonon scattering. Once their respective band minima are reached, the carriers then undergo interband relaxation, recombining by a variety of radiative and nonradiative processes. Because these processes occur on the picosecond and sub-picosecond time scales, ultrafast spectroscopic techniques are critical for their analysis. Fortunately the development of such technology in the past decade has made such studies possible.

The alloy in this study is $CdS_xSe_{1-x}$, which has several features that make it ideal. First of all, the alloy exists over the entire compositional range from pure

cadmium sulfide to pure cadmium selenide, *i.e.* from $x = 1$ to $x = 0$. Secondly, the crystal structure of $CdS_xSe_{1-x}$, the wurtzite, or hexagonal zinc sulfide, structure, is unchanged across the entire regime [1]. Finally, it also has a direct band gap across its entire substitutional range. The band structures for the two binary materials are shown in Fig. 1.1. This retaining of a direct band gap is in distinct contrast to $Al_xGa_{1-x}As$, so prominent for its use in the construction of semiconductor microstructures such as quantum wells and multiple quantum wells; $Al_xGa_{1-x}As$ exists as an indirect band gap semiconductor over the range $.45 \leq x \leq 1$. Comparison of $Al_xGa_{1-x}As$ samples across the entire range would therefore be complicated, since indirect band gap semiconductors relax radiatively via a mechanism which requires the creation or destruction of a phonon. Hence, such semiconductors exhibit reduced quantum efficiency and radiative decay times of several orders of magnitude greater duration [2, 3].

In addition to the above structural and electronic properties, $CdS_xSe_{1-x}$ alloys have several advantageous optical properties as well. The maximum wavelength of their emission has a linear dependence on composition according to

$$\lambda_{max} = 718 - 210x \text{ nm}, \tag{1.1}$$

where $x$ is the mole fraction of sulfur in the element VI A sites. A sample of the fluorescence spectra for the samples studied in the present work will be shown later in Figure 4.1. Furthermore, at room temperature the quantum efficiencies for photoluminescence are on the order of $10^{-3}$ to $10^{-4}$ [5, 6]. As a result of these properties, these alloys have been the object of many photoluminescence studies [7–9]. Finally, being semiconductors, these alloys are also amenable to studies by photoconductivity [10] and electroluminescence [6]. This latter point is of importance because of the

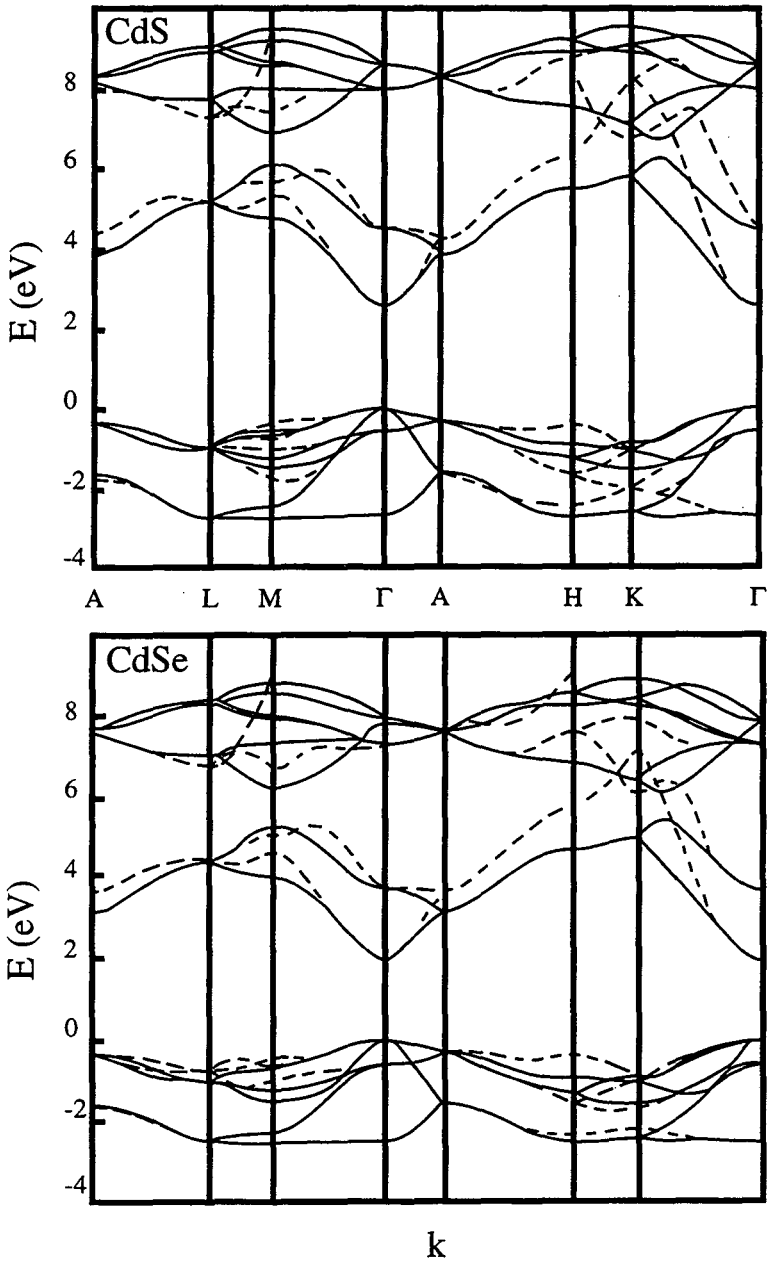


Figure 1.1: Shown above are the band structures for CdS and CdSe in the wurtzite structure as calculated by a semi-empirical tight-binding model (solid line) in comparison to the results of a pseudo-potential calculation (dashed line)[4].

interest in these materials as light emitting diodes and in photoelectrochemical cells for the conversion of optical energy to electronic energy [11, 12].

For the present work, a couple of methods of ultrafast fluorescence spectroscopy have been employed in order to examine the carrier recombination dynamics in these alloys. In particular, the methods of time correlated single photon counting and fluorescence upconversion have been used. These methods offer several advantages over other forms of ultrafast spectroscopy. First of all, fluorescence has a very low background signal. In addition, when studying the samples via back fluorescence, sample thickness is not critical unlike the case for pump–probe absorption experiments. Finally, the use of low power, high repetition rate lasers allows for the collection of single photons and yields excellent signal to noise ratios in a relatively short time.

## 1.2 Graded Alloys

In addition to studying nominally homogeneous alloys of $CdS_xSe_{1-x}$ [6], Carpenter and coworkers carried out both photoluminescence and electroluminescence studies on inhomogeneous samples of $CdS_xSe_{1-x}$ in which the composition, and hence the band gap, varied smoothly as a function of position. Their samples were prepared by vapor-phase diffusion of one of the VI A elements into the bulk cadmium compound of the other. Such samples had previously been demonstrated to exhibit good fluorescence properties [13]. Samples in which sulfur was diffused into cadmium selenide, hereafter referred to as CdSe/S, were studied first [14] and later followed by studies of samples prepared by diffusion of selenium into cadmium sulfide [15], or CdS/Se. In both cases Auger electron spectroscopy in conjunction with Ar ion sputtering indicated a monotonic and nearly linear decrease over a dis-

tance of about 1 mm in the diffused species with a concomitant increase in the other. These results thus support the conclusion that the diffused species was substituting on the lattice rather than entering it interstitially. The CdSe/S samples varied from virtually pure CdS at the surface to pure CdSe in the bulk whereas the CdS/Se samples varied from $CdS_{.2}Se_{.8}$ at the surface to pure CdS in the bulk. This depth profile analysis in combination with the known variation in the emission wavelength with composition (Eq. 1.1) then provides a unique and useful probe of the spatial origins of the luminescence spectra.

Hane and coworkers recognized the novelty of this system for studying carrier transport with both spatial and temporal resolution [16]. Using a time correlated single photon counting spectrometer nearly identical to that described in this work, luminescence decays were measured for samples excited by 315 nm light. At this wavelength and at the intensities used, the light was estimated to be absorbed within about .1 $\mu$m of the surface and created a carrier density of about $10^{17}cm^{-3}$. Carriers thus produced near the surface then diffused into the bulk under the influence of both the concentration gradient and the band gap gradient as described elsewhere [17]. In the CdS/Se samples, the concentration gradient had to drive the carriers up the band gap gradient.

Luminescence decays in CdSe/S exhibited 1/e time constants of 100–400 ps, indicating transport occurred on a time scale comparable to that for the characteristic band to band recombination times in these materials. That these decays include information beyond just the differences in the carrier lifetimes as a function of composition was inferred from the luminescence decays in CdS/Se. If the decays in this material were dominated by the intrinsic lifetimes, then the wavelength dependence of the decays would be expected to be identical to that for the CdSe/S samples. This, however, was not observed. In fact, the decays displayed a depend-

ence on a variety of factors such as the depth at which the recombination occurred and the band gap gradient.

Analysis of the CdSe/S data is the more straightforward of the two, since both the concentration gradient and the band gap gradient are in the same direction. Luminescence was observed over the entire spectral range from 500 nm to 700 nm as expected for transport all the way into the bulk. In addition, the 1/e decay times increased monotonically for wavelengths from 500 nm to 620 nm, corresponding to compositions of $x = 1.0$ to $x = .50$. Based on this data, estimates of the ambipolar mobility were made and found to be in accord with the values for unalloyed CdS and CdSe. However, in the range of 620 nm to 700 nm, the trend in 1/e times reversed itself; the decay times became shorter with increasing wavelength. Furthermore, the spread in the 1/e times was shorter for this wavelength range than for the former.

In comparison to the CdSe/S results, the analysis of the CdS/Se data proved more difficult still, since the band gap increases with depth. Despite this complexity emission was observed at the bulk wavelength of 500 nm. That the carriers were diffusing into the bulk was further supported by gated spectra, in which the spectra were recorded for a narrow time window at a fixed delay time after excitation. More interesting still was the trend of the 1/e decay times as a function of wavelength. As compared to that of the CdSe/S samples, the trend was completely reversed. For the wavelength range from 700 nm to 620 nm, the 1/e times became shorter with decreasing wavelength before turning over and increasing with decreasing wavelength in the range from 620 nm to 500 nm.

Hane and co-workers advanced two possible explanations for this turnover in CdS/Se. First, they noted that after the carriers spread out the band gap gradient could start to drive the carriers back towards the surface. Second, strong self-absorption and re-emission in the portion of the crystal nearer the surface where the

band gap was smallest could account for the initial trend. However, these processes could not have occurred in the CdSe/S samples in which the band gap decreases with depth. Furthermore, the turnaround in the decay time trends in both samples occurred at 620 nm, which corresponds to $x = .5$, presumably the most disordered composition. In order to clarify these results, the behavior of the lifetimes as a function of the composition, and hence of the disorder, is therefore appropriate. In order to divorce the effects of the band gap gradient from those of the composition, however, examination of the homogeneous alloys is required. The study of the effects of disorder on the room temperature luminescence decays of the homogeneous $CdS_xSe_{1-x}$ alloys thus constitutes the thrust of the present dissertation.

The present dissertation is laid out as follows. The next chapter provides a brief overview of the theory of composition fluctuations, including an introduction to Anderson localization theory, and their effect on the density of states. Some specifics as they apply to $CdS_xSe_{1-x}$ will be discussed. The third chapter reviews the details of time correlated single photon counting and fluorescence upconversion, including the important considerations to be made in designing such experiments as well as a comparison of the two techniques. Chapter four provides the experimental details for all of the present work, including the specifics of the instrumentation employed. The results themselves are then presented in chapter five along with a discussion of their ramifications. The final chapter summarizes this work.

# Chapter 2

# Localization and the Density of States

## 2.1 Introduction

Inhomogeneities in the local crystal potential have been of interest for some time. As far back as 1950, Bardeen and Shockley considered inhomogeneities due to a varying elastic strain in a crystal that was otherwise homogeneous [18]. That work was generalized several years later by Kroemer, who treated arbitrary variations of the local crystal potential regardless of the source [19]. In the present case, these inhomogeneities arise from the random substitution of selenium atoms for sulfur atoms on the group VI sub-lattice (or *vice versa*). Despite these local potential variations, the delocalization of the carriers is generally sufficient to average out differences locally, resulting in electronic and optical properties that are intermediate to those of unalloyed cadmium sulfide and cadmium selenide. This averaging corresponds to the virtual crystal approximation (VCA) in which the correct one electron potential,

which depends on the specific atomic configuration of the alloy, is replaced by the potential averaged over all the possible configurations [21]. This model was supported by the results of first-order perturbation theory as presented by Muto [23] and as corrected for higher order effects by Parmenter [20]. One result of such an averaging is the occurence of the band edge at energies which vary smoothly with $x$, the mole fraction of sulfur in the element VI A sites.

Despite such an averaging of properties, Parmenter found that the density of states should penetrate the forbidden band gap while rapidly tailing off. One result of this tailing of the density of states can be observed in the band edge smearing in substituitional alloys, in which case the tailing results from local composition fluctuations. A mechanism for such smearing was proposed by Alferov and co-workers for the pseudobinary alloy $CA_xB_{1-x}$, where the atoms A and B occupy one of the sets of sub-lattice sites and the C atoms occupy the other [24]. In their model $n$ is taken to be the concentration of the sub-lattice sites occupied by the A and B atoms, and $x$ is the average fraction of these sites occupied by A atoms. Then one can estimate the fluctuational change in the fraction of A sites in a volume $R^3$, where $R$ is large in comparison to the interatomic distances. From the binomial distribution [25], one can readily show that the mean number of A atoms in the volume is given by $xnR^3$ with a standard deviation given by $(xnR^3)^{1/2}(1-x)^{1/2}$. This yields a change in the fraction of A atoms given by

$$\Delta x = \frac{(xnR^3)^{\frac{1}{2}}(1-x)^{\frac{1}{2}}}{nR^3} \tag{2.1}$$

in the volume. As a result of this compositional fluctuation, a shift in local band energy $E$ from its virtual crystal average value $E(x)$ would then be expected to

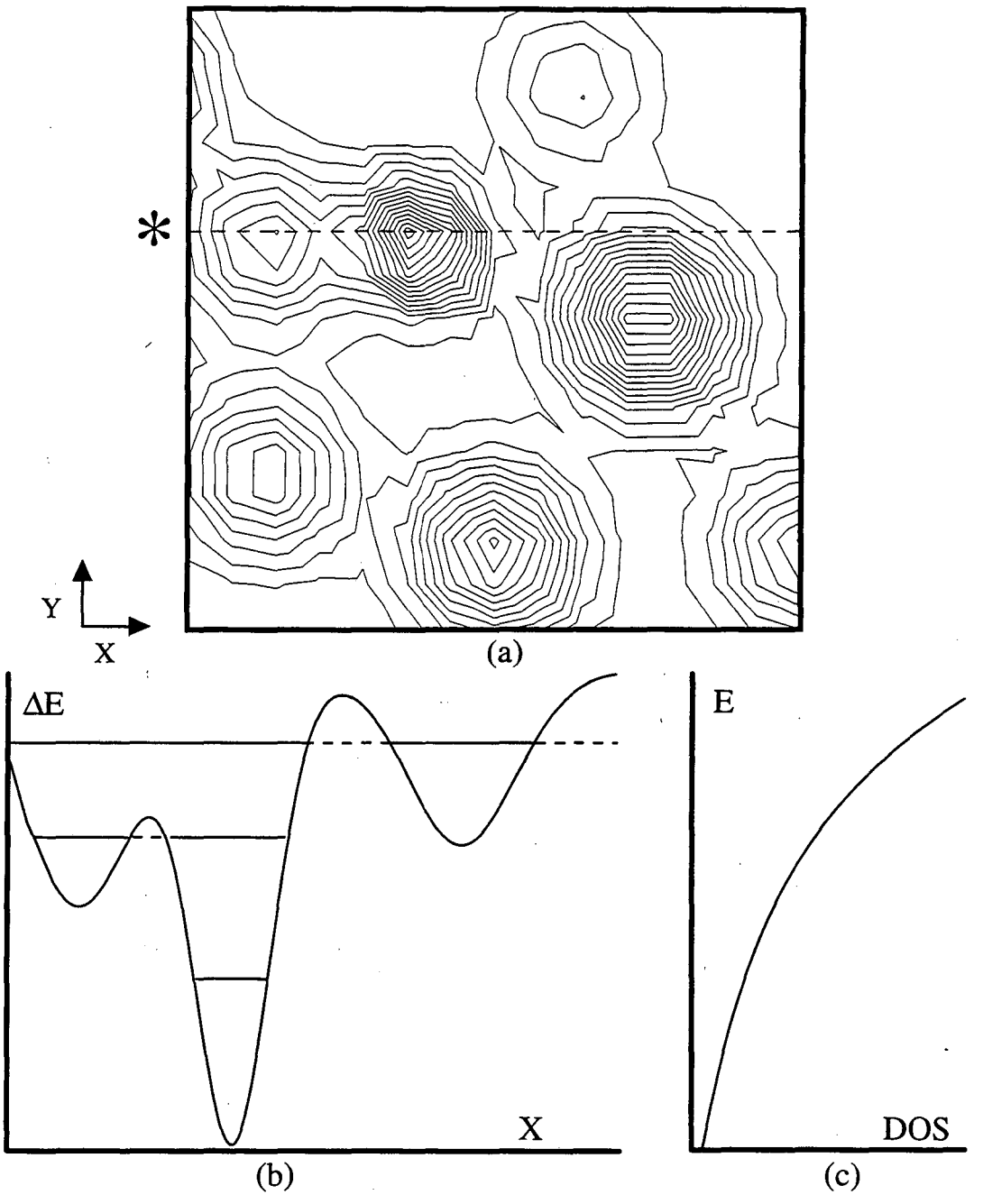


Figure 2.1: Shown above in (a) is a contour plot of potential wells due to composition fluctuations. The potential as a function of position along the dotted line (*) is shown in (b). Because the deeper states bound by such wells are expected to occur less frequently, the density of states will tail off into the gap as shown in (c).

occur. For small values of the fluctuation, this can be expressed as

$$E = E(x) + \alpha \Delta x, \tag{2.2}$$

where $\alpha = (dE/dx)_{\Delta x=0}$. Equation 2.2 says that a local potential well (or hump) is created in the volume of interest and has a depth given by

$$\epsilon = \frac{\alpha (xnR^3)^{\frac{1}{2}}(1-x)^{\frac{1}{2}}}{nR^3} \tag{2.3}$$

with respect to the virtual crystal band edge $E(x)$. Such wells will be particularly important if they sustain electronic levels; furthermore, since the well deepens as $R$ decreases, the most important wells are those for which $R$ is the minimum capable of sustaining such levels. This occurs for

$$R = \frac{\pi \hbar}{\sqrt{8m\epsilon}} \quad , \tag{2.4}$$

where $m$ is the effective mass of the band under consideration. The net result is a smearing of the band edge of order

$$\epsilon = \frac{512\alpha^4 m^3 x^2 (1-x)^2}{\pi^6 \hbar^6 n^2} \quad . \tag{2.5}$$

Thus, in this simple model, smearing is expected to be a maximum for $x = 1/2$ when the disorder is greatest; furthermore, smearing will be greater for the valence band where the holes have a greater effective mass.

## 2.2 Anderson Localization Theory

The treatment of the preceding section focuses on the statistical distribution of atoms, not on averages. As Anderson demonstrated in 1958, this focus is crucial to understanding the nature of the states that tail into the band gap [26]. His theory predicts that states may exist as a continuum in energy but be localized in space, a prediction that though challenged on several occasions [27, 28] has been supported by experiments [29–31] and simulations [32] alike. Furthermore, there should exist a characteristic energy, called the mobility edge, at which the states undergo a transition from being localized to being delocalized [33, 34]. Spectral evidence for the mobility edge has also been reported [35, 36]. Although this theory has been discussed at length in the literature [26, 37–41], a brief review is in order.

Anderson began with a simple linearized tight-binding model of non-interacting particles that could be described by the Hamiltonian

$$H = \sum_i E_i n_i + \sum_{ij} V_{ij} c_i^\dagger c_j \quad , \tag{2.6}$$

where $E_i$ is the energy of a particle at site $i$ and is chosen from a random distribution of width $W$ and $V_{ij}$ is the matrix element describing hopping from the site $j$ to the site $i$. The remaining operators are the number operator $n_i$, the creation operator $c_i^\dagger$, and the destruction operator $c_j$ for particles at site $i$ or $j$ as indicated. In this general form, the particles could be electrons, phonons, or even spinors, this latter being historically related to the development of the theory [42, 43]. In any case, this Hamiltonian leads to the linear equation of motion expressed by

$$i\hbar \dot{c}_i^\dagger = E_i c_i^\dagger + \sum_j V_{ij} c_i^\dagger \quad . \tag{2.7}$$

If $W = 0$ such that all $E_i$ are the same, this equation describes a standard band of Bloch states. However, when $W \neq 0$, one must consider the size of $W$ relative to $V = \overline{V_{ij}}$ and treat the smaller of the two as a perturbation. Which is chosen crucially affects the outcome. If $W$ is chosen as the perturbation, the outcome predicts that at every site there exists a continuum of energy states of every energy in the band; *i.e.* the states are extended. On the other hand, if one chooses $V_{ij}$ as the perturbation, localization is predicted.

In order to demonstrate this result, the initial eigenstates and energies are taken as $\phi_i^o = \phi_i$, the tight binding wave functions, and $E_i^o = E_i$ respectively with $V_{ij}$ taken as the perturbation. Then an approach strikingly similar to a Green's function propagator technique [44] is employed. However, whereas the propagator

$$G_{kk} = \frac{1}{E - E_k - \Sigma_k(E)} \tag{2.8}$$

describes the temporal evolution of a particle in a state with a wave vector $\overrightarrow{k}$, the "locator"

$$G_{ii} = \frac{1}{E - E_i - \Sigma_i(E)} \tag{2.9}$$

describes the evolution of a particle at location $i$. The symbol $\Sigma$ used in these equations represents the self-energy, which is itself a perturbation series. For the locator case,

$$\Sigma_i(E) = \sum_{j \neq i} \frac{|V_{ij}|^2}{E - E_j} + \sum_{j,k \neq i} \frac{V_{ij} V_{jk} V_{ki}}{(E - E_j)(E - E_k)} + \dots \ . \tag{2.10}$$

This expression can perhaps be more clearly represented diagramatically as shown in Fig. 2.2 where the expansion of $\Sigma_i(E)$ as a locator series is compared to the

analagous expansion of $\Sigma_k(E)$ in the standard Feynmann diagrams.

Using the locator series, the next step is to add a complex component $is$ to the energy and consider the result of taking $s$ to 0. If $\lim_{s\to 0} \operatorname{Im} \Sigma_i(E + is)$ is non-zero, then $-\operatorname{Im} G_{ii}$, which is essentially just the density of states, will consist of a continuum of energy states of every energy in the band at every site $i$. Each state would then be proportional to $N^{-1/2}$, where $N$ is the number of sites; therefore, as $N \to \infty$, each state at site $i$ would become infinitesimally small. The states would clearly be extended. On the other hand, if $\lim_{s\to 0} \operatorname{Im} \Sigma_i(E + is)$ is zero, then $-\operatorname{Im} G_{ii}$ will be the sum of a discrete infinite series of delta functions with convergent coefficients. As will be shown, this latter result is true for the case at hand.

The remaining analysis can be limited to the first term in the series representation of $\Sigma_i(E)$. With a finite $s$,

$$\Sigma_i(E + is) = \sum_{j\neq i} \frac{|V_{ij}|^2}{(E - E_j) + is} \tag{2.11}$$

so that

$$\operatorname{Im}(\Sigma_i) = \sum_{j\neq i} \frac{s|V_{ij}|^2}{(E - E_j)^2 + s^2} \quad . \tag{2.12}$$

In the limit as $s$ tends to zero, the imaginary part of the self-energy also tends to zero with $\operatorname{Im}\Sigma$ proportional to $s$ in the limit. To see how this tendency gives rise to localized states, one should consider the necessary conditions for $\operatorname{Im}(\Sigma/s)$ to be peaked at an energy $E$. For finite $s$ a peak will occur only when $|E - E_j| < s$ and $V_{ij} > s$. Taking

$$V_{ij} = V_o e^{-R_{ij}/R_o} \quad , \tag{2.13}$$

which is physically reasonable for hopping between states separated sufficiently such that only the exponential tails of the wavefunctions overlap, $V_{ij} > s$ implies that

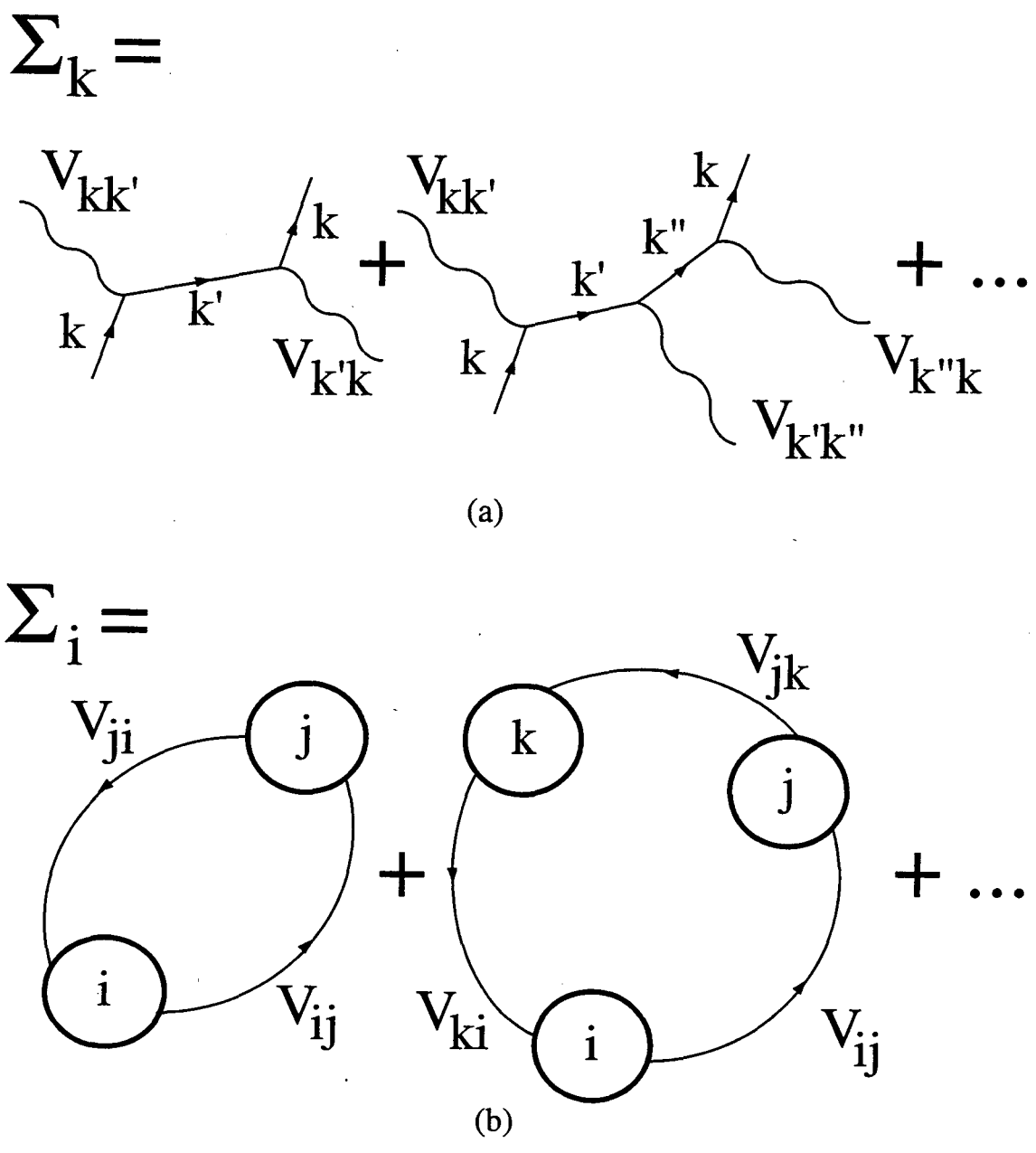


Figure 2.2: Shown above is the expansion of the self-energy in terms of (a) a propagator series and (b) a locator series. Which expansion is appropriate depends on whether the states are best defined as plane waves or spatially localized states. In the former case, the propagator expansion applies; in the latter case, the locator expansion applies.

$R_{ij} < R_o \ln(s/V_o)$. Now, as $s$ tends to zero, the probability that a given energy $E$ will be a peak also goes to zero as the peaks collapse to infinitesimal width. Furthermore, the probability of making a hop from the site $j$ to the site $i$ also goes to zero. In the end $-\operatorname{Im} G_{ii}$ is the sum of a discrete infinite series of delta functions with convergent coefficients. The normalization condition $\sum_i |\phi_\alpha(r_i)|^2 = \sum_\alpha |\phi_\alpha(r_i)|^2 = 1$ is satisfied not because each of the states $\phi_\alpha(r_i)$ is infinitesimal as in the extended case but because the states form a hierarchy with the largest being of order 1, the second of order 1/2, *etc.*, in a convergent series.

One should note that at no time in the preceding analysis was an average taken. This is critical for arriving at the prediction of localized states. For example,

$$\langle \lim_{s\to\pm 0} \operatorname{Im}\Sigma(E+is)\rangle \approx \pm\frac{\langle V_{ij}^2\rangle}{W} \quad . \tag{2.14}$$

If this is used in determining $G_{ii}$, one arrives at the conclusion that

$$G_{ii} = \frac{1}{E - E_j - \operatorname{Re}\Sigma \mp i\langle V_{ij}^2\rangle/W} \quad , \tag{2.15}$$

which implies there is a continuum of states at each site. This clearly contradicts the results from above and occurs because of the fundamental failure to treat the system in terms of the distribution. Indeed most of the advances in the theory of amorphous systems stem from this shift in attitude from treating them as dirty regular systems to treating them as fundamentally different.

In general, the higher order terms do not change these results qualitatively but only serve to renormalize the matrix elements $V_{ij}$ and the energies $E_j$. However, if the higher order terms of the perturbation expansion renormalize $V(R_{ij})$ to such an extent that it is no longer effectively exponentially localized with respect to the

distances between sites, the perturbation treatment fails; the extended case results. Furthermore, because the self-energy is a function of energy, there can exist a critical energy $E_m$ at or above which this perturbation approach does indeed fail. In this case there are both extended and localized states within the same energy band, albeit at different energies. The energy separating these states is called the mobility edge [34, 45], since the states below $E_m$ are localized while those above $E_m$ are extended.

This mobility edge can also be predicted by considering the average separation $\langle R_{ij} \rangle$ between tail states within an energy range of $\Delta\epsilon$ about $\epsilon$, where $\epsilon$ is taken as the energy of a state below the unsmeared band edge. This value is essentially given by

$$\langle R_{ij} \rangle \sim [g(\epsilon)\Delta\epsilon]^{-1/3} \quad , \tag{2.16}$$

where $g(\epsilon)$ is the density of states [30]. In order for a resonant transition from one site to another to occur, the matrix element $V_{ij}$ must satisfy $V_{ij} \gtrsim \Delta\epsilon$. For tunneling between isolated states, this matrix element can be taken as

$$V_{ij} \sim \epsilon e^{-(2m\epsilon/\hbar^2)^{1/2}\langle R_{ij} \rangle} \tag{2.17}$$

as the result of Eqs. 2.4 and 2.13, where $m$ is the appropriate mass for the particle under consideration, *e.g.* the effective mass of the electron or of the hole or of the exciton. The state will therefore be delocalized if there exists an energy $\Delta\epsilon$ such that

$$\ln\left(\frac{\epsilon}{\Delta\epsilon}\right) \gtrsim \sqrt{\frac{2m\epsilon}{\hbar^2}}[g(\epsilon)\Delta\epsilon]^{-1/3} \quad . \tag{2.18}$$

For sufficiently small $g(\epsilon)$, that is deep enough in the tail, no such energy will be possible; the states will be localized. Higher up in the tail, however, the states become delocalized. The energy of this transition is again just the mobility edge $E_m$.

## 2.3 Density of States

Because these localized states exist in the tail of the density of states, excitons immediately come to mind. Indeed at low temperatures the dominant photoluminescence band in some semiconductor alloys occurs below the free exciton reflectivity structure and has been interpreted as due to the recombination of excitons localized due to composition fluctuations [46, 47]. Models of the tail of the density of states have been developed by several authors [30, 48–50] and shown to vary as

$$g(\epsilon) \sim e^{-(\epsilon/\epsilon_o)^a} \quad , \tag{2.19}$$

where $a$ varies from .5 to 2 depending on the model and $\epsilon_o$ is a characteristic energy. In most of these studies, $\epsilon_o$ was taken as a fitting parameter. However, in the model of Ouadjaout and Marfaing [50], $\epsilon_o$ was calculated *a priori*. Thus, in order to more fully appreciate the localization of excitons in $CdS_xSe_{1-x}$, the model of the density of localized exciton states induced by composition fluctuations as developed by Ouadjaout and Marfaing will be explored.

The model assumes a typical pseudobinary semiconductor alloy of the form $CA_xB_{1-x}$ as considered previously. Again $x$ is taken as the macroscopic average fraction of A–B sub-lattice sites occupied by A atoms. For a small volume $V$, however, the average local composition is given by $\xi$ such that the fluctuations in the composition can be expressed as $\Delta x = \xi - x$ for the particular volume under consideration. In order to obtain the crucial probability distribution for the composition fluctuations, Ouadjaout and Marfaing took advantage of the fact that the probability $p$ should vary as $e^{-S}$, where $S$ is the entropy fluctuation and can be obtained from the

work of Lifshitz as [51, 52]

$$S = \frac{V}{V_c}[\sigma(\xi) - \sigma(x) - (\xi - x)\sigma'(x)] \quad . \tag{2.20}$$

Here $\sigma(x)/V_c$ is the entropy concentration for a homogeneous concentration $x$ in a volume $V_c$ called the interaction volume. Taking this interaction volume as the smallest volume in which the fluctuation $\Delta x$ can exist, one obtains

$$V_c = \frac{1}{n|\Delta x|} \quad , \tag{2.21}$$

where $n$ is the concentration of sub-lattice sites. In this case $\sigma(x)$ may be expressed as

$$\sigma(x) = -x \ln x - (1 - x) \ln (1 - x) \quad . \tag{2.22}$$

After some rearrangement one can then derive

$$p(\xi, V) \sim \exp\left\{-\frac{V}{V_c}\left[\xi \ln\left(\frac{\xi}{x}\right) + (1 - \xi) \ln\left(\frac{1 - \xi}{1 - x}\right)\right]\right\} \quad . \tag{2.23}$$

Assuming $\Delta x$ is small with respect to $x$ and $1 - x$, which assumption corresponds to a reasonably random distribution of the atoms on the sites as is generally the case experimentally, one obtains

$$p(\Delta x, V) \sim \exp\left[-\frac{V}{V_c}\frac{|\Delta x|^2}{x(1 - x)}\right] \quad . \tag{2.24}$$

If one were to assume that $V_c$ were the volume of a single sub-lattice site, then this would correspond to the results of Alferov [24] since the binomial distribution converges to the normal distribution for large numbers of sites [25]. However, using

Eq. 2.21, this probability becomes

$$p(\Delta x, V) \sim \exp\left[-\frac{nV|\Delta x|^3}{x(1-x)}\right] \quad . \tag{2.25}$$

The next step is to determine how these fluctuations localize excitons and at what energies they do so. As it turns out, there are essentially three manners in which localization can be achieved dependent on the width and depth of the wells associated with the fluctuations. First, the well could localize only one of the electron and hole with the other bound by Coulombic attraction. Second, the exciton could be localized as a single particle in a well larger than its Bohr radius. Since both of these cases correspond to the localization of a single particle, they can be treated simultaneously. The third manner of localization is the result of confining both the electron and hole in the same narrow well. This approach must be considered when the localization is stronger than the electron–hole interaction; the result is a confined exciton.

For the sake of simplicity, the local potential well giving rise to localization in the first two cases will be taken as spherical with a radius $R$ and depth $\Delta E$ as indicated in Fig. 2.3(a). The well can be taken to support a single bound state at an energy $\epsilon$ below that of the free exciton energy $E_X$ for the average macroscopic composition $x$. A necessary condition for this model to be valid for the localization of an entire exciton of mass $M = m_n + m_p$, where $m_n$ and $m_p$ are the effective masses of the electrons and holes respectively, is that $\epsilon$ is small compared to the exciton binding energy. The kinetic energy of confinement can then be given by $\theta = \Delta E - \epsilon$. Taking $\alpha_X = dE_X/dx$, one obtains

$$\Delta E = \theta + \epsilon = -\alpha_X \Delta x \quad , \tag{2.26}$$

where fluctuations are assumed to create a true well, *i.e.* $\Delta E > 0$. The next step is to determine the characteristics of the well, $\Delta E_{op}$ and $R_{op}$, which maximize the probability $p(\Delta x, V)$ for a given $\epsilon$. This optimization may be executed by solving

$$\frac{d}{dR}(R^3|\Delta x|^3) = \frac{d}{dR}\left[\frac{R^3(\Delta E)^3}{|\Delta x|^3}\right] = 0 \tag{2.27}$$

subject to the constraints imposed by the relationship of $\epsilon$ to $\Delta E$ and $R$. Assuming the spherical potential well, this relationship is given by the transcendental equation

$$\cot\sqrt{2Z} = -\sqrt{\frac{Y}{Z}} \quad , \tag{2.28}$$

where the dimensionless variables $Z \equiv \theta M R^2/\hbar^2$ and $Y \equiv \epsilon M R^2/\hbar^2$ have been used [53]. Ouadjaout and Marfaing numerically solved the constraint as a third degree polynomial and substituted the resulting fit into Eq. 2.27, which can then be solved to yield

$$R_{op} = 1.4\frac{\hbar}{\sqrt{M\epsilon}} \tag{2.29}$$

and

$$\Delta E_{op} = 2.33\epsilon = 4.567\frac{\hbar^2}{MR_{op}^2} \quad . \tag{2.30}$$

Taking the density of states to be proportional to the probability $p(R_{op}, \Delta E_{op})$ gives

$$g(\epsilon) = g_o \exp\left[-\left(\frac{\epsilon}{\epsilon_o}\right)^{3/2}\right] \quad , \tag{2.31}$$

where $\epsilon_o$ is the characteristic energy given by

$$\epsilon_o = \gamma(1)\frac{M\alpha_X^2\,[x(1-x)]^{2/3}}{\hbar^2 n^{2/3}} \quad , \tag{2.32}$$

where

$$\gamma(1) = \frac{3^{2/3}}{(1.4 \times 2.33)^2 (4\pi)^{2/3}} = .036 \quad . \tag{2.33}$$

Ignoring any variation of $E_X$ with $x$, $\alpha_X$ may be replaced by $\alpha_g = dE_g/dx$, where $E_g$ is the band gap energy. For electrons and holes, $M\alpha_X^2$ is replaced by $m_n\alpha_n^2$ and $m_p\alpha_p^2$ respectively, where $\alpha_n$ and $\alpha_p$ are the conduction and valence band edge rates of change with composition.

For the simultaneous localization of both the electron and hole in the same well, a pair of spherical potential wells of radius $R$ as indicated in Fig. 2.3(b) is assumed. In this case the following equations apply:

$$\Delta E_n = \theta_n + \epsilon_n = -\alpha_n \Delta x \quad , \tag{2.34}$$

$$\Delta E_p = \theta_p + \epsilon_p = -\alpha_p \Delta x \quad , \tag{2.35}$$

and

$$\Delta E_g = \theta_n + \theta_p + \epsilon = -\alpha_g \Delta x \quad , \tag{2.36}$$

where $\epsilon = \epsilon_n + \epsilon_p$ and $\alpha_g = \alpha_n - \alpha_p$ have been used. Again the assumption is that $\Delta E_n$ and $\Delta E_p$ are both greater than zero. A constrained optimization problem similar to the one considered above is then solved to obtain the optimal well radius $R_{op}$ and composition fluctuation $\Delta x_{op}$ for a given energy $\epsilon$. The results are

$$R_{op} = .617(1 - 4.39u)^{1/2} \frac{\hbar}{\sqrt{\mu\epsilon}} \tag{2.37}$$

and

$$\Delta x_{op} = \frac{13.7\epsilon}{(1 - 4.39u)\alpha_g} \quad , \tag{2.38}$$

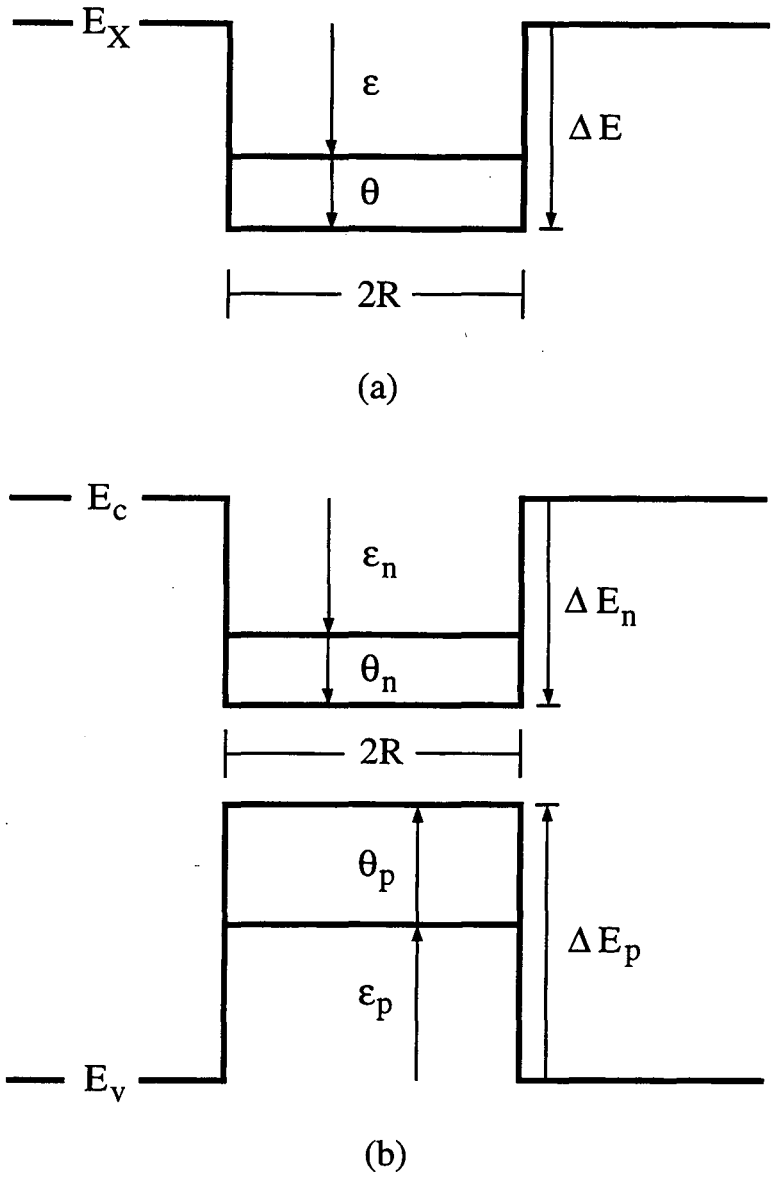


Figure 2.3: Shown above are the spherical potential wells assumed in the model of Ouadjaout and Marfaing [50] for both (a) single particle localization and (b) double particle localization.

where

$$u = \frac{\alpha_n^2 m_n + \alpha_p^2 m_p}{\mu \alpha_g^2} \tag{2.39}$$

and $\mu$ is the reduced mass $\mu = [m_n^{-1} + m_p^{-1}]^{-1}$. Solving for the density of states yields the same form as Eq. 2.31 but with

$$\epsilon_o = \gamma(u) \frac{M \alpha_X^2 \left[x(1-x)\right]^{2/3}}{\hbar^2 n^{2/3}} \quad , \tag{2.40}$$

where

$$\gamma(u) = \left(\frac{3}{4\pi}\right)^{2/3} (.0802)(u + .228) \quad . \tag{2.41}$$

Using this model of the density of states, the value of $\epsilon_o$ can be calculated for any alloy if the values of $m_n$, $m_p$, $\alpha_n$, $\alpha_p$, and $n$ are known or can be reasonably estimated. For example, for $CdS_{.36}Se_{.64}$, one can linearly interpolate the values of the effective masses from their values in CdS and CdSe [54] to get $m_n/m_o = .14$ and $m_p/m_o = .5$. Similarly one can obtain $n = 1.88 \times 10^{22}$ cm$^{-3}$ from linearly interpolated lattice parameters. Furthermore, since the experimental [56] and calculated [57] values of $\alpha_n$ have opposite signs, its value can be taken to be zero. Then the value of $\alpha_p$ can be taken from the variation of the band gap with composition. Its value at $x = .36$ is then $\alpha_p = -.64$. Using these values and assuming that hole localization is dominant for $CdS_{.36}Se_{.64}$, which dominance will be shown in the next section, one calculates a value of $\epsilon_o = 5.2$ meV. This compares well with the value of $\epsilon_o = 4.3$ meV determined from fluorescence lineshape modeling [50].

In contrast to Eqs. 2.31 and 2.32, the model of Baranovskiĭ and Éfros [48], one of the few other models of the density of states that gives an *a priori* expression for $\epsilon_o$, indicates that the density of states is given by Eq. 2.19 with $a = .5$. This expression can also be used to determine $\epsilon_o$ from fluorescence lineshape analyses and

gives reasonable values around 3–4 meV. However, using their expression for $\epsilon_o$ and the alloy parameters used above gives a value that is several orders of magnitude smaller. The primary difference between their model and that of Ouadjaout and Marfaing is that they did not use a fluctuation dependent critical volume, which thus appears critical to obtaining good results.

## 2.4 Determining the Type of Localization

Up to this point, the different types of localization have been considered in terms of what effect each will have on the density of states. However, no mention has been made of what conditions will determine which type of localization. As has already been noted, in either the case of single particle or of double particle localization, the characteristics of the localization of the alloy are expressed in terms of the quantities $m_n$, $m_p$, $\alpha_n$, $\alpha_p$, and $n$. In fact, the relative probabilities of the different types of localization can actually be determined solely from the ratios $m_n/m_p$ and $\alpha_n/\alpha_p$ as will now be shown.

Localization of the exciton as a whole requires only that the potential well be wide with respect to the Bohr radius of the exciton, $a_B$. This weak localization condition can then be expressed as

$$R \gg a_B = \sqrt{\frac{\hbar^2}{2\mu E_{BX}}} \quad , \tag{2.42}$$

where $E_{BX}$ is the exciton binding energy. Incorporating the expression for the optimal radius for such a localization leads to the condition

$$\frac{\epsilon}{E_{BX}} \ll \frac{1.4}{1 + m_n/2m_p + m_p/2m_n} \quad . \tag{2.43}$$

Since $m_n/m_p$ is generally around 0.2, $\epsilon$ is typically much smaller than the free exciton energy and corresponds to only a small part if any of the luminescence bands. This center of mass localization can therefore be ignored in the analysis of the remaining three types of localization.

If the electron alone is to be localized, then the hole potential well must satisfy

$$\Delta E_p = \alpha_p \Delta x < \frac{\pi^2}{8} \frac{\hbar^2}{m_p R_n^2} \quad , \tag{2.44}$$

where

$$R_n = 1.4 \frac{\hbar}{\sqrt{m_n \epsilon}} \tag{2.45}$$

is the optimum well width for localizing the electron from Eq. 2.29. Condition (2.44) as an equality is simply the minimal condition for the well to support a state as determined by setting Eq. 2.28 to zero. This condition can then be expressed in terms of $\alpha_n/\alpha_p$ as

$$-\frac{\alpha_n}{\alpha_p} > 3.7 \frac{m_p}{m_n} \quad . \tag{2.46}$$

The probability that this type of excitation will occur must also be greater than the probability of localizing the hole alone or of localizing both the electron and hole in the same well. Such an optimization is equivalent to ensuring that at a given energy $\epsilon$ the density of states, or alternatively $\epsilon_o$, is greatest for localizing the electron alone. From Eqs. 2.32 and 2.40, this criterion becomes

$$\gamma(1) m_n \alpha_n^2 > \max\left\{\gamma(1) m_p \alpha_p^2, \gamma(u) \mu \alpha_g^2\right\} \quad . \tag{2.47}$$

If $-(\alpha_n/\alpha_p) > 0$, this is just equivalent to the previous condition (Eq. 2.46) for the typical case of $m_n < m_p$. On the other hand, if $-(\alpha_n/\alpha_p) < 0$, the same well will

not localize both an electron and a hole, so this condition is reduced to

$$\frac{\alpha_n}{\alpha_p} > \left(\frac{m_p}{m_n}\right)^{1/2} \quad . \tag{2.48}$$

If the hole alone is to be localized, then an analysis symmetrical to that for localizing solely the electron can be performed. In this case the condition that no electron be localized is given by

$$-\frac{\alpha_n}{\alpha_p} < \frac{1}{3.7}\frac{m_p}{m_n} \quad , \tag{2.49}$$

while the condition of maximum probability is given by

$$\gamma(1)m_p\alpha_p^2 > \max\left\{\gamma(1)m_n\alpha_n^2, \gamma(u)\mu\alpha_g^2\right\} \quad . \tag{2.50}$$

If $-(\alpha_n/\alpha_p) < 0$, the latter condition is reduced to

$$\frac{\alpha_n}{\alpha_p} < \left(\frac{m_p}{m_n}\right)^{1/2} \tag{2.51}$$

as the well will not simultaneously localize both a hole and an electron. However, in contrast to the previous analysis, the case of $-(\alpha_n/\alpha_p) > 0$ is more complicated. In order to ensure the simultaneous satisfaction of both the conditions (2.49) and (2.50), the following inequality must be satisfied [50]:

$$P(s,t) = \sum_{i=0}^{4} C_i(t)s^i < 0 \quad , \tag{2.52}$$

where $s \equiv -\alpha_n/\alpha_p$, $t \equiv m_n/m_p$, and the coefficients $C_q$ are given by

$$\begin{aligned} C_0(t) &= -.19t^{-1} - .815 \ , \\ C_1(t) &= .369 \ , \\ C_2(t) &= .805(t+1) \ , \\ C_3(t) &= .369t \ , \\ C_4(t) &= .81t^2 + .995 \ . \end{aligned} \tag{2.53}$$

If the electron and hole are both localized in the same well, then, from the above analyses, $-(\alpha_n/\alpha_p) > 0$ as a result of the criteria for the existence of a localized electron and a localized hole given by

$$-\alpha_n \Delta x > \frac{\pi^2}{8}\frac{\hbar^2}{m_n R^2} \tag{2.54}$$

and

$$-\alpha_p \Delta x > \frac{\pi^2}{8}\frac{\hbar^2}{m_p R^2} \tag{2.55}$$

respectively. Incorporating the optimal values of $R$ and $\Delta x$ given by Eqs. 2.37 and 2.38, these conditions are transformed into

$$\left[4.23\left(1+\frac{m_n}{m_p}\right)-1\right]^{-1} < -\frac{\alpha_n}{\alpha_p} < \left[4.23\left(1+\frac{m_n}{m_p}\right)-1\right] \ . \tag{2.56}$$

The maximum probability condition in this case is given by

$$\gamma(u)\mu\alpha_g^2 > \max\left\{\gamma(1)m_n\alpha_n^2, \gamma(u)m_p\alpha_p^2\right\} \ . \tag{2.57}$$

The combination of the conditions (2.56) and (2.57) then just yields the previously

encountered conditions (2.46) and (2.52) with the inequalities reversed:

$$-\frac{\alpha_n}{\alpha_p} < 3.7\frac{m_p}{m_n} \tag{2.58}$$

and

$$P(s,t) > 0 \quad , \tag{2.59}$$

where $P(s,t)$ is defined by Eqs. 2.52 and 2.53.

A plot of $-(\alpha_n/\alpha_p)$ verses $(m_n/m_p)$ can then be used to describe the type of localization that will occur in a given semiconductor alloy, ignoring the center of mass localization. Such a plot is given in Fig. 2.4. This plot consists of four regions. The uppermost, for $-(\alpha_n/\alpha_p) > 3.7m_p/m_n > 0$, and the lowermost, for $-(\alpha_n/\alpha_p) < -(m_p/m_n)^{1/2} < 0$, correspond to alloys in which the electron alone is localized and the hole is bound to the electron through the Coulombic interaction. Of the two remaining regions, the one that straddles $-(\alpha_n/\alpha_p) = 0$ corresponds to alloys in which the hole alone is localized. The final region corresponds to alloys in which both the electron and hole are localized in the same well as a confined exciton. Exactly where a given alloy falls on this plot, and hence the type of localization that occurs in that alloy, then depends on the specific values for the critical parameters. For example, in $CdS_xSe_{1-x}$, the value of $\alpha_n$ is taken to be zero as indicated in the last section. Thus, in these alloys the hole alone is localized. While hole localization has been assumed in earlier models [55], this was the first *a priori* prediction of such a result.

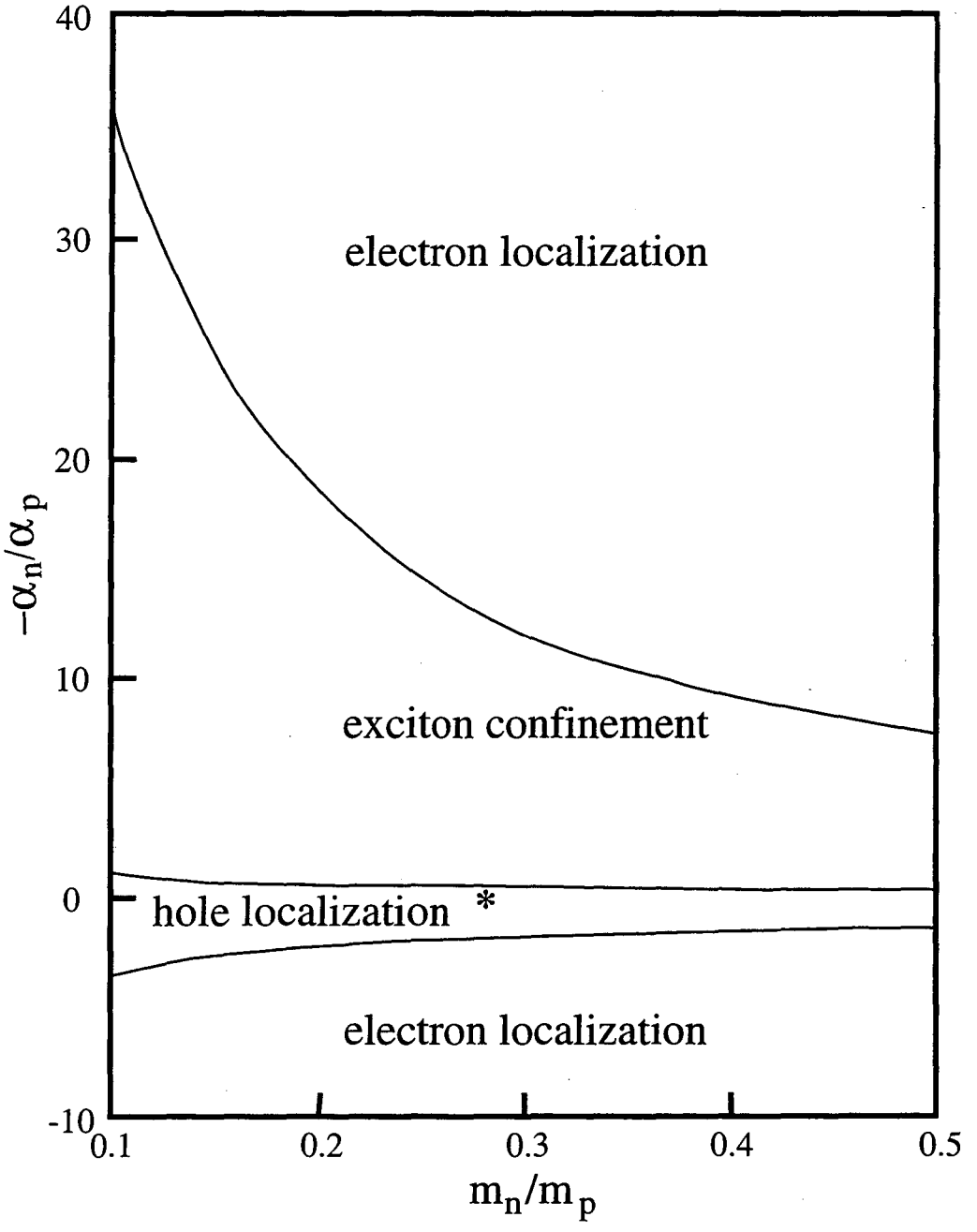


Figure 2.4: Shown is a graph of $-\alpha_n/\alpha_p$ versus $m_n/m_p$ with the boundaries separating different types of localization. The $*$ marks the location of $CdS_{.36}Se_{.64}$ as given by Ouadjaout and Marfaing [50].

## 2.5 Summary

In summary, the virtual crystal approximation implies that the properties in an alloy should be an average of those of the unalloyed materials. However, such an approach, which fails to take into account the full statistical nature of the potential fluctuations but rather treats them in only an average way, is not always appropriate. As Anderson's model shows, there can exist a continuum of states in energy without those states being delocalized. Nevertheless, in real alloy systems such localized states only exist in a tail of the density of states which extends into the band gap. The majority of states in the band are indeed extended and the energy that separates the extended states from the localized states is called the mobility edge.

Furthermore, from the model of Ouadjaout and Marfaing, the density of states can be seen to vary with the energy below the virtual crystal average energy according to $\exp\left[-(\epsilon/\epsilon_o)^{3/2}\right]$ where the characteristic energy $\epsilon_o$ has a value of a few meV for the alloys $CdS_xSe_{1-x}$. In addition, in these alloys localization occurs for the holes in the valence band while the electrons are bound to the holes by Coulombic attraction.

Because of the small value of $\epsilon_o$, the samples must be cooled to very low temperatures in order to observe the localized states directly. Otherwise, the excitons are readily excited out of the states through interaction with phonons and dissociated into free carriers. In sub-nanosecond resolution studies, the extended exciton states were observed to become dominant above about 40 K [58]. However, as this dissertation will establish, even at room temperatures the states in the tail of the density of states have an effect on the carrier dynamics. While the carriers may spend little time in those states, the lifetime dependency on the energies of those states gives rise to a distribution of radiative pathways through which the carriers

can decay. This distribution affects the shape of the luminescence decay as will be demonstrated and discussed in Chapter 5.

# Chapter 3

# Ultrafast Fluorescence Techniques

## 3.1 Time Correlated Single Photon Counting

Time correlated single photon counting (TCSPC) is a well established instrumental method for following luminescence decay that has been in use for over three decades. It is based on the concept that the probability of emission of a single photon corresponds to the intensity as a function of time for all of the emitted photons. Collection of the photons emitted following many excitations then yields the probability distribution. The technique was first applied in 1961 by Bollinger and Thomas to the study of scintillators excited by alpha, beta, and gamma radiation [59]. Shortly thereafter scintillator emission decays were measured with resolution in the nanosecond regime [60]. Since that early work, TCSPC has been applied to many fluorescence decay experiments and seen great improvements in resolution and data collection rate through the development of instrumentation for excitation and detection.

One example of improvement in photoexcitation sources is the development

of cavity dumped dye lasers synchronously pumped by mode-locked ion or solid state lasers, which represent a significant advancement over previous flashlamp sources [61]. With pulse widths on the order of 1 ps currently available for such systems, resolution of about 40 ps is limited by the detection system. Additionally, the megahertz regime repetition rates of these systems have reduced the data collection time of high signal to noise ratio decays to a matter of a few minutes or less and made feasible the examination of weakly emitting samples. Finally, the dye laser provides flexibility in wavelength selection. Using a variety of dyes, wavelengths spanning the visible spectrum are available.

The detector of choice for TCSPC is the microchannel plate (MCP) detector [62]. Although the gain of MCP detectors is less than that of photomultiplier tubes (PMT's), the latter are currently limited to a resolution of about 300 ps while the former have resolutions as low as 70 ps. This enhancement in resolution results from the fact that electron multiplication occurs in small glass capillaries in between the photocathode and anode with lengths that can be as small as .5 mm. This is compared to a length that can easily be several centimeters for PMT's. Thus, in MCP's the transit time spread is greatly reduced. However, because of the lower gain of the MCP, amplification of the anode pulses in necessary. To avoid additional broadening of the instrument response function, the amplifier should have a 1 GHz or better bandwidth. In addition, the shape of the instrument response function of PMT's is wavelength dependent; MCP detectors present no such difficulty.

A standard sequence of events in the collection of data is as follows. A small portion of the exciting laser beam is split off from the main beam and directed to a fast photodiode which serves as a timing reference. It provides a start signal for a time to amplitude converter (TAC), which is just a temporally linear voltage ramp. Meanwhile, the remainder of the laser beam excites the sample. When an

emitted photon is subsequently collected by the collection optics, it is directed to the MCP detector, the anode pulse of which serves as a stop signal to the TAC. The resulting output of the TAC is then collected by a computer or a multi-channel analyzer in which the channels correspond to linear increments of time. This process is repeated until after many excitations a histogram of the probability distribution for the emitted photons is mapped out.

One minor variation of this scheme is to reverse the roles of the photodiode and MCP. In this scenario the MCP output starts the TAC while the photodiode signal due to the subsequent laser pulse stops it. Because the timing between pulses is well defined, the collection of this inverted data is equivalent to normal collection. The advantage of this scheme is that the TAC is not started with every laser pulse but only when a photon is actually collected. This minimizes the dead time associated with the resetting of the TAC.

Sending the start and stop signals through constant fraction discriminators before they pass on to the TAC is necessary to minimize the instrument response function width. Without such discrimination, the TAC must trigger on the relatively jittery edges of the MCP and photodiode pulses. Such jitter is the result of the fluctuations of the amplitude from pulse to pulse. A constant fraction discriminator serves as follows to sharpen the edge of the pulses on which the TAC triggers. It first splits an incoming pulse into two pulses with a constant amplitude ratio, one of which it inverts and delays with respect to the other. It then recombines the two pulses. At the point where the voltage of the summed pulses passes through zero, the discriminator emits an output pulse which is sent to the TAC. This pulse is far less jittery than the output of the MCP. Provided that the shape of the input pulses does not vary significantly from pulse to pulse apart from fluctuations in amplitude factors, the spread of triggering times for pulses arriving at the same nominal time

(as results for scattered excitation pulses, for example) is considerably narrowed. In fact, without constant fraction discrimination, the instrument response function width can increase from less than 100 ps to greater than 1 ns. A schematic of a typical TCSPC setup is presented in Figure 3.1.

Even with an instrument function as short as 70 ps or less, there are many phenomena with timescales of the same (or shorter) order. In this case, obtaining quantitative information from the decays requires taking into account the size and shape of the instrument response function. The measured decay is the convolution of the instrument response function with the fluorescence decay of the sample according to

$$s(\tau) = \Phi(t) * r(t) = \int_{-\infty}^{\infty} \Phi(t) r(\tau - t) dt \quad , \tag{3.1}$$

where $s(t)$, $\Phi(t)$, and $r(t)$ are the measured signal, the fluorescence decay function, and the instrument response function respectively. When the instrument response function is much shorter than the characteristic decay times of the fluorescence, it may then be treated as if it is a delta function, in which case $s(t) = \Phi(t)$. On the other hand, when the two are of comparable magnitude, they must be deconvolved in order to fit the fluorescence decay function. The method by which such a deconvolution is achieved will be covered in the next chapter.

In order to obtain undistorted data, the detector must receive at most one photon per excitation. Because only the first photon to arrive at the detector is counted, arrival of multiple photons per excitation will skew the decay curve to early times [63]. To ensure that arrival of multiple photons is rare, the excitation intensity and collection optics must be adjusted so that detection events occur a maximum of once for every twenty excitation pulses. The probability of such a rare event can

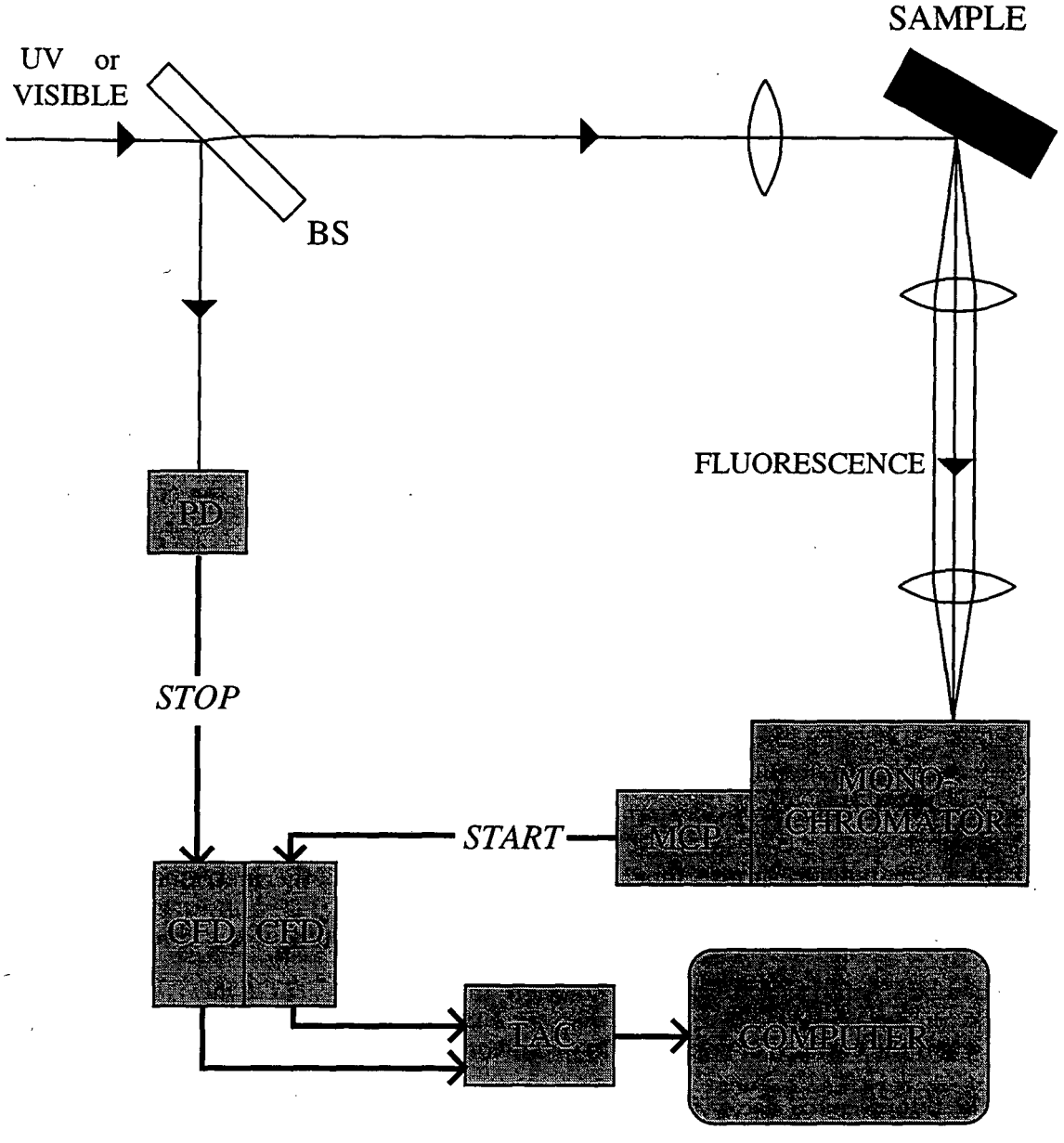


Figure 3.1: Shown is a schematic of a typical TCSPC setup. The ultraviolet or visible laser beam is split at the beamsplitter (BS), with one portion going on to excite the sample. The fluorescence is collected and focused into a monochromator before reaching the micro-channel plate (MCP) detector. The output of the MCP provides the *start* pulse for the time to amplitude converter (TAC). The *stop* pulse comes from a photodiode (PD) in response to the succeeding laser shot. Both the start and stop pulses pass through a constant fraction discriminator (CDF) before passing on to the TAC. The TAC signal is then passed to a computer which serves as a multi-channel analyzer.

then be calculated via the Poisson distribution

$$p(n;\lambda) = \frac{\lambda^n e^{-\lambda}}{n!} \quad , \tag{3.2}$$

where $p(n;\lambda)$ is the probability of $n$ photons arriving at the detector following a single excitation pulse and $\lambda$ is the average number of photons detected per such pulse. An average of $\lambda = .05$ then yields a probability of $p(2;.05) = .001$ for the arrival of two photons at the detector following a single excitation pulse. Attenuation of the emission so that detection occurs only once for every one hundred excitation pulses is better still, yielding a probability of $p(2;.01) = 5 \times 10^{-5}$. Fortunately repetition rates of the order of megahertz are available so that data collection at or above 10 kHz is readily achievable.

Because semiconductor samples can be damaged if excited by high power lasers, low power, high repetition rate lasers should be employed in their study. Fortunately these laser systems lend themselves wonderfully to TCSPC experiments as described above because of the high sensitivity and need for high repetition rates in order to avoid the difficulties imposed by the counting statistics. In addition, because the signal counts are synchronized to the laser pulse, the dark count rate can be kept to a minimum, as low as a few counts per second. Thus, the method allows for a superb signal-to-noise ratio. The method of TCSPC is therefore a very useful technique for measuring fluorescence decays with lifetimes of 100 ps or longer.

As a result of these features, fluorescence decays have been measured by TCSPC for a wide variety of systems, especially chemical and biological ones. For example, solvation dynamics have been probed via TCSPC measurements of the fluorescence decays of probe molecules such as aminophthalamides [64, 65]. The fluorescence decays of trans-stilbene [68], tryptophan [66, 67], and mutants of *Chlamydo-*

*monas reinhardii* [69] have also been measured by this technique. The lattermost measurements provided valuable insight into the interactions of photosystems I and II during photosynthesis.

The method does have its drawbacks, however. First is the limit in the resolution. Because the timing is provided by the electronics, the full capabilities of ultrafast lasers, especially of sub-picosecond lasers, go unused. Furthermore, the instrument function can be quite complicated as a result of structured output pulses from the MCP. As will be seen later, this can make deconvolution of the instrument function from the measured decay quite difficult. Both of these limitations can be overcome, albeit at the expense of time and noise, by the use of fluorescence upconversion as described in the next section.

## 3.2 Fluorescence Upconversion

### 3.2.1 Introduction

The second ultrafast fluorescence technique employed in these studies was fluorescence upconversion, also known as sum frequency generation, which was first employed in 1975 by Mahr and Hirsch [70]. Upconversion is an example of a class of nonlinear spectroscopic techniques known as frequency mixing. In such techniques photons of two, often different, frequencies are combined in a nonlinear crystal to produce photons with frequencies equal to the sum or difference of the two input frequencies. In the particular case of fluorescence upconversion, an ultrafast laser beam is split into two beams. The first serves as a source of photoexcitation for the sample. The luminescence produced is then combined with the second, or probe, beam in a nonlinear crystal to produce radiation with a frequency equal to the sum of

the frequencies of the luminescence and the laser. The theory of these techniques has been well discussed elsewhere [71–73] and will therefore only be briefly considered below.

Since upconversion can occur only when the luminescence spatially and temporally overlaps with the probe pulse in the nonlinear crystal, that pulse acts as a light gate and allows sampling of the luminescence at the time of overlap. Figure 3.2 gives a pictoral representation of this action. Furthermore, this gating provides temporal resolution on the order of the pulse width, provided certain conditions are fulfilled. This is of particular interest due to the advent of femtosecond laser systems [74], which can improve resolution to less than .5 ps [75–77] in comparison to electronic techniques such as time correlated single photon counting, which is limited to at best 40 ps due to the electronics employed. By varying the time at which the laser pulse passes through the nonlinear crystal, the researcher can sample the luminescence as a function of time. Provided that the gating beam is stable, a histogram of the probability for photon emission can therefore be mapped out.

Control of the time at which the laser pulse passes through the nonlinear crystal is achieved by varying the path length of the probe pulse with respect to the sum of the path lengths of the excitation and fluorescence. All time-domain optical spectroscopies with pulse-width limited resolution employ such path length variation as a means of obtaining temporal information. Precise spatial control as small as 1 $\mu$m is routinely achieved with readily available commercial translation stages. Such a distance corresponds to advancing or delaying a pulse of light by a mere 3 fs. Thus, control of the temporal overlap in the nonlinear crystal need place no restraints on the ultimate resolution of this technique.

Sum frequency generation, of which frequency doubling is but a special case,

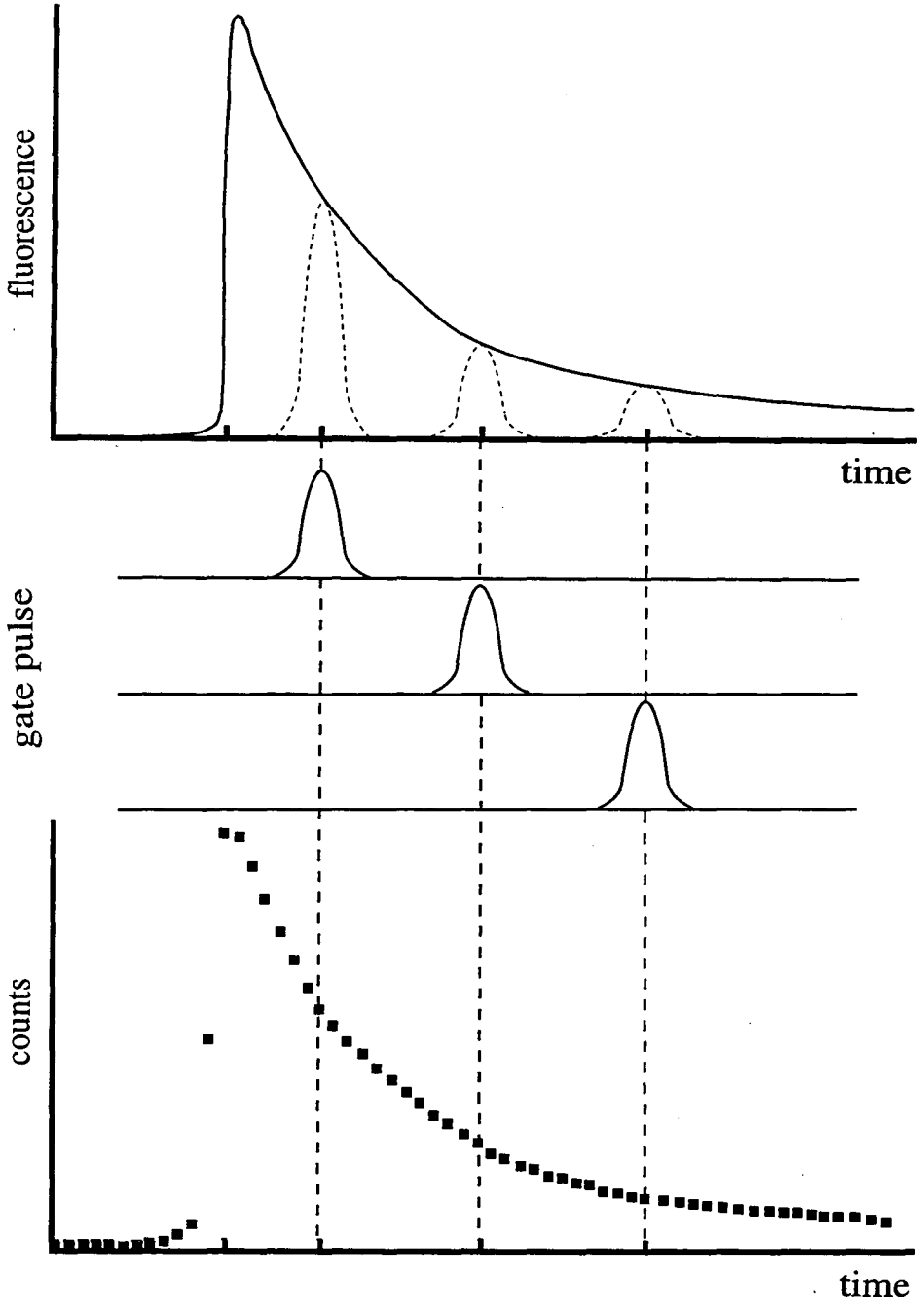


Figure 3.2: Shown is a schematic of how a gate pulse controlled to pass through the nonlinear crystal at various times acts as a light gate for luminescence from the sample at those times. By changing when the gate pulse passes through the crystal with respect to the excitation of the sample, each part of the decay can be gated.

results from a non-zero second order susceptibility, $\chi^{(2)}_{ijk}(\omega = \omega_1 + \omega_2)$. The susceptibility naturally results from the electronic structure of the material in question and must therefore reflect the structural symmetry of that material. An important consequence of this fact is that for a material with inversion symmetry $\chi^{(2)} = 0$ in the electric dipole approximation. Thus, in order to carry out fluorescence upconversion, frequency mixing must occur in an anisotropic, or nonlinear, material. In such materials the index of refraction depends on the direction of propagation. Let $n_x$, $n_y$, and $n_z$ be the indexes of refraction along the principal dielectric axes $x$, $y$, and $z$ of the medium. In the most common nonlinear crystals used for sum-frequency generation, $n_x = n_y \equiv n_o$, and $n_z \equiv n_e$ where $n_o$ is called the ordinary index, $n_e$ is called the extraordinary index, and the $z$ axis is termed the optic axis. Light polarized normal to the optic axis is said to be ordinary (O) while light polarized normal to the ordinary direction is said to be extraordinary (E). The remainder of the discussion on the theory of sum frequency generation will therefore be limited to these uniaxial crystals.

### 3.2.2 Phase-matching

In order to obtain efficient upconversion of the fluorescence, strict phase-matching conditions must be fulfilled. The equations describing phase-matching are

$$\omega_{sum} = \omega_{sig} + \omega_{gate} \tag{3.3}$$

and

$$\vec{k}_{sum} = \vec{k}_{sig} + \vec{k}_{gate} \quad , \tag{3.4}$$

where $\omega_i$ and $\vec{k}_i$ are the frequency and wave vector respectively for the fluorescence signal to be upconverted (sig), the gate pulse (gate), and the sum frequency photons (sum). In the special case of collinear phase-matching, *i.e.* $\vec{k}_{sig} \parallel \vec{k}_{gate} \parallel \vec{k}_{sum}$, Eq. 3.4 reduces to

$$\frac{n_{sum}}{\lambda_{sum}} = \frac{n_{sig}}{\lambda_{sig}} + \frac{n_{gate}}{\lambda_{gate}} \quad , \tag{3.5}$$

where $n_i$ and $\lambda_i$ are the index of refraction and wavelength respectively for the indicated photons. Because the index of refraction in an anisotropic crystal depends on the direction of propagation, these conditions can often be fulfilled by adjusting the angle of incidence $\theta$ of the incoming photons with respect to the optical axis, provided that the three wavelengths bear the appropriate polarization dependence for the crystal being used. For negative uniaxial crystals, those in which $n_o > n_e$, phase-matching requires the sum frequency photons to be E polarized. In the case of both the signal and gate photons polarized ordinarily, *i.e.* O + O $\rightarrow$ E, the phase-matching angle $\theta_m$ is given by [78]

$$sin^2\theta_m = \frac{(1/n^2_{sum}(\theta_m)) - (1/n^2_{o,sum})}{(1/n^2_{e,sum}) - (1/n^2_{o,sum})} \quad , \tag{3.6}$$

where

$$n_{sum}(\theta_m) = n_{o,sig}\frac{\lambda_{sum}}{\lambda_{sig}} + n_{o,gate}\frac{\lambda_{sum}}{\lambda_{gate}} \tag{3.7}$$

and $n_{o,i}$ and $n_{e,i}$ are the ordinary and extraordinary indexes of refraction at wavelength $\lambda_i$ respectively. In practice a commercially obtained crystal cut so as to fulfill the phase-matching conditions for a given pair of input wavelengths at normal incidence to the crystal surface can be employed over a somewhat narrow frequency range by adjusting the angle of incidence to fulfill the phase-matching conditions for new combinations of wavelengths. This is referred to as angle tuning.

Given perfect phase-matching, the quantum efficiency, $\eta_q(0)$, can be calculated by solving the three coupled wave equations subject to the appropriate boundary conditions [71, 72]. The result indicates that $\eta_q(0)$ depends on $L^2$ and $I_{gate}$ in addition to the susceptibility, where L is the crystal thickness and $I_{gate}$ is the intensity of the gating beam. In short, the more intense the gate beam and the greater its spatial overlap with the signal beam, the more photons at the sum frequency that will be generated. When the phase-matching conditions are not satisfied either due to wavelength or angle mismatch, the quantum efficiency is decreased. In terms of the phase mismatch, $\Delta k = k_{sum} - k_{sig} - k_{gate}$, the quantum efficiency falls off with increasing $\Delta k$ as

$$\eta_q(\Delta k) = \eta_q(0)\frac{sin^2(L\Delta k)}{(L\Delta k)^2} \quad . \tag{3.8}$$

The spectral bandwidth of the crystal can therefore determine the spectral resolution if the luminescence spectrum is broad. However, the spectral bandwidth of the crystal is generally much broader than the desired spectral resolution. To improve the resolution, the upconverted signal can be dispersed in a monochromator. Such dispersion will also help reduce stray light. The drawback to this technique is that the crystal must be angle tuned synchronously with the wavelength tuning of the monochromator in order to cover the entire spectral range of the luminescence. Further complication can arise because tuning over a wide angle can displace the beam at the entrance slit of the monochromator.

Because the sum frequency intensity $I_{sum}(\tau)$ at a delay time $\tau$ is proportional to the cross correlation of the gate intensity with the fluorescence signal intensity,

$$I_{sum}(\tau) = \text{Corr}\,(I_{gate}, I_{sig}) \sim \int_{-\infty}^{\infty} I_{gate}(t+\tau)I_{sig}(t)dt \quad , \tag{3.9}$$

both the fluorescence and gate beams are focused into the crystal to increase the output. However, the focused beams then form a cone of focused photons only a fraction of which can be perfectly phase-matched. The remainder are mismatched to a greater degree as their angles deviate further from the phase-matching angle. The range of angles over which mismatch can occur without significantly reducing the upconversion efficiency is called the acceptance angle and can be expressed as [79]

$$\Delta\theta = \frac{4\pi}{L}\left(\frac{\partial k}{\partial \theta}\right)^{-1} . \tag{3.10}$$

Since the acceptance angle varies inversely with crystal thickness $L$, the beams may be more tightly focused into thinner crystals. This focusing roughly offsets the losses in sum frequency generation due to decreased crystal thickness, though the benefits from tighter focusing are limited by the extent to which the fluorescence can be focused.

Another consideration in designing an upconversion experiment in terms of the efficiency of the upconversion process concerns walk-off. This phenomenon occurs because the direction of energy flow for extraordinary rays, in contrast to the case for ordinary rays, differs from the direction of wave propagation given by the wave vector. The angle between these two directions is called the walk-off angle, $\theta_w$, and for the case of the sum frequency ray being extraordinary as considered above is given by [78]

$$\tan\theta_w = \frac{n_{sig}^2(\theta_m)\sin(2\theta_m)}{2}\left(\frac{1}{n_{e,sig}^2} - \frac{1}{n_{o,sig}^2}\right) . \tag{3.11}$$

Clearly this limits the efficiency if the signal and gate rays are not both ordinary, since the spatial overlap will decrease as the rays traverse the crystal and walk-off

from one another. Fortunately, for the case O + O → E, the gate and signal rays are both ordinary and thus undisturbed by this effect. The sum frequency ray, however, will emerge from the crystal at a position offset from that of the fluorescence and gate. The optical alignment must therefore be adjusted to account for this effect.

### 3.2.3 Dispersion

As has been indicated above, maximizing spatial overlap of the fluorescence and gate beams favors thick crystals. However, the dispersive nature of the crystal can decrease resolution and therefore places practical limits on the crystal thickness. Treating an optical pulse as a wave packet with central frequency $\omega_o$, the wave number $k$ can be expanded in a Taylor's series with respect to frequency $\omega$ to obtain [80]

$$k(\omega) = k(\omega_o) + \left(\frac{\partial k}{\partial \omega}\right)_{\omega=\omega_o} d\omega + \left(\frac{\partial^2 k}{\partial \omega^2}\right)_{\omega=\omega_o} d\omega^2 + \dots \ , \tag{3.12}$$

where higher order terms may generally be neglected for pulses with widths exceeding 10 fs [81, 82]. The inverse of the magnitude of the linear term, $|(\partial k/\partial \omega)|^{-1} \equiv v_g$, is termed the group velocity of the pulse. The quadratic term, $(\partial^2 k/\partial \omega^2)$, is the group velocity dispersion and represents the change in pulse shape during propagation. Both of these terms can contribute to the temporal broadening of an ultrafast pulse.

Noting that $\vec{k} = n(\lambda)\omega/c$ and $\lambda = 2\pi c/\omega$, the group velocity can also be expressed as

$$v_g = c\left[n(\lambda_o) - \lambda_o\left(\frac{\partial n}{\partial \lambda}\right)_{\lambda=\lambda_o}\right]^{-1} \ , \tag{3.13}$$

where $c$ is the speed of light in a vacuum. Clearly pulses of different wavelengths will generally have different group velocities. As a result, in addition to the transverse walk-off described previously, the input pulses will move ahead of the sum frequency

pulse they are generating. As they continue to generate more sum frequency, the upconverted pulse grows wider in time. Fortunately, in fluorescence upconversion experiments, the resolution is not determined by the width of the sum frequency pulse but rather by the uncertainty in the temporal overlap of the input pulses. However, the input pulses themselves will become separated, which does increase this uncertainty. Resolution is therefore reduced. Now one might consider attempting to angle tune the crystal in hopes of matching group velocities. Unfortunately, in the visible and ultraviolet regions of the spectrum, simultaneously matching both the phase and the group velocities is generally impossible [80].

In addition to the group velocity mismatch, the group velocity dispersion leads to pulse broadening and thus also to decreased resolution. The group velocity dispersion can be expressed as

$$\frac{\partial^2 k}{\partial \omega^2} = \frac{\lambda^3}{2\pi c^2} \frac{\partial^2 n}{\partial \lambda^2} \quad . \tag{3.14}$$

The effect of the group velocity dispersion is to cause the components of the pulse with different wavelengths to travel at different rates. In the visible and ultraviolet regions of the spectrum, the longer wavelengths generally travel faster than the shorter wavelengths. The resulting pulse therefore contains a frequency sweep in addition to being broadened. As the input pulses pass through the nonlinear crystal, their overlap is increased temporally; and resolution is decreased.

Broadening due to group velocity dispersion can be compensated by introducing a grating pair or a prism pair in order to introduce an anomalous dispersion, *i.e.* a group velocity dispersion in which the blue components travel faster than the red ones. On the other hand, broadening due to group velocity mismatch cannot be corrected; its minimization through the insightful selection of a crystal (and other

optics) is the best that can be achieved. Fortunately neither of these broadening mechanisms is serious enough to require correction for pulse widths greater than about 1 ps; however, for pulses just one order of magnitude shorter, ignoring these effects can lead to a pulse's broadening by a factor of five or more [83].

When choosing a nonlinear crystal for the upconversion process, all of the factors affecting the upconversion frequency must be considered. In addition, the frequency at which the crystal itself absorbs must also be considered. Thus, while $LiIO_3$ has a quantum efficiency roughly double that of $\beta$-barium borate (BBO) and fifty times that of potassium dihydrogen phosphate (KDP), it absorbs wavelengths shorter than 300 nm. As a result, if sum frequencies near or below 300 nm will be produced, $LiIO_3$ cannot be used. In contrast, BBO does not begin to absorb until 190 nm. BBO is also much less dispersive than $LiIO_3$; for sub-picosecond laser pulses, $LiIO_3$ will introduce much more broadening. Although KDP is less dispersive still than BBO, a BBO crystal of 1/3 the length of a crystal of KDP has a higher efficiency with comparable broadening as a result of BBO's much higher quantum efficiency [83]. In general BBO seems to show the best combination of properties for upconversion in the visible region of the spectrum.

### 3.2.4 Upconversion Systems

The laser system to be employed in fluorescence upconversion experiments is not as limited as that in the TCSPC experiments. Upconversion has been executed using both high power, low repetition rate lasers as well as low power, high repetition rate lasers. The former can yield larger signals but only at repetition rates of several kHz at best. For semiconductors, however, both damage thresholds and carrier densities must be taken into consideration. High power ultrafast lasers can

easily deposit sufficient heat to melt or otherwise damage semiconductor samples. Furthermore, at carrier densities above about $10^{18}$ $cm^{-3}$, plasma effects begin to complicate the carrier dynamics. Unless damage or plasmas are the particular focus of a study, low power lasers are more appropriate to such studies. The lower power is not a great limitation in any case, as the high repetition rates of such systems allow averaging over several million shots per second.

In addition to the differences in power and repetition rate, these two system designs differ in their means of signal averaging. The high power systems often achieve their mJ per pulse energies through multipass dye laser amplifiers. Such amplification, however, introduces significant noise, which is particularly detrimental in upconversion spectroscopy. Because the laser is used both to excite the sample and to generate the sum frequency that is ultimately detected, the resulting upconverted signal varies in intensity as the square of the laser intensity. Furthermore, if the laser is frequency doubled for sample excitation, the upconverted signal varies as the cube of the laser intensity. Thus, it is crucial to monitor the laser power fluctuations and normalize the upconverted signal to these changes. In contrast, the low power systems are unamplified and considerably more stable. As a result, the laser intensity fluctuations in these systems, which typically employ single photon counting detection, are generally not the major source of noise. Normalization is therefore unnecessary, and signal averaging is achieved by collecting data over a period of a few million laser shots per data point. Of course, at the available repetition rates, a few million laser shots require but a second or two.

Shah and coworkers at Bell Laboratories have made many of the significant recent developments in the application of low power, high repetition rate lasers to the fluorescence upconversion studies of semiconductors. The system used for the work presented in this dissertation was based primarily on their work; a schematic of such

a typical fluorescence upconversion setup is presented in Figure 3.3. In addition to using a laser system very similar to that used in the present work, they extended the technique by the application of optical fiber pulse compression techniques. In their earlier work [75], they compressed the output of a dye laser to obtain a resolution of less than 500 fs. They later compressed the fundamental of their Nd:YAG pump laser instead, which allowed them to pump their dye laser with 3.5 ps pulses as opposed to the typical 90 ps pulses [84]. This change improved their resolution to less than 400 fs. Most recently they improved the resolution to approximately 100 fs by employing pulse compression in both locations [85]. One should note that in this work the consideration of the dispersion of the optics as described above is of critical importance.

As noted above, temporal resolution in fluorescence upconversion experiments is pulse width limited. The sum frequency signal is the result of a cross correlation of the gate pulse with the fluorescence signal of the sample according to Eq. 3.9. However, the fluorescence signal is itself a convolution of the excite pulse, $I_{ex}(t)$, with the fluorescence decay function, $\Phi(t)$, *i.e.*

$$I_{sig}(\tau) \sim \Phi(t) * I_{ex}(t) = \int_{-\infty}^{\infty} \Phi(t) I_{ex}(\tau - t) dt \quad . \tag{3.15}$$

Since the excite pulse and the gate pulse are from the same laser pulse, they can be taken to differ only in magnitude. The measured decay then varies as the convolution of the fluorescence decay function with the autocorrelation of the gate pulse,

$$I_{sum}(\tau) \sim \Phi(t) * I_{resp}(t) = \int_{-\infty}^{\infty} \Phi(t) I_{resp}(\tau - t) dt \quad , \tag{3.16}$$

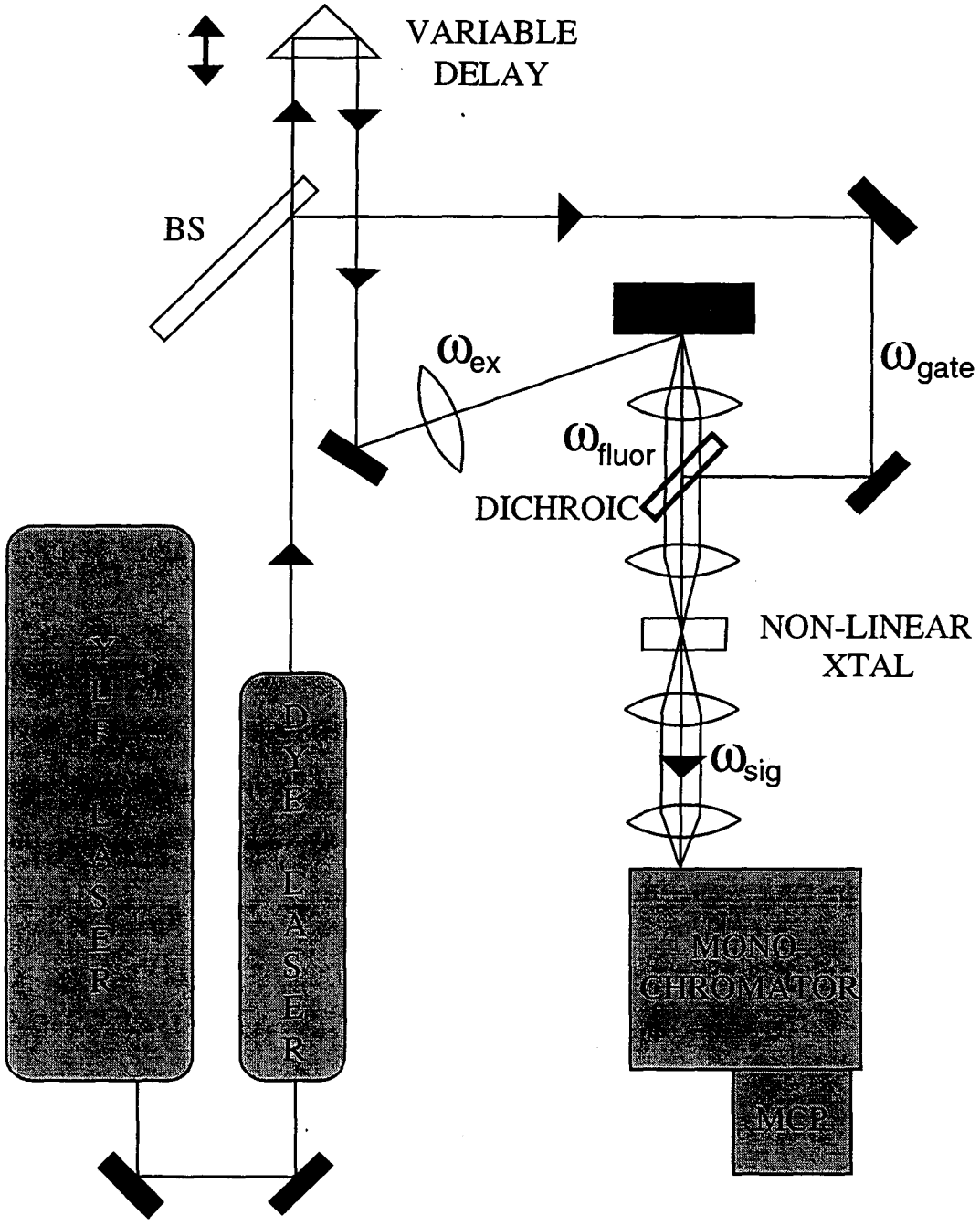


Figure 3.3: Shown is a schematic of a typical upconversion experiment using collinear phase-matching. Because the laser is used for both the excite and gate pulses, the excite and gate frequencies are the same, *i.e.* $\omega_{ex} = \omega_{gate}$. Then, from the phase matching conditions, the sum-frequency $\omega_{sig}$ must be the sum of the fluorescence and gate frequencies, or $\omega_{sig} = \omega_{fluor} + \omega_{gate}$.

where

$$I_{resp}(\tau) \sim \mathrm{Corr}\left(I_{gate}(t), I_{gate}(t)\right) = \int_{-\infty}^{\infty} I_{gate}(\tau + t) I_{gate}(t) dt \tag{3.17}$$

is the autocorrelation of the gate pulse. This autocorrelation serves as the instrument response function. If the laser pulse autocorrelation is much shorter than the characteristic decay times of the fluorescence, the instrument response function may be treated as if it is a delta function, in which case $I_{sum}(t) = \Phi(t)$. If this is not the case, the instrument function must be effectively deconvolved from the measured decay in order to fit the fluorescence decay function.

To obtain the instrument response function, the standard procedure is to scatter the excite beam off of the sample and mix it with the gating beam. If the two beams are of equal intensity, $I$, at the nonlinear crystal, then, when they are perfectly overlapped, the autocorrelation obtained will have an intensity proportional to $(2I)^2 = 4I^2$. However, in the case of collinear phase-matching, doubling can occur between two photons from the same pulse as well as between one photon from each. Thus, when the pulses are not overlapped at all, there will be a background proportional to $2I^2$, since each pulse will be individually doubled. The use of non-collinear phase-matching allows one to avoid this difficulty. In that case the phase-matching condition for adding together one photon from each pulse is not the same as for adding together two photons from within the same pulse. Furthermore, even if some same-pulse doubling does occur, albeit at a reduced efficiency, its wave vector will be different, allowing it to be spatially filtered. The drawbacks to non-collinear phase-matching are a reduced spatial overlap resulting in reduced efficiency and a greater difficulty in optimizing the overlap.

### 3.2.5 Summary

Clearly fluorescence upconversion is a powerful technique. As such it has been applied to a wide range of problems. In studies of semiconductors alone, it has seen widespread application from its earliest applications [98]. It has been used to study the slow rise of luminescence in GaAs following excitation by pulses at about .6 eV above the band gap [86]. This has been interpreted as due to intervalley scattering to the $L$ valley from which the carriers only slowly return to the $\Gamma$ valley where they act as a source of heating before cooling and finally decaying. This behavior is is stark contrast to that seen in InP where excitation was insufficient to achieve intervalley scattering [87]. In both cases carrier–carrier scattering was seen to play an important role. Fluorescence upconversion has also been employed in the studies of exciton transfer between the monolayer-flat islands that differ by one monolayer in thickness in GaAs single quantum wells [88] as well as in studies of spin relaxation in quantum wells of dilute magnetic semicondutor alloys [89].

In chemical systems upconversion has also found wide application, though generally through the use of high power laser systems as decscribed above [83]. For example, such systems have been used to study solvation dynamics by measuring the fluorescence decays of coumarin dyes [90, 91]. Fluorescence depolarization has also been measured by upconversion to obtain anisotropies for tryptophan [92]. These measurements are of interest due to the desire to use tryptophanyl residues as probes of protein structure and conformational dynamics.

## 3.3 A Comparison: Upconversion vs. TCSPC

As a conclusion to this chapter, a brief comparison of fluorescence upconversion and time correlated single photon counting will be made. Of the two upconversion clearly has the advantage in temporal resolution. Upconversion is pulse width limited whereas TCSPC is electronically limited. As a result, even using the same 1 ps laser system, a resolution difference nearly two orders of magnitude applies. With the application of pulse compression techniques or the implementation of lasers with inherently shorter pulses such as the Ti:sapphire laser, this difference only increases. Finally, the true resolution of TCSPC experiments is hindered by the electronic structure that is added to the instrument response function by the detection system. This leads to a difficulty of deconvolving the response from the signal. Because the response of the upconversion system is just the laser pulse autocorrelation trace, a much less noisy and unstructured response is obtained. Deconvolution techniques can then enhance the resolution.

Nevertheless, fluorescence upconversion does have some drawbacks. In comparison to other time resolved fluorescence spectroscopies such as TCSPC, upconversion yields worse signal to noise ratios. This is rooted fundamentally in the fact that even under ideal conditions only that fraction of the fluorescence signal which is upconverted can be detected. Because upconversion is a nonlinear process depending on the second order nonlinear susceptibility, this generally amounts to only a few percent at best. Furthermore, since fluorescence will generally contain a mixture of polarizations, only that fraction which is properly polarized for the phase-matching can even be upconverted. This will reduce the output by a further factor of 2. In addition, if the beams are poorly overlapped or the angle of the nonlinear crystal is not optimally set for phase-matching, still more of the fluorescence will be

lost. Optimal overlap can prove particularly difficult to obtain if the fluorescence is weak or, worse still, in the infrared where it is invisible. As a result, this technique is limited to the study of materials with a good quantum yield for fluorescence such as dyes and semiconductors.

In addition to the poorer signal to noise ratio, fluorescence upconversion suffers from the problem of the high background signal from second harmonic of the gate beam as discussed above. Even when the wavelengths of second harmonic and upconverted fluorescence are separated by as much as 50 nm, second harmonic can still be detected since the gate beam is typically about 5–7 orders of magnitued more intense. Although this interfering background may be reduced by using filters and a monochromator, this will also further reduce the amount of the upconverted signal that reaches the detector and thus decrease further the signal to noise ratio. One direct result of this difficulty is the need to excite well away from the peak of the luminescence to be examined. In semiconductors this amounts to exciting well above the band gap. Thus, studies of selective excitation, as for example exciting solely from the heavy hole sub-band as opposed to the light hole sub-band, are unfeasible.

A further drawback to fluorescence upconversion lies in the potential drift of the laser power over time. In upconversion the data for a single small time window is collected at each stage position. Thus, the data collected during a specific time interval in the lab corresponds to data from a specific time interval in the decay. If the laser drifts appreciably on a time scale of the same order as a full scanning of the stage, the data at the different positions will not be properly normalized. As already noted, the intensity of the signal is proportional to both the gate and fluorescence intensities, so laser drift can be particularly problematic. By comparison, TCSPC does not suffer from this difficulty, because there is never a time when it is collecting soley from a limited time window. At any given time during an experiment, the data

collected apply to all times in the decay with a probability weighted solely by the emission probability, provided the limits of the counting statistics are not breached. An additional result of this difference in collection is that it takes much longer to collect the decay using upconversion. Unfortunately, as already noted, laser drift places a practical limit on how long one can collect data with upconversion.

# Chapter 4

# Experimental

## 4.1 Samples

The $CdS_xSe_{1-x}$ samples studied in this work were obtained from Cleveland Crystals where they were grown by vapor deposition techniques. They were then cleaved along the major clevage plane perpendicular to the c axis and mounted on the end of a glass rod with epoxy without the introduction of any stresses. Samples with $x = 0.00$, .25, .50, .75, and 1.00 were employed in the time correlated single photon counting experiments while only those with $x = 0.00$, .25, and .50 were used in the fluorescence upconversion experiments. Attempts to measure decays via fluorescence upconversion using doubled laser fundamental for the excitation proved unfruitful in samples with $x = .75$ and $x = 1$. Spectra for the samples are shown in Figure 4.1.

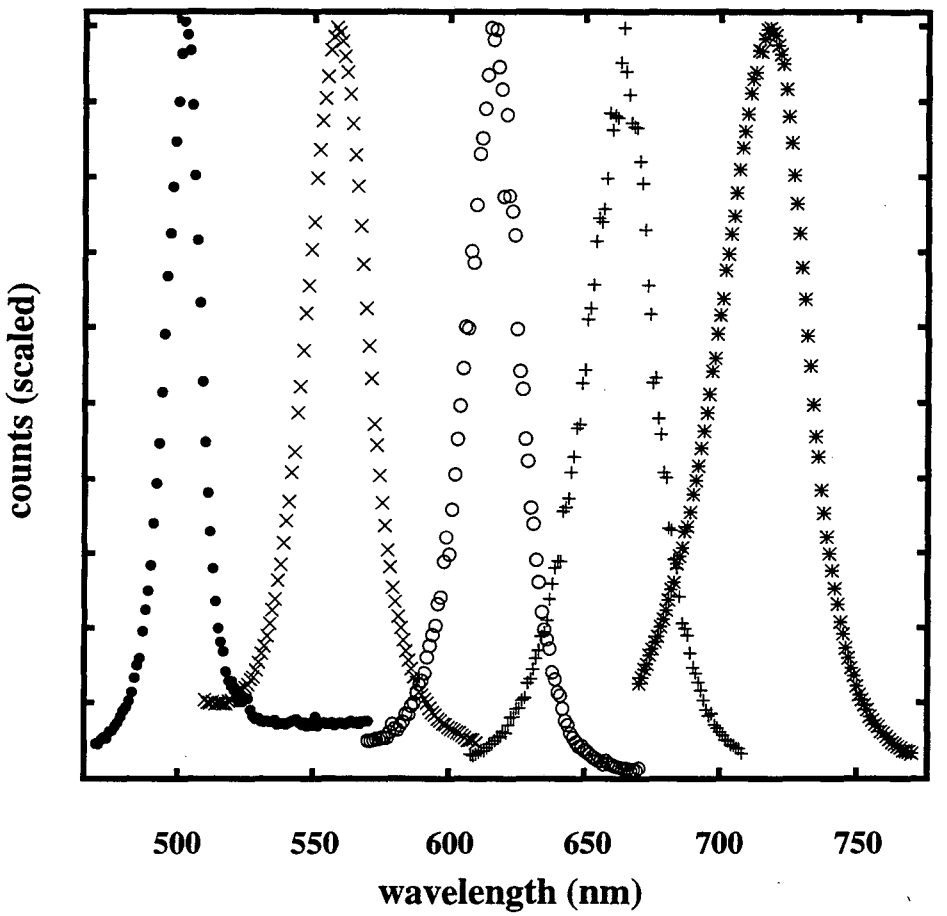


Figure 4.1: Shown above are the fluorescence spectra for the five samples studied in the present work. The spectra correspond to $x = 1$ ($\bullet$), $x = .75$ ($\times$), $x = .50$ ($\circ$), $x = .25$ ($+$), and $x = 0$ ($*$). The fluorescence maxima occur at wavelengths in agreement with Eq. 1.1.

## 4.2 The Laser System

The laser system employed for all of the present work was a Coherent 700 series dye laser synchronously pumped by a Coherent Nd:YLF laser. The pump laser was mode-locked at a 76 MHz repetition rate. The fundamental frequency was doubled to 527 nm using a heated KDP crystal. The output pulses were typically 70 ps wide and provided an average power of 2.0 to 3.0 W. The dye laser contained two dye jets, a gain jet and a saturable absorber jet, though the latter was employed only in the fluorescence upconversion experiments. The dye laser was cavity dumped at 3.8 MHz. In single jet mode, the output pulses were typically about 8 ps, far shorter than the resolution of 100 ps imposed by the electronics in the TCSPC experiments. In this case the average output power was typically 250 to 300 mW. In dual jet mode, pulses as short as 1.0 ps were obtained at the expense of power, which typically averaged 150 to 200 mW. In either case the output light was polarized vertically.

Several gain jet dyes were employed throughout this work, but the most frequently used dye was rhodamine 6-G. This dye has a broad gain curve centered at 590 nm, is extremely stable, and has the highest gain coefficient of any of the dyes used in this work. Using rhodamine 6-G, the laser could easily be tuned using a single plate birefringent filter from 570–630 nm, though power was of course lower for wavelengths away from the center. In addition to rhodamine 6-G, several other dyes were employed to provide a greater range of available wavelengths. Chief among these was DCM special, which was used at longer wavelengths than the rhodamine 6-G since its gain curve is peaked at 630 nm. To the blue of the rhodamine 6-G, rhodamine 570 was employed for a few experiments. This dye's gain curve is peaked at 570 nm. Unfortunately rhodamine 570 is considerably less stable than the other dyes employed. It degrades over a period of a few days as evidenced by a drop

in power and the concomitant color change from clear yellow to opaque orange-brown. This degradation is believed to be the result of a reaction with brass fittings commonly found in dye pumps; thus, the replacement of these parts with those of a non-reactive material such as stainless steel could alleviate this problem and extend the useful lifetime of this dye.

In order to obtain the shortest pulses, rhodamine 6-G was used in combination with the saturable absorber dye 3,3'-diethyloxadicarbocyanine iodide (DODCI). Its use to optimize pulse widths is, however, non-trivial. At a constant pump power, optimization requires carefully increasing the DODCI concentration while continually adjusting the cavity length and simultaneously monitoring the autocorrelation trace. For such monitoring a commercially available Femtochrome Research Inc. FR-103 autocorrelator was employed. The shortest pulses are achieved just before the laser stops lasing. However, since the average output power also decreases with the increase in saturable absorber concentration, the absolutely shortest pulses were not always ideal. Pulses in the 1.0–1.4 ps range were generally employed. Under the conditions of usage, the DODCI underwent a fairly rapid photodecomposition as evidenced by noticeable broadening of the autocorrelation trace. For example, over the course of several hours, the pulse could broaden by a picosecond or more unless fresh, concentrated DODCI was periodically added to the pump reservoir and the cavity dumper was finely adjusted.

## 4.3 Time Correlated Single Photon Counting

For the time correlated single photon counting experiments, the output of the dye laser was tuned to 630 nm and doubled in a $LiI0_3$ crystal to obtain 315 nm ultraviolet pulses for exciting the samples. This excitation was well above the band

gap of the widest band gap material in these studies, CdS, which has a band gap corresponding to 508 nm. After the remaining fundamental wavelength was filtered out, the ultraviolet pulse was focused at an oblique angle onto the semiconductor surface. A bright fluorescence spot was readily visible to the naked eye. The glass rod on which the semiconductor sample was epoxied was mounted in a post holder. This fluorescence was then filtered of scattered ultraviolet light with a KV-470 filter before being collected by a low f-number aspheric lens. Afterwards the fluorescence was recollimated and focused onto a .1 mm slit of a .10 m ISA single pass grating monochromator mounted on the front of the housing of a Hammamatsu R1564U microchannel plate detector. A resolution of less than 2 nm could thus be achieved. The anode pulse of the MCP was passed through a Hewlitt-Packard 8447D two-stage GHz pre-amplifier before passing on to an Ortec Model 583 constant fraction discriminator. Timing reference was provided by a fast EG&G FND-100 photodiode connected to a second Ortec constant fraction discriminator. The discriminator outputs were used to trigger a Canberra 2043 time-to-amplitude converter with the photodiode signal serving as the start pulse. The TAC output was then passed to an ADAC Model 1023 analog-to-digital converter, the output of which was sent to a computer which served the role of a multichannel buffer.

The instrument function of this TCSPC system was measured by scattering the visible dye laser fundamental off the semiconductor surface. A typical instrument function is shown in Figure 4.2. As can readily be seen, the instrument function possesses a 100 ps rise time and a FWHM of about 70 ps. These values are typical for many sub-nanosecond TCSPC systems. Because the samples exhibited characteristic decay times within an order of magnitude of this instrument function width, the instrument function cannot itself be ignored in fitting data as is the case when the width of the instrument function is smaller than the characteristic decay times by a

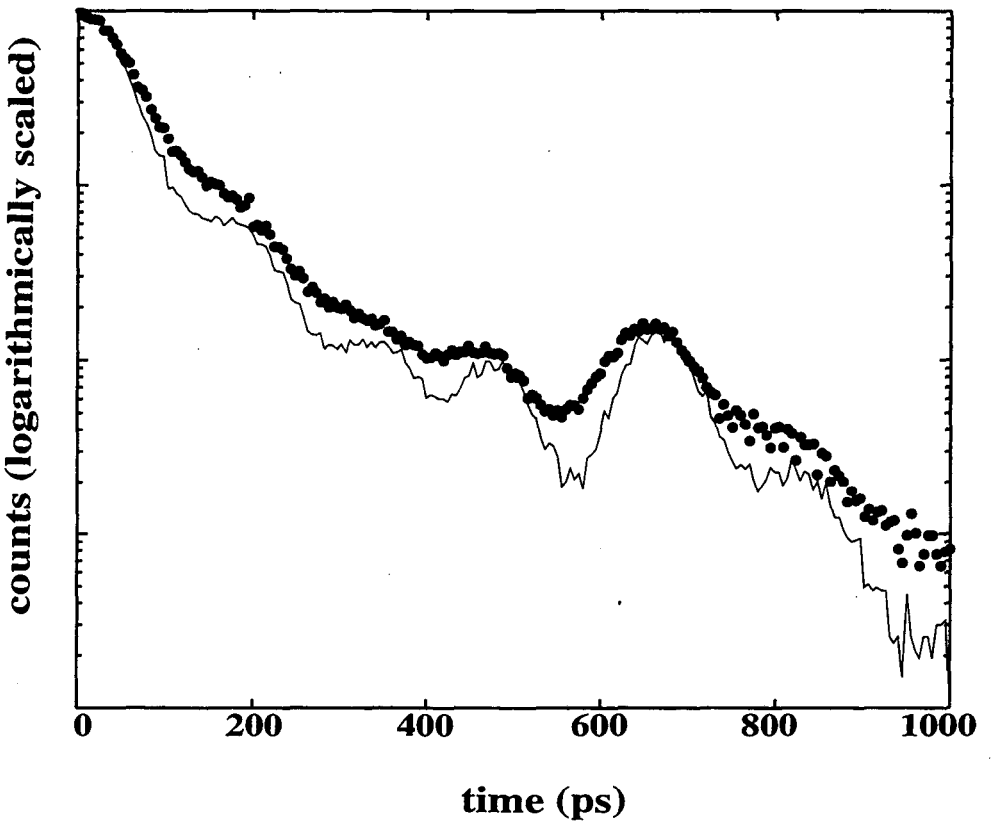


Figure 4.2: Shown above as the solid line is a typical instrument function taken with the time correlated single photon counting experimental setup described in the text. The structure in the instrument function is due to ringing in the MCP output and shows up quite clearly in the data as demonstrated by a sample of CdSe data given by the points.

factor of 10 or more. Thus, optimization of the instrument function, which has been described elsewhere [93], is critical. Furthermore, the output of the MCP detector contained ringing after the main pulse. Despite attempts to remove this feature, the ringing persisted. Thus, the instrument function contains structure that complicates fitting of the data.

## 4.4 Fluorescence Upconversion

As indicated above, the laser system used in the fluorescence upconversion experiments was essentially the same as that used for time correlated single photon counting. Whenever possible the dye combination rhodamine 6-G and DODCI was used, and the system was operated at 590 nm in which case the output pulses were typically 1 ps with an average power of 200 mW. For the $x = .50$ samples in which the band gap corresponds to 613 nm, however, this was not possible. While the excitation wavelength of 590 nm was still above the band gap, the upconverted signal could not be isolated from the frequency doubled gating pulse. In order to alleviate this difficulty, rhodamine 570 was employed and the laser was tuned to 550 nm to yield 9.5 ps pulses with an average power of 250 mW. No saturable absorber could shorten the pulses without lowering the power to unusable levels. In all cases the laser was operated at a repetition rate of 3.8 MHz and emitted vertically polarized pulses. All of the succeeding optics were aligned so as to ensure that this polarization was retained. The laser was then split using a Newport model 50G00AV.2 circular variable beamsplitter, which was set to transmit approximately 10% of the beam and reflect the remainder. The transmitted portion of the pulse train served as the excitation source; the reflected portion served as the gating beam.

After being split the transmitted beam passed through a path of vari-

able length consisting of a pair of mirrors mounted on a 150 mm Klinger model MT160.150 translation stage and aligned so as to steer the beam along a path parallel to the motion of the stage. The stage was driven by a Klinger model UE72 motor which was controlled by a Klinger CC1.1 programmable stepper motor controller. The controller was itself driven by a personal computer running a data collection program written by the author of the present work. The full program is given in Appendix A. The controller could step the stage in 10 $\mu$m increments, though the stage itself could potentially be stepped in 1 $\mu$m increments with a slightly different controller. In the configuration employed, however, steps of 10 $\mu$m correspond to a pathlength change of 20 $\mu$m, which advances or delays the pulse by 66.6 fs. With pulses of 1 ps FWHM, the resolution is clearly not limited by the stage control.

Having passed through the variable path, the beam was then focused onto the semiconductor sample with a CVI plcx25.4/18.0 uv-grade plano-convex lens having a 50 mm focal length. The beam was focused onto the surface of the sample at an angle less than 45 degrees. As in the TCSPC experiments, the sample itself was epoxied on the end of a glass rod which was mounted in a post holder. Luminescence was collected off the front side of the sample with an NRC microscope objective having a 20X amplification and a working distance of 1.8 mm from the tip of the lens. Preliminary work demonstrated that such a microscope objective functioned better than a low f-number lens by collecting luminescence over a greater solid angle and collimating the output into a tighter beam. Because the upconverted signal proved extremely sensitive to the fluorescence collection, several degrees of freedom were added to this portion of the optical arrangement in order to enhance the optimization of the collection efficiency. The collection optic could be rotated about a pair of perpendicular axes normal to the optical axis as well as be translated in all three dimensions. The latter adjustments proved to be the most crucial. The collected

fluorescence, having been collimated by the collection lens, then passed along the optical axis towards the nonlinear crystal used for upconversion.

After having been reflected from the variable beamsplitter, the gating beam traversed a fixed path to be collimated along the optical axis with the fluorescence prior to the nonlinear crystal. To achieve this collimation, a dichroic beamsplitter custom made by CVI was employed. The dichroic employed depended on the wavelengths of the gate and fluorescence in the particular case. In any case, however, the dichroic was designed to transmit the fluorescence while reflecting the gating pulse.

The two beams, fluorescence and gate, having been collimated along the optical axis, were then focussed into the nonlinear crystal, a 2 mm thick BBO crystal obtained from CSK, using a 5X microscope objective obtained from NRC. The microscope objective mounting assembly was mounted on a translation stage aligned along the optical axis to allow the tightness of focus in the BBO to be varied. Furthermore, because the orientation of the BBO crystal with respect to the experimental optical axis is critical for fulfilling the phase-matching requirements, the BBO crystal was mounted in a mirror mount providing rotational adjustment about the experimental optical axis as well as about a pair of axes perpendicular thereto. On the far side of the BBO crystal, a CVI plcx25.4/25.8 uv-grade plano-convex lens was employed to recollimate the upconverted fluorescence. In order to aid with overlapping the fluorescence and gate beams in the BBO crystal, a COHU high performance CCD camera was occasionally employed to image the back of the BBO crystal.

Following the upconversion assembly, a pair of filters were commonly employed to remove interfering light of which there were two primary sources. The first was remaining visible light, principally the laser fundamental used as a gating pulse; the visible was removed with a 3 mm thick UG11 filter. The second source

| sample | excite wavelength (nm) | pulse width (ps) | filters |
|---|---|---|---|
| CdSe | 590 | .9 | WG320, UG11 |
| $CdS_{.25}Se_{.75}$ | 590 | 1.1 | WG305, UG11 |
| $CdS_{.50}Se_{.50}$ | 550 | 9.5 | WG290, UG11 |

Table 4.1: These are the standard experimental conditions used in the fluorescence upconversion experiments. The large pulse widths in the experiments on the $x = .50$ sample were the result of not being able to find a saturable absorber at 550 nm that didn't cut down the power too greatly.

of interfering radiation was second harmonic of the laser fundamental. Even though the BBO crystal was angle tuned to phase-match the sum frequency of the gate with the fluorescence and not the second harmonic of the gating pulse, the angle of acceptance was sufficiently wide for the gating pulse, which was much more intense than the fluorescence, to add to itself. For the unalloyed CdSe, a 3 mm WG320 was standardly employed to remove the second harmonic. For the other samples, high pass filters with shorter wavelength cut-offs were used since both the sum frequency and second harmonic moved to shorter wavelengths. In general, for the $CdS_{.25}Se_{.75}$ samples, a WG305 was used while for the $CdS_{.50}Se_{.50}$ samples a WG295 was used. Table 4.1 contains the details of the standard experimental conditions used in this work.

The filtered signal was then focused by a Melles Griot 01-CMP-119 uv-grade condenser onto the entrance slit of a monochromator. As with other focussing

optics in this assembly, translation along the optical axis allowed adjustment of the focus. The monochromator employed was the same .10 m ISA single pass grating monochromator mounted on the front of the housing of the Hammamatsu R1564U MCP used in the TCSPC experiments. Due to the lower signal levels, however, .25 mm slits were generally employed. These provided a resolution of approximately 4 nm in the ultraviolet region of the spectrum under investigation. The output was also amplified with the same Hewlitt-Packard 8447D two-stage GHz pre-amplifier.

Although temporal information in fluorescence upconversion experiments is obtained from the stage position and not from the electronics, similar electronics to those employed for TCSPC can be usefully employed. In particular, because the upconverted signal is synchronized to the laser shot, time correlated detection can be used to eliminate dark counts not arriving in a narrow window, 20 ns for the electronics employed. In the present case, the dark count rate was generally less than 50 counts per second, low enough to make such synchronization unnecessary. Nevertheless, the Ortec Model 583 constant fraction discriminator was again employed. The SCA output of the discriminator was passed through a Fast/Slow Logic interface to produce larger pulses which were sent to the counter-timer chips of an IOtech Power 488CT board mounted in a 486 personal computer. This board also conveniently contained an IEEE-488.2 interface, which was used to communicate with the Klinger CC1.1 stepper motor controller.

In brief, the data collection program used a menu-driven interface to allow the user to input where the stage would begin, how many data points to collect, how many steps to take between points, and how long to collect data at each point. This latter point is crucial since the histogram will not accurately represent the fluorescence decay if each data point is not collected for the same amount of time. To ensure that no error arose from this effect, one of the counter-timer chips was

configured to output a single square wave pulse with a resolution of 1 ms or better, the duration being set by the user as the amount of time to collect each data point. This pulse was then used to gate the counter-timer chips configured to count the incoming data pulses. Since data were routinely collected for 1 to 5 seconds, the millisecond resolution of the square pulses was well below the other sources of noise in the data.

As noted previously in Chapter 3, the instrument response function of the fluorescence upconversion spectrometer is essentially just an autocorrelation of the laser pulse. To obtain the response, the dichroic beamsplitter was replaced by a 1 mm glass flat so that the laser light scattered off of the crystal and the remaining laser light in the gating beam could be overlapped. The variable beamsplitter was then adjusted so that the intensity of these two beams was roughly the same at the nonlinear crystal. The crystal was angle tuned to the second harmonic and the stage was scanned. As discussed in the previous chapter, there is a background of roughly half the intensity of the peak intensity of the autocorrelation trace. Nevertheless, the autocorrelation is quite clear on top of a strong background, provided that noise is sufficiently small. An example with the background subtracted is provided in Figure 4.3. The peak of the autocorrelation trace also marks the time zero point of the system, though minor correction must be made for the difference in the optical path lengths through the dichroic and the glass flat.

## 4.5 Numerical Fitting

As already noted in the previous chapter, the instrument response function must often be deconvolved from the data. When this is the case, the deconvolution is generally achieved by using the convolution theorem, which states that the Fourier

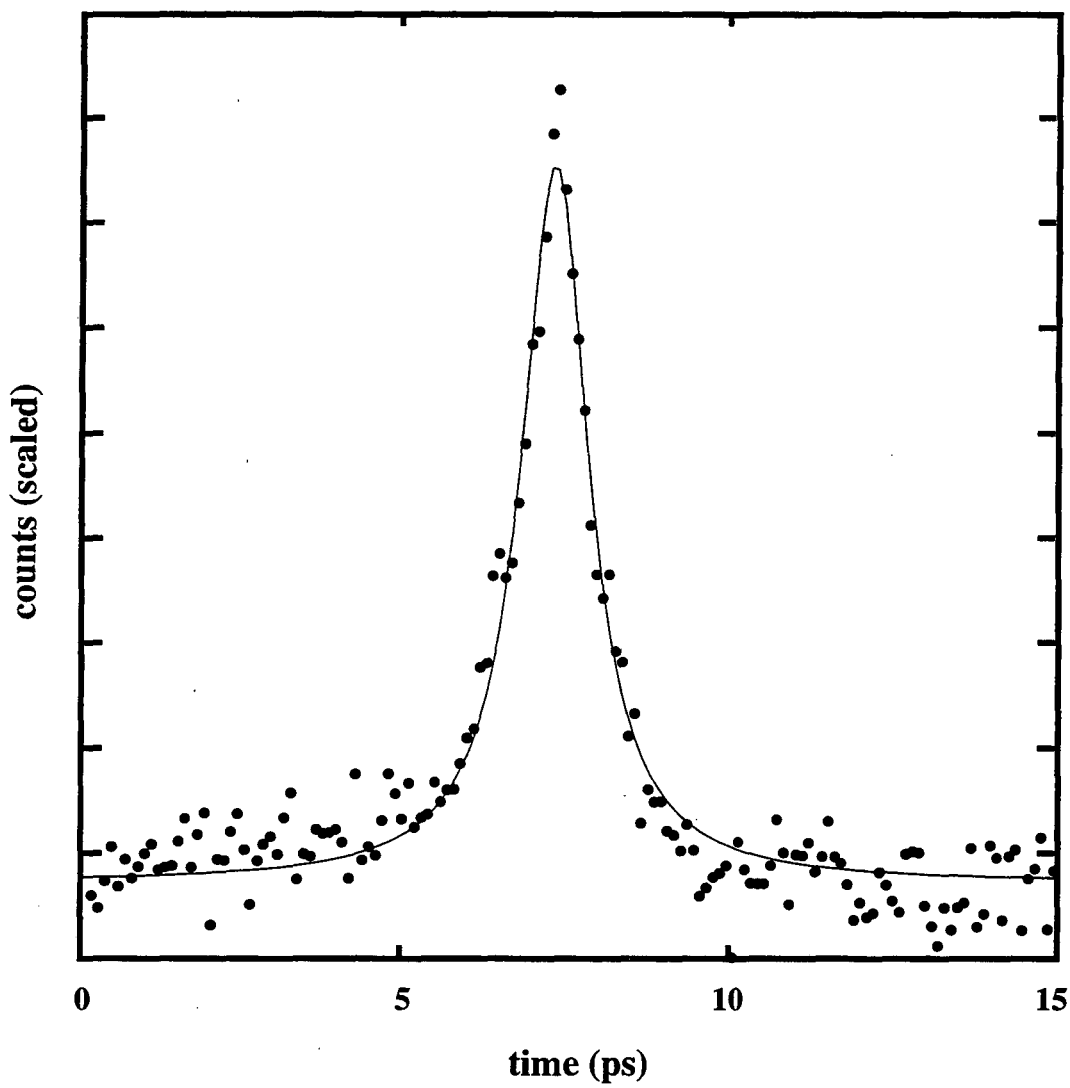


Figure 4.3: Shown above as the data points is a sample of the instrument function taken with the fluorescence upconversion setup described in the text. The solid line is a Lorentzian curve fit to the data using a Levenberg-Marquardt fitting algorithm. This autocorrelation has a FWHM of 1.3 ps.

transform of a convolution of a function $f(t)$ and a response $r(t)$ is equal to the product of the individual Fourier transforms of the functions. In other words, if $S(f)$, $F(f)$, and $R(f)$ are the Fourier transforms of $s(t)$, $f(t)$, and $r(t)$ respectively, where $s(\tau) = f(t) * r(t)$ is the convolution of $f(t)$ and $r(t)$, then $S(f) = F(f)R(f)$. Given $r(t)$ and $s(t)$, one can then obtain $f(t)$ from $F(f) = S(f)/R(f)$.

Because the data are discrete points, discrete Fourier transforms must be employed. One can consider $N$ linearly sampled data points given by $h_k \equiv h(t_k)$ and $t_k \equiv k\Delta$, where $k = 0, 1, 2, \ldots, N-1$ and $t_k$ is the time of the $k$th data point. Here $\Delta$ is the sampling interval. Clearly, with only $N$ values to be analyzed, at most $N$ independent values can be produced. One therefore seeks estimates of the Fourier transform for the discrete frequencies given by $f_n = n/N\Delta$, where $n = -N/2, \ldots, N/2$ and $N$ is assumed even for simplicity. These can be obtained according to

$$H(f_n) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f_n t} dt \approx \sum_{k=0}^{N-1} h_k e^{2\pi i f_n t_k} \Delta = \Delta \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N} \quad . \tag{4.1}$$

The final summation in Eq. 4.1,

$$H_n = \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N} \quad , \tag{4.2}$$

is called the *discrete Fourier transform* of the $N$ points $h_k$.

By a similar analysis, one can show that the discrete *inverse* Fourier transform is given by

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{-2\pi i k n/N} \quad . \tag{4.3}$$

The only differences between Eqs. 4.2 and 4.3 besides the swapping of positions of $H_n$ and $h_k$ are the changing of the sign in the exponential and the division of the summation by $N$. Thus, the same routine that calculates the discrete Fourier

transform can also be used with minimal modification to calculate the inverse as well. In the present work, these calculations were carried out using a fast Fourier transform (FFT) alogrithm based on one originally written by N. Brenner of Lincoln Laboratories [94].

If $S_n$, $\Phi_n$, and $R_n$ are now taken as the discrete Fourier transforms of $s(t)$, $\Phi(t)$, and $r(t)$ respectively as calculated by the FFT algorithm, then $\Phi_n = S_n/R_n$. By application of Eq. 4.3, one can then obtain

$$\Phi(t) = \frac{1}{N}\sum_{n=0}^{N-1} \Phi_n e^{-2\pi ikn/N} = \frac{1}{N}\sum_{n=0}^{N-1} \frac{S_n}{R_n} e^{-2\pi ikn/N} \quad . \tag{4.4}$$

Clearly this procedure will fail if $R_n = 0$ for some value $n$. Furthermore, the results of the deconvolution are particularly sensitive to any noise in $s(t)$ as well as to the accuracy of $r(t)$. For these reasons true numerical deconvolutions can often prove troublesome.

When these difficulties arise, fitting can be achieved by employing a recursive routine instead. First, an initial set of parameters for the assumed decay function is chosen. Then the instrument response function is convolved with the decay function using the FFT algorithm, and the results are compared to the measured data. The results from each such comparison are then used to guide the choice of the parameters of the decay function for the next iteration. This process is repeated until reasonable fits are obtained as determined by the minimization of the squared error. The full fitting program is provided in Appendix B.

As indicated in the program, fitting was possible for a variety of decay functions including single and double exponential decays as well as the stretched

exponential decay. A stretched exponential decay is one of the form

$$\phi(t) \sim \exp^{-(t/\tau)^{\beta}}, \qquad 0 < \beta < 1. \tag{4.5}$$

In addition, the fitting program was able to fit exponential rises, which fitting proved useful for the upconversion data.

# Chapter 5

# Results and Discussion

## 5.1 Results

### 5.1.1 Time Correlated Single Photon Counting

As already indicated in the last chapter, TCSPC was used to measure the fluorescence decays of all five of the samples. For each sample the decays were measured for a variety of wavelengths about the fluorescence maximum. Figure 5.1 shows the results for all five of the samples at their respective emission peaks. While the decays for the pure CdS and CdSe samples are barely above the instrument function (not shown), the decays for the alloys display much greater 1/e decay times. In fact the sample with $x = .50$, the most disordered sample, shows the longest decay. Furthermore, this sample appears to be slightly nonlinear on this logarithmic plot. Such nonlinearity could be the result of a decay function more complicated than a simple exponential decay . However, the complexity of the instrument function alone could be responsible for this minor curvature.

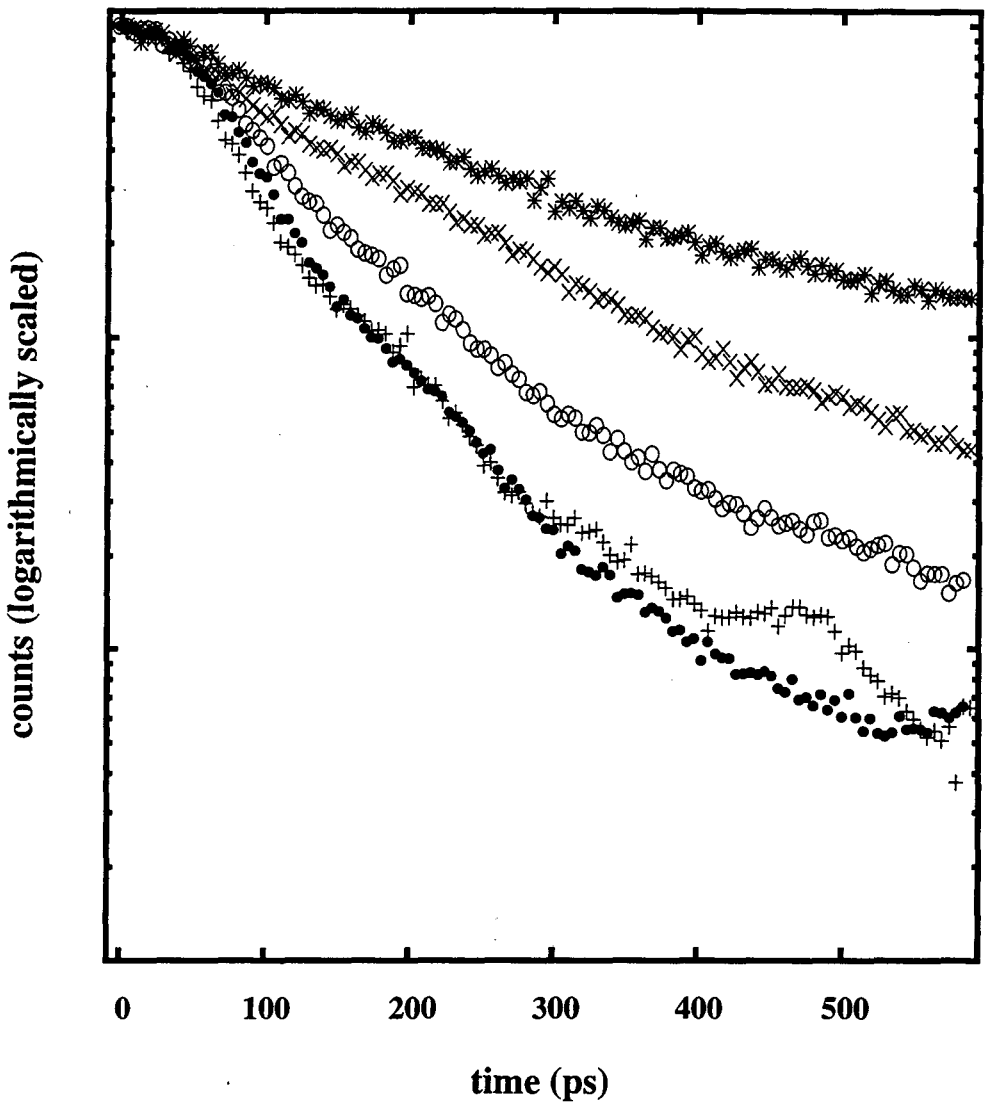


Figure 5.1: Shown above are the fluorescence decays of the five samples at their respective luminescence maxima. The decays correspond to $x = 1$ ($\bullet$), $x = .75$ ($\circ$), $x = .50$ ($*$), $x = .25$ ($\times$), and $x = 0$ ($+$).

The correlation of the longer 1/e times for the alloys compared to the unalloyed CdS and CdSe samples is believed to be due to the presence of the localized states in the valence band as discussed in Chapter 2. Despite the fact that the holes would readily be thermally excited back into the extended states of the band, they nevertheless spend sufficient time in the localized states to show the longer 1/e times in the disordered systems, particularly in the $x = .50$ sample. With the 1/e times of the CdS and CdSe on the order of 100 ps, the primary form of relaxation must be nonradiative since the radiative lifetimes of these materials is approximately 3 ns [58, 95]. However, while the holes are in the localized states, they are not able to recombine with the electrons at the sites of nonradiative recombination. Thus, the 1/e times are extended in the alloys with the increase corresponding roughly to the increase in disorder.

This trend is also useful in understanding the results from the graded semiconductor alloys mentioned in Chapter 1 [16]. The turnaround in that data occurred for $x = .50$ just as the increase in 1/e times in the present data turns around for $x = .50$. In particular, for the CdSe/S samples in which sulfur was diffused into the bulk CdSe to form the gradient, the trend in 1/e times can be explained by the present data. As the carriers diffuse into the bulk material, they pass through the region of greatest disorder in which the greatest number of localized states exist. The holes then spend more time in such states as compared to other regions of the gradient. While in such states, however, the holes cannot reach the sites of nonradiative recombination which dominate the dynamics. The observed decay is therefore lengthened. Beyond this region the number of localized decreases and the decay rate speeds back up. The trend in the CdS/Se samples, on the other hand, is still unilluminated by these results. The explanation of the combination of the reversed band-gap gradient and the self-absorption and re-emission of emitted photons near

the surface [16] is perhaps sufficient to overcome any opposite trends resulting from disorder.

The results for the $x = .50$ sample at the wavelengths 590, 600, 610, 620, 630, and 640 nm are shown in Figure 5.2. Although there appears to be some difference in the decay curves at different wavelengths, the difference is on the order of the noise in the signals. Furthermore, the differences if any are small compared to the instrument function. The hump around 700 ps is an artifact of the instrument function. A similar structure can be observed in Figure 4.2 as well as in the CdSe data of Figure 5.1. These results are typical of all the samples studied. Using TCSPC, variation as a function of emission wavelength within a given sample could not be clearly observed.

In order to obtain more quantitative information from these decays and determine whether or not the apparent curvature is the result of the instrument function or the underlying decay function, the instrument function had to be deconvolved from the measured decay in order to fit the fluorescence decay function. Attempts at directly deconvolving the data proved futile due to the complex nature of the instrument response function, an example of which was shown in Figure 4.2. Thus, the recursive method of fitting had to be employed as described in Section 4.5. Using this method, the unalloyed CdS and CdSe samples both displayed single exponential decays with lifetimes of approximately 30 ps, a value on the extreme edge of our resolution. On the other hand, the alloys could not be satisfactorily fit to a single exponential decay. They all showed stretched tails in the decay which required fitting to a stretched exponential decay (Eq. 4.5). The results of these fits are displayed in Table 5.1.

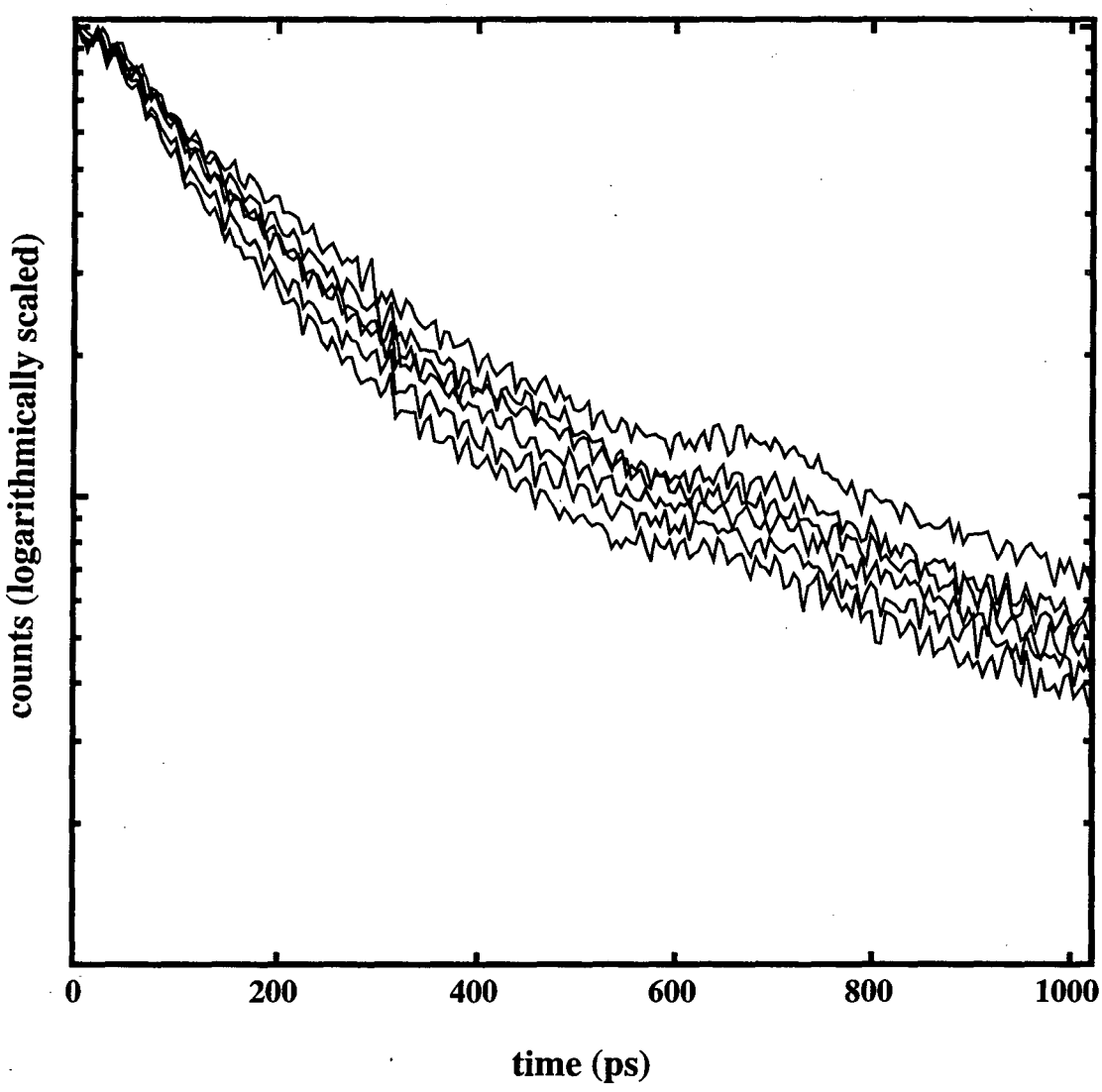


Figure 5.2: Shown above are the fluorescence decays of $CdS_{.50}Se_{.50}$ at the wavelengths 590, 600, 610, 620, 630, and 640 nm.

| sample | CdS | $CdS_{.75}Se_{.25}$ | $CdS_{.50}Se_{.50}$ | $CdS_{.25}Se_{.75}$ | CdSe |
|---|---|---|---|---|---|
| $\tau$ (ps) | $\sim 30$ | $\sim 30$ | $\sim 30$ | $\sim 30$ | $\sim 30$ |
| $\beta$ | 1.0 | .65 | .45 | .55 | 1.0 |

Table 5.1: Shown above are the results of the fits for the data presented in Figure 5.1 for the parameters $\tau$ and $\beta$.

### 5.1.2 Fluorescence Upconversion

As already indicated, upconversion was used to measure the fluorescence decays of the samples with $x = 0.0$, .25, and .50. As in the TCSPC experiments, decays were measured for a variety of wavelengths for each sample in hopes of observing a trend in the decays as a function of wavelength. As can be seen from samples of the data in Figures 5.3, 5.4, and 5.5, there is little variation in the decay within a given sample as a function of wavelength. The initial edge of the fluorescence, however, rises on a timescale of the same order as the instrument function; thus, deconvolution is necessary to examine the rising edge quantitatively.

As discussed in Chapter 3, the autocorrelation of the laser pulse may be taken as the instrument response function of the fluorescence upconversion spectrometer. A sample instrument response has already been shown in Figure 4.3. Deconvolution was then carried out using the mathematical routines described in Section 4.5. Noise in the measured autocorrelation traces of the laser pulse made fitting problematic as the deconvolution would often produce nonsense. Thus, a similar recursive routine of forward convolution followed by fitting was employed. To reduce

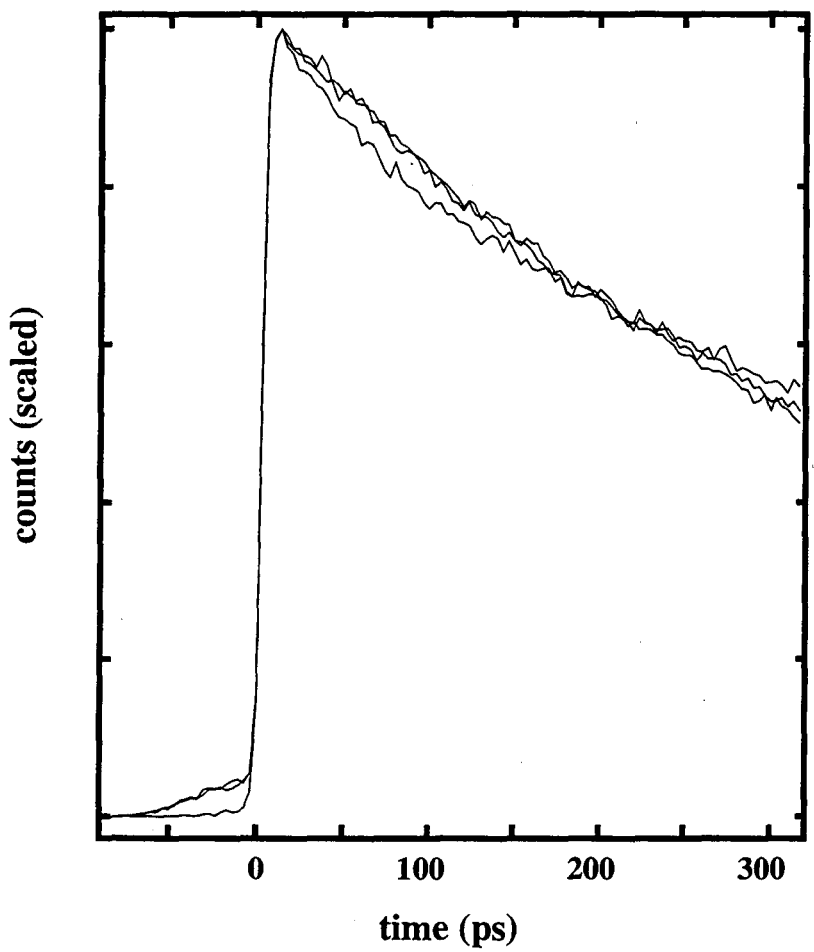


Figure 5.3: Shown above are a series of fluorescence decays of unalloyed CdSe. The decays were measured at wavelengths of 316, 318, and 320 nm.

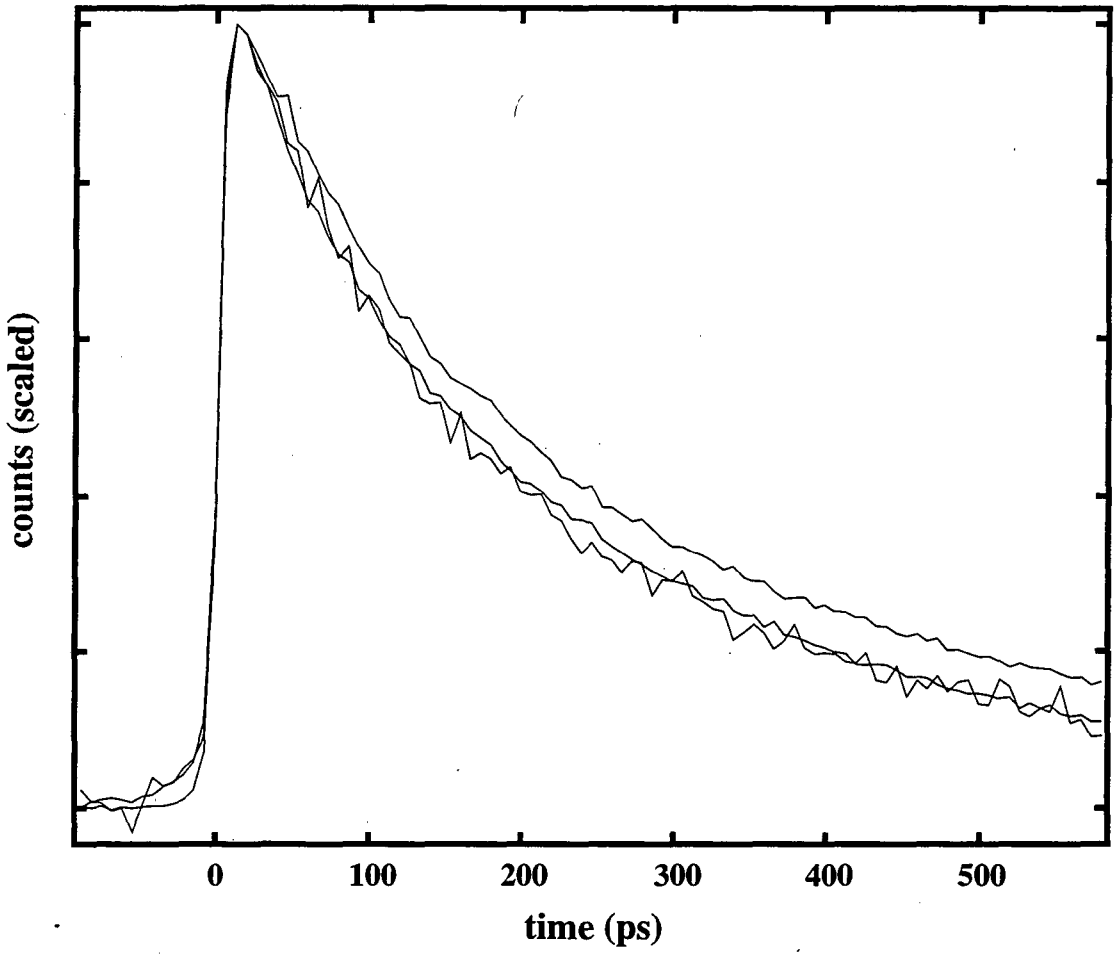


Figure 5.4: Shown above are a series of fluorescence decays of $CdS_{.25}Se_{.75}$. The decays were measured at wavelengths of 305, 307, and 309 nm.

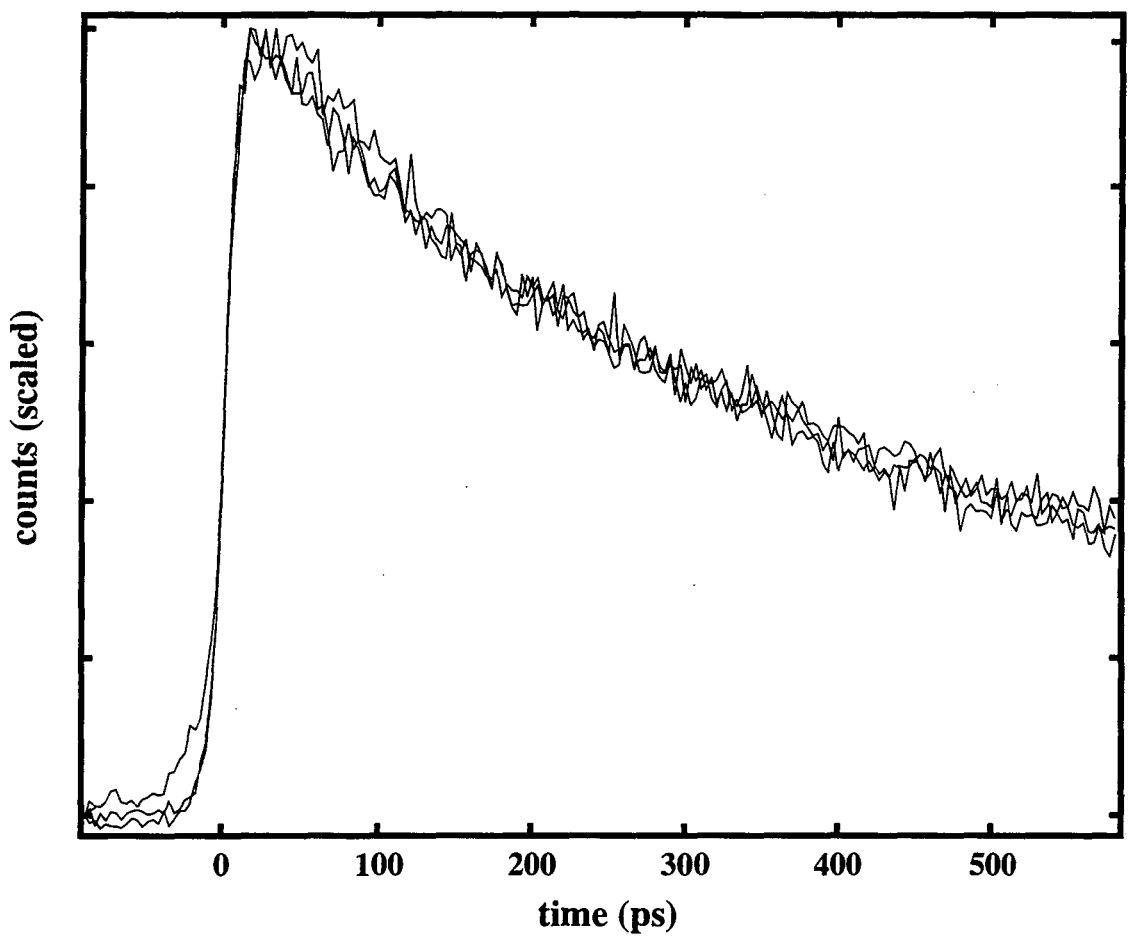


Figure 5.5: Shown above are a series of fluorescence decays of $CdS_{.50}Se_{.50}$. The decays were measured at wavelengths of 287, 289, and 291 nm.

| sample | wavelength (nm) | $\tau$ (ps) | $\beta$ |
|---|---|---|---|
| CdSe | 316 | 385 | 1.0 |
| | 318 | 444 | 1.0 |
| | 320 | 434 | 1.0 |
| $CdS_{.25}Se_{.75}$ | 305 | 143 | .67 |
| | 307 | 182 | .67 |
| | 309 | 160 | .70 |
| $CdS_{.50}Se_{.50}$ | 287 | 400 | .66 |
| | 289 | 433 | .59 |
| | 291 | 404 | .59 |

Table 5.2: Shown above are the results of the fits for data presented in Figures 5.3, 5.4, and 5.5 for the parameters $\tau$ and $\beta$ at the indicated wavelengths.

noise in the convolved fit, the autocorrelation trace was first fit to an analytic function using a Levenberg-Marquardt algorithm [96, 97]. In general the autocorrelation trace was best fit by a Lorentzian, and this shape was used extensively in the fitting. Other functions used to try to fit the autocorrelation trace were the gaussian and the $\text{sech}^2$. The result of this fit was then used as the response function in the convolution with the assumed decay function to produce the convolved fit for comparison to the data.

Using this method of fitting, the data were fit to single exponential, biexponential, and stretched exponential decays both with and without exponential rises. While the unalloyed CdSe sample could be adequately fit with a single exponential decay, the alloys could not. Furthermore, the stretched exponential function fit better than the biexponential despite the fact that it contained one less fitting parameter. The results of the fits to a stretched exponential decay are presented in Table 5.2. Interestingly, the lifetime for the unalloyed CdSe and the $x = .50$ alloy

were both very similar while that for the $x = .25$ alloy was less than half of that value. The reason for these results is unclear; however, as will be discussed in the next section, the value of the lifetime in the samples fit with a stretched exponential decay is related to the statistical distribution of the localized states in a very complicated manner. Thus, the near equivalence of the values for CdSe and the alloy with $x = .50$ is probably just a coincidence. Nevertheless, the observed lifetime for the unalloyed CdSe is in good agreement with previously reported values [58, 98].

In addition, in order to get good fits to the rising edge of the decays, an exponential rise with a rise time of 1 to 2 ps was required, although the exact value was indeterminant due to the resolution. An example of the rising edge of the fluorescence of a sample of CdSe is given in Figure 5.6 along with fits both including and excluding an exponential rise. For the known pulse width, even if there is some error in the value of $t_o$, the position of zero time, the curve calculated assuming no exponential rise still rises far too rapidly to match the rising edge of the measured signal. This can be explained as being due to the amount of time required for the initially excited carriers to relax back to their respective band edges. The measured values of 1 to 2 ps for this intraband relaxation is in good agreement with that measured in other direct band-gap semiconductors [99–105].

## 5.2 Discussion

Although the stretched exponential decay given by Eq. (4.5) may seem a rather odd choice at first, it has been empirically observed in quite a wide variety of relaxations in complex condensed media. As far back as 1847, Rudolf Kohlrausch introduced the stretched exponential to describe the loss of charge from Leyden jars [106]. Later his son Frederick Kohlrausch used the same function in empirical fits

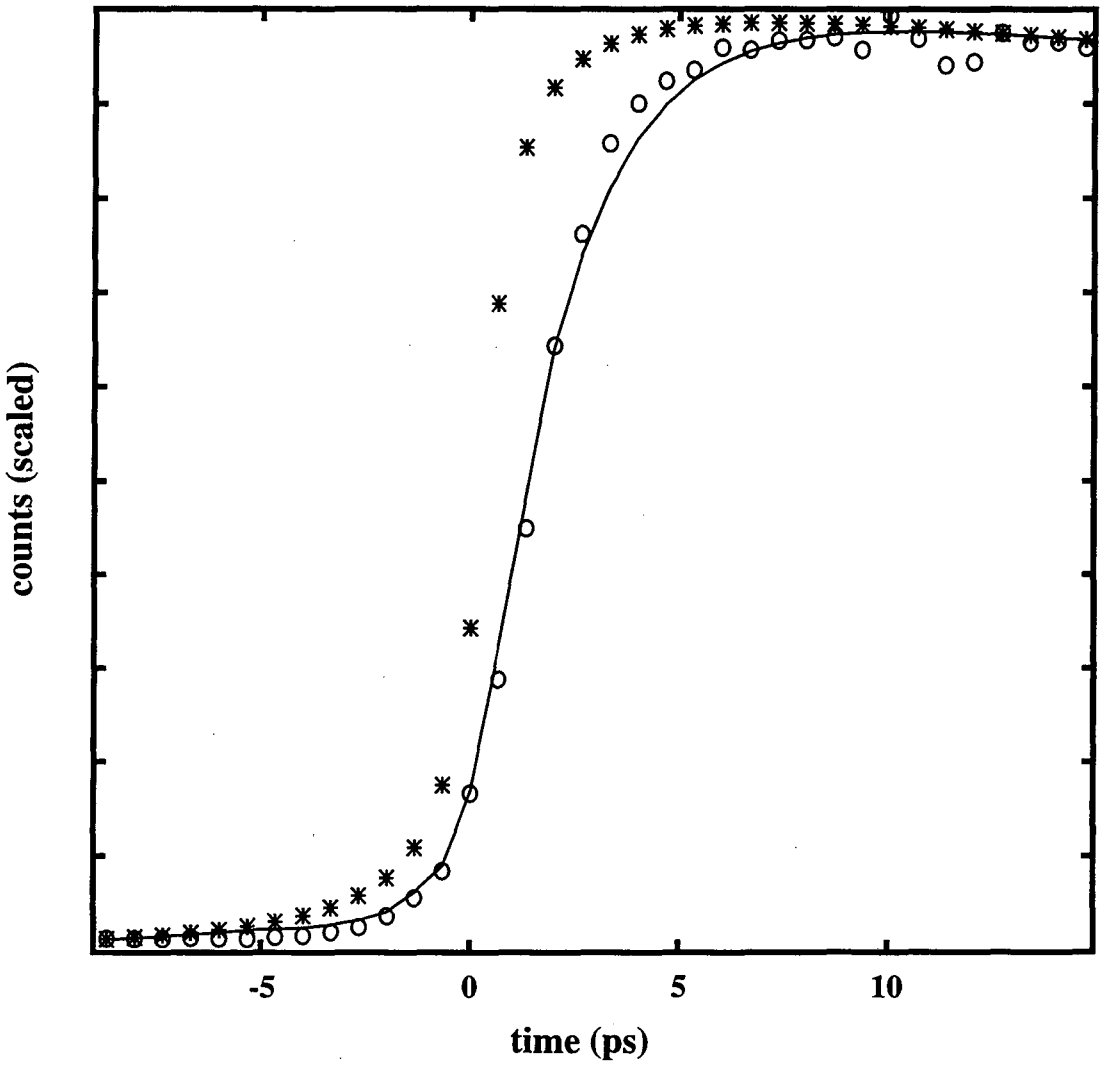


Figure 5.6: Shown above are the rising edge of the fluoresecence emitted by a sample of CdSe (o) as fit with an expression containing an exponential rise (–) verses the same expression without such a rise (*).

of his data on mechanical creep in silk, glass, and rubber [107]. As a result of these seminal uses of the stretched exponential in describing relaxations, the stretched exponential decay (Eq. 4.5) has also been referred to as the Kohlrausch law in the literature. The stretched exponential was reintroduced in 1970 by Williams and Watts to describe dielectric relaxation in polymers [108]. Since that time it has been found to occur in an increasing number of cases [109–111] including the relaxation of remnant magnetization in spin glasses [112], spin relaxation in spin glass alloys [113], electronic structure relaxation in amorphous hydrogenated silicon [114, 115], and luminescence decay in porous glasses [116].

With all of the many natural occurrences of this unusual decay law, the fact that several different physical mechanisms have been advanced to derive Eq. (4.5) should come as no surprise. These include the Förster direct transfer model of many parallel decay channels [117, 118], the hierarchically constrained dynamics model of sequential relaxation [119, 120], and the defect diffusion model [121–123], which has been shown to contain parallel decay channels each of which is inherently sequential [124]. Despite the fundamental physical differences of all of these models, they have been shown to share a common mathematical basis in the form of scale-invariant relaxation rates [125].

In the present case, the stretched exponential can best be described in terms of a parallel decay channel model analagous to the Förster direct transfer model. This model begins with an initially excited donor molecule whose position defines the origin. For decay via direct energy transfer to a defect at site $R_i$ on some given structure, the probability that the donor is still excited at time $t$ is given by the decay function $\Phi_i(t)$ according to

$$\Phi_i(t) = \mathrm{e}^{-tW(R_i)} \quad , \tag{5.1}$$

where $W(R_i)$ is the relaxation rate that depends on the particular donor defect. For a fixed configuration of defects at a collection of sites $R_i$ (excluding the origin where the initially prepared donor is), the net configuration dependent relaxation can be described by

$$\Phi_{R_i} = \prod_i \mathrm{e}^{-tW(R_i)} \quad . \tag{5.2}$$

If one then assumes that the probability for a defect at site $R_i$ is given by $p$, averaging over the possible configurations gives [117, 118]

$$\Phi(t) = \prod_i \left(1 - p + p\mathrm{e}^{-tW(R_i)}\right) \quad . \tag{5.3}$$

If $p \ll 1$, Eq. 5.3 can be approximated by

$$\Phi(t) \approx \exp\left[-p\sum_i \left(1 - \mathrm{e}^{-tW(R_i)}\right)\right] \quad . \tag{5.4}$$

If one then introduces a site density function given by

$$\rho(R) = \sum_i \delta(R - R_i) \quad , \tag{5.5}$$

the summation in Eq. 5.4 can be transformed into an integral, which transformation yields

$$\Phi_i(t) = \exp\left[-p\int \rho(R)\left(1 - \mathrm{e}^{-tW(R_i)}\right) dR\right] \quad . \tag{5.6}$$

Now in the case of Förster transfer, the interaction is taken to be isotropic with a rate $W(R)$ that varies with position according to

$$W(R) = aR^{-s} \quad , \tag{5.7}$$

where $s \geq 6$ Taking the underlying structure to be regular, the density $\rho(R)$ will be constant. One then obtains

$$\Phi(t) = \exp\left[(-t/\tau)^{\beta}\right] \tag{5.8}$$

and

$$\beta = d/s \quad , \tag{5.9}$$

where $\tau$ is a constant with units of time and $d$ is the dimensionality. Equation 5.8 is of course just a stretched exponential of the form of Eq. 4.5.

The primary differences between the present case and that described by the Förster model begin with the fact that while the lifetimes of the carriers have a distribution, that distribution is as a function of energy not of position. Jiang *et. al.* showed that for the states in the tail, the lifetime $\tau(\epsilon)$ varies with the energy $\epsilon$ below the mobility edge according to [126]

$$\tau(\epsilon) \sim \epsilon^2 \quad . \tag{5.10}$$

Furthermore, the energy is distributed one dimensionally. Thus, since the lifetime is related to the rate of relaxation according to $\tau(\epsilon_i) = 1/W(\epsilon_i)$, one can take $s = 2$ and $d = 1$. Then by Eq. 5.8, the decay should appear as a stretched exponential with an exponent of $\beta = .5$. Despite some weaknesses with this analogy, this is indeed what has been observed in the present research.

One of the most important oversights in the above explanation is in the total neglect of nonradiative mechanisms of relaxation. Since the relaxation rates in the pure CdSe were around 400 ps, the radiative transition is certainly not dominant in that case, as CdSe should have a radiative lifetime of approximately 3 ns as noted

in the previous section [58, 95]. However, nonradiative transitions cannot account for the stretched exponential form of the decays in the alloys. A more accurate picture of the decays should therefore include both an account of the nonradiative transitions and the stretched decay via the radiative channels.

That the decays must involve a combination of these two pathways is amply demonstrated by the different decay times in the TCSPC and the upconversion decays of the unalloyed CdSe. While the decays measured in the two experiments were always single exponential, the lifetimes fit to the deconvolved data were markedly different. The TCSPC data was fit by a lifetime of approximately 30 ps while the upconversion data was fit by a lifetime of over 400 ps. The reason for this difference lies in the different excitation wavelengths used in the two experiments. In the TCSPC experiments, the sample was excited with ultraviolet light at 315 nm; in the upconversion experiments, on the other hand, the sample was excited with visible light at 590 nm. The result of this difference is that in the former case the light will only penetrate on the order of .1 $\mu$m as a result of extinction coefficients of about $5 \times 10^5$ cm$^{-1}$ [127, 128]. The visible light, however, will penetrate much more deeply, since the extinction coefficient at that wavelength is an order of magnitude lower; for CdSe the value at 590 nm is $8 \times 10^4$ cm$^{-1}$. Similar results can be observed in the alloys where, for example, the value at 590 nm is $6 \times 10^4$ cm$^{-1}$ for $CdS_{.25}Se_{.75}$ [128]. Because the surface contains many sites of nonradiative recombination, the carriers excited deeper in the bulk will have a decreased likelihood of scattering into these sites. In this case the localized states will play a greater role in the relaxation than in the case of excitation nearer the surface.

Understanding the trends in the decay constants $\tau$ in relation to the lifetimes for the unalloyed samples is not so clear, however. As already noted, the exact relation between the observed constants and the physical parameters of the alloys

is quite complex. That the values as measured via TCSPC are all about the same is undoubtedly due to that fact that the nonradiative mechanisms of relaxation are stongly dominant in those experiments. In the upconversion experiments where this dominance is not as strong, there is a clear difference in the decay constants of the two alloys. Unfortunately, the exact reason for this difference is still unclear.

In comparison to the constant $\tau$, the values of $\beta$, the stretch exponent in Eq. 4.5, are easier to understand. The values for CdSe and CdS were of course $\beta = 1$. CdSe and CdS displayed single exponential decays as expected. On the other hand, the alloys all displayed values of $\beta$ that are less than one as indicated in Tables 5.1 and 5.2. While these are not the value of .5 predicted by the simple model above, they are close considering the simplifying assumptions of that model. In addition, the value is closest for the sample with $x = .5$, the most disordered sample. Finally, in the TCSPC data, the value of $\beta$ for the $x = .25$ alloy is a little closer to .5 than that for the $x = .75$ alloy. This assymetry in the values of $\beta$ implies that the disorder is not truly greatest for $x = .50$ but rather for a value of $x$ slightly less than .5. This assymetry is in agreement with calculations of the disorder reported elsewhere [47].

# Chapter 6

# Summary

In conclusion, fluorescence decays were measured for the series of substitutional semiconductor alloys $CdS_xSe_{1-x}$. These decays were measured by both time correlated single photon counting and fluorescence upconversion. The latter technique had a resolution that was two orders of magnitude better than the former and had a smoother instrument response function. Deconvolution of the data was thereby facilitated. Unfortunately, the lower sensitivity of the method of upconversion made obtaining data for the samples with $x > .5$ impossible by that method with the current setup. The method of TCSPC, though suffering from reduced resolution, did have sufficient sensitivity to obtain data for samples spanning the entire compositional range from $x = 1$ to $x = 0$.

The alloys in both sets of data displayed a stretched exponential decay form as given in Eq. 4.5 with exponents in the general range .5–.7, while the unalloyed CdS and CdSe samples displayed a single exponential decay form. In the upconversion data, a 1–2 ps rise time could also be observed as evidence of the intraband relaxation in all the samples. Although the TCSPC data also displayed the stretched

exponential decay form, the decays exhibited very fast decay constants, approximately 30 ps, due to a strong domination of nonradiative recombination via surface traps which resulted from near surface excitation by ultraviolet light. Furthermore, in addition to the reduced temporal resolution, the instrument function in TCSPC displayed electronic structure that made deconvolution and fitting difficult. Finally, in neither of the techniques was a clear variation in the decay observed as a function of emission wavelength, nor was there any variation in the rise times as a function of wavelength in the decays measured by upconversion.

The observed stretched exponential decay of the fluorescence can be understood as resulting from the existence of localized excitons which exist as localized holes and Coulombically bound electrons. These states can be explained in terms of Anderson localization theory as resulting from potential wells created by local compositional fluctuations and give rise to a tailing of the density of states into the band gap. Theory predicts that this tailing varies with the energy $\epsilon$ below the virtual crystal band edge energy according to $\exp\left[-(\epsilon/\epsilon_o)^{3/2}\right]$, where $\epsilon_o$ is a characteristic energy of approximately 4 meV. Despite the fact that such a small value of $\epsilon_o$ implies that the holes must have spent but a small fraction of their time in any given localized state before being thermally excited back into the extended states, the stretched exponential was observed at room temperature. Furthermore, because these states have radiative lifetimes that are energy dependent, electron–hole pairs have many parallel channels of radiative recombination available. This is the source of the stretched exponential decay. Obtaining a more thorough understanding of the decays, however, requires also taking into account the nonradiative mechanisms of decay, particularly in the TCSPC data.

These results can be used to explain the trend observed in the CdSe/S graded samples studied by Hane and co-workers. As the carriers diffuse down the

combined band gap and concentration gradients, they pass through the $x = .50$ region where the greatest number of localized states exist. In this region the holes therefore spend the greatest proportion of their time in such states as compared to other regions. While in such states, the holes cannot recombine with the electrons as readily. As they move beyond this region, however, the number of localized states again decreases and recombination is enhanced.

# Bibliography

[1] N. I. Vitrikhovskiĭ and I. B. Mizetskaya, *Sov. Phys. Solid State* **1**, 358 (1958).

[2] M. V. Klein, M. D. Sturge, and E. Cohen, *Phys. Rev. B* **25**, 4331 (1982).

[3] M. D. Sturge, E. Cohen, and R. A. Logan, *Phys. Rev. B* **27**, 2362 (1983).

[4] A. Goldman and E.-E. Koch (Eds.), Subvol. a of *Electronic Structure of Solds: Photemission Spectra and Related Data*, Vol. 23 of *Landolt–Börnstein, Numerical Data and Functional Relationships in Science and Technology*, Group III, New Series, Springer–Verlag, Berlin (1989).

[5] H. H. Streckert, J. Tong, and A. B. Ellis, *J. Am. Chem. Soc.* **104**, 581 (1982).

[6] H. H. Streckert, J. Tong, M. K. Carpenter, and A. B. Ellis, *J. Electrochem. Soc.* **129**, 772 (1982).

[7] D. W. Langer, Y. S. Park, and R. N. Euwema, *Phys. Rev.* **152**, 788 (1966).

[8] I. B. Ermolovich, N. I. Vitrikhovskiĭ, I. I. Georgieva, G. I. Matvievskaya, A. M. Pavelets, and M. K. Sheĭnkman, *Sov. Phys. Semicond.* **15**, 519 (1981).

[9] S. Yu. Verbin, S. A. Permogorov, and A. N. Reznitskiĭ, *Sov. Phys. Solid State* **25**, 195 (1983).

[10] Y. S. Park and D. C. Reynolds, *Phys. Rev.* **132**, 2450 (1963).

[11] A. J. Bard, *Science* **207**, 139 (1980).

[12] M. S. Wrighton, *Acc. Chem. Res.* **12**, 303 (1979).

[13] H. F. Taylor, V. N. Smiley, W. E. Martin, and S. S. Pawka, *Phys. Rev. B* **5**, 1467 (1972).

[14] M. K. Carpenter, H. H. Streckert, and A. B. Ellis, *J. Solid State Chem.* **45**, 51 (1982).

[15] M. K. Carpenter and A. B. Ellis, *J. Electroanal. Chem.* **184**, 289 (1985).

[16] J. K. Hane, M. G. Prisant, C. B. Harris, G. J. Meyer, L. K. Leung, and A. B. Ellis, *J. Phys. Chem.* **93**, 7975 (1989).

[17] L. J. Van Ruyven and F. E. Williams, *Am. J. Phys.* **35**, 785 (1967).

[18] J. Bardeen and W. Shockley, *Phys. Rev.* **80**, 72 (1950).

[19] H. Kroemer, *Arch. Elek. Übertragung* **8**, 499 (1954).

[20] R. H. Parmenter, *Phys. Rev.* **97**, 587 (1955).

[21] L. Nordheim, *Ann. Physik* **9**, 607 (1931).

[22] L. Nordheim, *Ann. Physik* **9**, 641 (1931).

[23] T. Muto, *Sci. Papers Inst. Phys. Chem. Research (Tokyo)* **34**, 377 (1938).

[24] Zh. I. Alferov, E. L. Portnoi, and A. A. Rogachev, *Sov. Phys. Semicond.* **2**, 1001 (1969).

[25] J. E. Freund and R. E. Walpole, *Mathematical Statistics*, 4th Ed., Prentice–Hall, Edgewood Cliffs, New Jersey (1987).

[26] P. W. Anderson, *Phys. Rev.* **109**, 1492 (1958).

[27] F. Brouers, *J. Non-cryst. Solids* **4**, 428 (1970).

[28] P. Lloyd, *J. Phys. C: Solid St. Phys.* **2**, 1717 (1969).

[29] S. Lai and M. V. Klein, *Phys. Rev. Lett.* **44**, 1087 (1980).

[30] E. Cohen and M. D. Sturge, *Phys. Rev. B* **25**, 3828 (1982).

[31] G. Noll, U. Siegner, S. G. Shevel, and E. O. Göbel, *Phys. Rev. Lett.* **64**, 792 (1990).

[32] S. Yoshino and M. Okazaki, *J. Phys. Soc. Jpn.* **43**, 415 (1977).

[33] N. F. Mott and E. A. Davis, *Electronic Processes in Non-Crystalline Materials*, Oxford University Press, Oxford, England (1979).

[34] N. F. Mott, *Adv. Phys.* **16**, 49 (1967).

[35] S. Permogorov, A. Reznitsky, S. Verbin, and V. Lysenko, *Solid State Commun.* **47**, 5 (1983).

[36] S. A. Permogorov, A. N. Reznitskiĭ, S. Yu. Verbin, and V. G. Lysenko, *JETP Lett.* **37**, 463 (1983).

[37] P. W. Anderson, *Comments Solid St. Phys.* **2**, 193 (1970).

[38] P. W. Anderson, *Rev. Mod. Phys.* **50**, 141 (1978).

[39] J. M. Ziman, *J. Phys. C: Solid St. Phys.* **2**, 1230 (1969).

[40] D. J. Thouless, *J. Phys. C: Solid St. Phys.* **3**, 1559 (1970).

[41] D. J. Thouless, *J. Phys. C: Solid St. Phys.* **5**, 77 (1972).

[42] R. C. Fletcher, W. A. Yager, A. N. Holden, W. T. Read, and F. R. Merritt, *Phys. Rev.* **94**, 1392 (1954).

[43] G. Feher, R. C. Fletcher, and E. A. Gere, *Phys. Rev.* **100**, 1784 (1955).

[44] J. M. Ziman, *Elements of Advanced Quantum Theory*, Cambridge University Press, Cambridge, England (1969).

[45] M. H. Cohen, H. Fritzsche, and S. R. Ovshinsky, *Phys. Rev. Lett.* **22**, 1065 (1969).

[46] O. Goede, L. John, and D. Hennig, *Phys. Stat. Sol. (b)* **89**, K183 (1978).

[47] O. Goede, D. Hennig, and L. John, *Phys. Stat. Sol. (b)* **96**, 671 (1979).

[48] S. D. Baranovskiĭ and A. L. Éfros, *Sov. Phys. Semicond.* **12**, 1328 (1978).

[49] B. I. Halperin and M. Lax, *Phys. Rev.* **148**, 722 (1966).

[50] D. Ouadjaout and Y. Marfaing, *Phys. Rev. B* **41**, 12096 (1990).

[51] I. M. Lifshitz, *Adv. Phys.* **13**, 483 (1964).

[52] I. M. Lifshitz, *Sov. Phys. JETP* **26**, 462 (1968).

[53] A. Messiah, *Méchanique Quantique*, Dunod, Paris (1962) Vol. 1, p. 305.

[54] K. H. Hellwege (Ed.), *Physics of II–VI and I–VII Compounds, Semimagnetic Semiconductors*, Subvol. b of *Semiconductors*, Vol. 17 of *Landolt–Börnstein,*

*Numerical Data and Functional Relationships in Science and Technology*, Group III, New Series, Springer–Verlag, Berlin (1982).

[55] S. Permogorov, A. Reznitskii, S. Verbin, G. O. Müller, P. Flögel, and M. Nikiforova., *Phys. Stat. Sol. (b)* **113**, 589 (1982).

[56] R. D. Swank, *Phys. Rev* **153**, 844 (1967).

[57] W. A. Harrison, *J. Vacuum Sci. Technol.* **14**, 1016 (1977).

[58] J. A. Kash, A. Ron, and E. Cohen, *Phys. Rev. B* **28**, 6147 (1983).

[59] L. M. Bollinger and G. E. Thomas, *Rev. Sci. Instrum.* **32**, 1044 (1961).

[60] Y. Koechlin, *C. R. Acad. Sci. (Paris)* **252**, 391 (1961).

[61] S. Canonica and U. P. Wild, *Anal. Instrum.* **14**, 331 (1985).

[62] M. C. Chang, S. H. Courtney, A. J. Cross, R. J. Gulotty, J. W. Petrich, and G. R. Fleming, *Anal. Instrum.* **14**, 433 (1985).

[63] L. J. Cline-Love and L. A. Shaver, *Anal. Chem.* **48**, 364A (1976).

[64] W. R. Ware, S. K. Lee, G. J. Brant, and P. P. Chow, *J. Chem. Phys.* **54**, 4729 (1971).

[65] W. R. Ware, P. P. Chow, and S. K. Lee, *Chem. Phys. Lett.* **2**, 356 (1968).

[66] J. W. Petrich, M. C. Chang, D. B. McDonald, and G. R. Fleming, *J. Am. Chem. Soc.* **105**, 3824 (1983).

[67] E. Gudgin, R. Lopez-Delgado, and W. R. Ware, *Can. J. Chem.* **59**, 1037 (1981).

[68] H. P. Good, U. P. Wild, E. Haas, E. Fisher, E. P. Resewitz, and E. Lippert, *Ber. Bunsenges. Phys. Chem.* **86**, 126 (1982).

[69] R. J. Gulotty, L. Mets, R. S. Alberte, and G. R. Fleming, *Photochem. Photobiol.* **41**, 487 (1985).

[70] H. Mahr and M. D. Hirsch, *Opt. Commun.* **13**, 96 (1975).

[71] Y. R. Shen, *The Principles of Nonlinear Optics*, Wiley, New York (1984).

[72] F. Zernike and J. E. Midwinter, *Applied Nonlinear Optics*, Wiley, New York (1973).

[73] N. Bloembergen, *Nonlinear Optics*, W. A. Benjamin, New York (1965).

[74] G. R. Fleming and A. E. Siegman (Eds.), *Ultrafast Phenomena V*, Springer–Verlag, New York (1986).

[75] D. Block, J. Shah, and A. C. Gossard, *Solid State Commun.* **59**, 527 (1986).

[76] W. Rudolph and J. C. Diels in *Ultrafast Phenomena V*, G. R. Fleming and A. E. Siegman (Eds.), Springer–Verlag, New York (1986) pp. 71–74.

[77] E. W. Castner, Jr., M. Maroncelli, and G. R. Fleming, *J. Chem. Phys.* **86**, 1090 (1987).

[78] J. Shah, *IEEE J. Quantum Electron.* **24**, 276 (1988).

[79] D. Eimerl, L. Davis, S. Velsko, E. K. Graham, and A. Zalkin, *J. Appl. Phys.* **62**, 1968 (1987).

[80] I. V. Tomov, R. Fedosejevs, and A. A. Offenberger, *IEEE J. Quant. Electron.* **18**, 2048 (1982).

[81] R. L. Fork, C. H. Brito Cruz, P. C. Becker, and C. V. Shank, *Opt. Lett.* **12**, 483 (1987).

[82] C. H. Brito Cruz, P. C. Becker, R. L. Fork, and C. V. Shank, *Opt. Lett.* **13**, 123 (1988).

[83] M. A. Kahlow, W. Jarzęba, T. P. DuBruil, and P. F. Barbara, *Rev. Sci. Instrum.* **59**, 1098 (1988).

[84] J. Shah, T. C. Damen, B. Deveaud, and D. Block, *Appl. Phys. Lett.* **50**, 1307 (1987).

[85] T. Elsaesser, J. Shah, L. Rota, and P. Lugli, *Phys. Rev. Lett.* **66**, 1757 (1991).

[86] J. Shah, B. Deveaud, T. C. Damen, W. T. Tsang, A. C. Gossard, and P. Lugli, *Phys. Rev. Lett.* **59**, 2222 (1987).

[87] K. Kash and J. Shah, *Appl. Phys. Lett.* **45**, 401 (1984).

[88] B. Deveaud, T. C. Damen, J. Shah, and C. W. Tu, *Appl. Phys. Lett.* **51**, 828 (1987).

[89] M. R. Freeman, D. D. Awschalom, J. M. Hong, and L. L. Chang, *Phys. Rev. Lett.* **64**, 2430 (1990).

[90] M. A. Kahlow, T. J. Kang, and P. F. Barbara, *J. Chem. Phys.* **88**, 2372 (1988).

[91] M. A. Kahlow, W. Jarzęba, T. J. Kang, and P. F. Barbara, *J. Chem. Phys.* **90**, 151 (1989).

[92] A. J. Ruggiero, D. C. Todd, and G. R. Fleming, *J. Am. Chem. Soc.* **112**, 1003 (1990).

[93] J. K. Hane, Doctoral Dissertation, University of California at Berkeley (1995).

[94] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C*, Cambridge University Press, Cambridge (1988) pp. 411–412.

[95] P. Wiesner and U. Heim, *Phys. Rev. B* **11**, 3071 (1975).

[96] K. Levenberg, *Quart. Appl. Math.*, **2**, 164 (1944).

[97] D. W. Marquardt, *J. Soc. Ind. Appl. Math.* **11**, 431 (1963).

[98] T. Daly and M. Mahr, *Solid State Commun.* **25**, 323 (1978).

[99] C. V. Shank, D. H. Auston, E. P. Ippen, and O. Teschke, *Solid State Commun.* **26**, 567 (1978).

[100] R. F. Leheny, J. Shah, R. L. Fork, C. V. Shank, and A. Migus, *Solid State Commun.* **31**, 809 (1979).

[101] C. V. Shank, R. L. Fork, R. F. Leheny, and J. Shah, *Phys. Rev. Lett.* **42**, 112 (1979).

[102] D. von der Linde and R. Lambrich, *Phys. Rev. Lett.* **42**, 1090 (1979).

[103] D. N. Mirlin, I. Ja. Karlik, L. P. Nikitin, I. I. Reshina, and V. F. Sapega, *Solid State Commun.* **37**, 757 (1981).

[104] W. Z. Lin, L. G. Fujimoto, E. P. Ippen, and R. A. Logan, *Appl. Phys. Lett.* **50**, 124 (1987).

[105] R. W. Schoenlein, W. Z. Lin, E. P. Ippen, and J. G. Fujimoto, *Appl. Phys. Lett.* **51**, 1442 (1987).

[106] R. Kohlrausch, *Ann. Phys.* **12**, 393 (1847).

[107] F. Kohlrausch, *Pogg. Ann. Phys.* **119**, 352 (1863).

[108] G. Williams and D. C. Watts, *Trans. Faraday Soc.* **66**, 80 (1970).

[109] A. K. Jonscher, *Nature* **267**, 673 (1977).

[110] K. L. Ngai, *Comments Solid State Phys.* **9**, 127 (1979).

[111] K. L. Ngai, *Comments Solid State Phys.* **9**, 141 (1980).

[112] R. V. Chamberlain, G. Mozurkewich, and R. Orbach, *Phys. Rev. Lett.* **52**, 867 (1984).

[113] F. Mezei and A. P. Murani, *J. Magn. Mater.* **14**, 211 (1979).

[114] J. Kakalios, R. A. Street, and W. B. Jackson, *Phys. Rev. Lett.* **59**, 1037 (1987).

[115] J. Kakalios, *Hopping and Related Phenomena*, World Scientific, Singapore (1990).

[116] U. Even, K. Rademann, J. Jortner, N. Manor, and R. Reisfeld, *Phys. Rev. Lett.* **52**, 2164 (1984).

[117] T. Förster, *Z. Naturforsch Teil A* **4**, 321 (1949).

[118] A. Blumen, *Nuovo Cimento Soc. Ital. Fis. B* **63**, 50 (1981).

[119] R. G. Palmer, D. L. Stein, E. Abrahams, and P. W. Anderson, *Phys. Rev. Lett.* **53**, 958 (1984).

[120] E. W. Knapp, *Phys. Rev. Lett.* **60**, 2386 (1988).

[121] M. F. Shlesinger and E. W. Montroll, *Proc. Natl. Acad. Sci. USA* **81**, 1280 (1984).

[122] M. F. Shlesinger, *Annu. Rev. Phys. Chem.* **39**, 269 (1988).

[123] H. Scher, M. F. Shlesinger, and J. T. Bendler, *Physics Today* **44**, 26 (1991).

[124] J. Klafter and A. Blumen, *Chem. Phys. Lett.* **119**, 377 (1985).

[125] J. Klafter and M. F. Shlesinger, *Proc. Natl. Acad. Sci. USA* **83**, 848 (1986).

[126] H. X. Jiang, L. Q. Zu, and J. Y. Lin, *Phys. Rev. B* **42**, 7284 (1990).

[127] D. Dutton, *Phys. Rev.* **112**, 785 (1958).

[128] R. B. Parsons, W. Wardzynski, and A. D. Yoffe, *Proc. Roy. Soc. A* **262**, 120 (1961).

# Appendix A

# Data Collection Program

## A.1 smain.c

```
/*      Compilation of smain requires the following files:
smain.c, sdata.c, snice.c, splot.c, ssave.c, slook.c, sconst.c, and
ieeeio.c.  The header files in the local directory are sgadd.h and
ieeeio.h.
        This file contains the following subroutines:
main(), readfile(), setupfile(), stage_start(), p_scan(),
stage_fin(), s_step(), interval(), choose_stage(), set_motor(),
dur_set(), read_comment(), display(), storefile(), and bound().
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <graph.h>
#include <malloc.h>
#include <process.h>
#include <conio.h>
#include "sgadd.h"

/*      The following variables are system dependent and are based
on the graphics coordinate system defined in the Microsoft C 5.1
graphics library.  The variables rect_xmin, rect_ymin rect_xmax,
and rect_ymax are dependent on the type of video board, although
```

```
the values in this program will work for EGA and VGA cards,
although the VGA will be running in EGA mode.
*/

long int  rect_xmin = 50, rect_ymin = 50, rect_xmax = 550,
        rect_ymax = 300;
struct rccoord        rcoord;
struct videoconfig  vc;

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

int  channel_dis1 = 0, channel_dis2 = 0;

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

int channel_color1 = 12, channel_color2 = 13, box_color = 11;

/*      The variable points_scan describes the number of points
the stage is going to scan; stage_step describes the number of
stage/motor steps between these points.  nm_per_point, the number of
nanometers per point, is only used when taking spectra.  0 is an
invalid value for points_scan while for stage_step the author´s
birthday has been used as an invalid value (assuming it to be an
unusual value to choose), since the user may wish to move 0 stage
steps per data point.  The invalid value of nm_per_point is 0.
*/

int  points_scan = 0, stage_step = -114, nm_per_point = 0;

/*      The character array dur holds the duration in milliseconds
that data is to be collected for a given point.  The invalid string
is "0".  maxdur is the maximum acceptable value for dur.
*/

char  dur[8];
int  maxdur = 10000;

/*      multi_count is the number of times that the stage should be
scanned; reverse_flag is 1 if data is taken in only one direction of
stage movement and 2 if data is to be taken in both directions.
*/

int  multi_count = 1, reverse_flag;
```

```
/*      When using the stage, stage_beg stores where the stage
should begin taking data; when taking spectra, stage_beg is the
initial wavelength in nanometers of the monochromator. Note that
this variable is a long integer to allow for values that can be
greater than 65000. The invalid data value is -11467. When taking
spectra, stage_end is the final wavelength in nanometers; when using
the stage, it is the ending stage position. The invalid value is
-114. steps_per_nm is a conversion factor for converting motor
steps to nanometers; the invalid value is 0. initpos is the initial
position of the monochromator in nanometers. Its invalid value is
0.
*/

long int  stage_beg = -11467;
int  stage_end = -114, steps_per_nm = 0, initpos = 0;

/*      minwave and maxwave indicate the bounds for stage_beg and
stage_end when taking spectra. These bounds protect the
monochromator from being turned too far in either direction.
*/

int  maxwave = 875, minwave = 250;

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

char  file_name[20], file_num[3], data_file[25];

/*      The array comment stores the user´s comment for a particular
data file. It can store up to 498 characters and ends with the
character "~" to indicate the user has finished the comment. stage
contains the character string for the stage to be used, namely
"short" or "long", though currently only the former is used.
"steve" is the invalid string.
*/

char  comment[500], stage[10];

/*      *data is an array of pointers to arrays where the data are
stored. The arrays are dynamically allocated in the program. The
number of arrays is 4 due to the fact that at a later point in time
someone may want to use the other channels to collect a reference,
to collect the actual laser power as a function of time, and/or to
normalize the data.
```

```
*/

double  *data[4];

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

int  tcolor=7, ecolor=4;

/*      buffer is exactly that, a general-purpose short-term
character buffer.
*/

char  buffer[50];

/*      main() initializes several strings and sets the video mode
before printing out the main menu.  It then reads the input and
determines a course of action.
*/

int  main()
{
char  inputs[3];
int  n, is_set = 0, i, j, k;
char  *input;

input = inputs;
comment[0] = '~';
comment[1] = '\0';
strcpy(stage,"short");
strcpy(dur, "1000");
data_file[0] = '0';
data_file[1] = '\0';
strcpy(file_name, "default");

_setvideomode(_DEFAULTMODE);
do{
    _settextcolor(tcolor);
    _settextposition(9, 25);
    rcoord = _gettextposition();

/*      Output the main menu and get input, calling the appropriate
function for the given input.
*/

    for (n=1; n<10; n++){
        _outtext(main_string(n));
        rcoord.row++;
```

```
        _settextposition(rcoord.row, rcoord.col);
        }

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

    _clearscreen(_GCLEARSCREEN);
    if (n == 1 || n == 2)
        readfile(n);
    else if (n == 3 || n == 4)
        setupfile(n - 2);
    else if (n == 5 || n == 6){
        is_set = 0;
        _settextposition(10, 25);
        rcoord = _gettextposition();
        _settextcolor(ecolor);

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

        if ((stage_step == -114 || stage_beg == -114671 ||
                points_scan == 0 || (strcmp(stage,"short") != 0 &&
                strcmp(stage,"long") != 0)) && n == 5){
            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            is_set = 1;
            _outtext(error_string(1));
            }
        if ((stage_beg == -11467 || stage_end == -114 ||
                nm_per_point == 0) && n == 6){
            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            is_set = 1;
            _outtext(error_string(2));
            }
        if ((atoi(dur)) <= 0){
            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            is_set = 1;
            _outtext(error_string(3));
            }

        /*      At this point, if all the parameters are found to be
        valid, the data collection can begin; otherwise, the user
        must acknowledge the error strings printed above by hitting
        any key.  Note that before moving on to the menu for taking
        data, memory is allocated for the data by first calling
        p_scan(0) or stage_fin(0) for stage/motor data.
```

```
        */

        if (is_set == 0){
            _settextcolor(tcolor);
            if (n == 5)
                p_scan(0);
            else if (n == 6)
                stage_fin(0);
            take_data_menu(n - 4);
            }
        else if (is_set == 1){
            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            _outtext(odd_string(1));

            while (i == 0)
                i = kbhit();
            i = getche();
            _clearscreen(_GCLEARSCREEN);
            }
        }
    else if (n == 7 || n == 8){

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

        for (i=0; i<4; i++)
            if(data[i] != NULL)
                free (data[i]);

        look_data(n - 6);

        /*      If points_scan is defined, the space is
        reallocated.
        */

        if (points_scan != 0){
            for (i=0; i<4; i++){
                data[i]=(double *)malloc(points_scan*sizeof(double));
                if (data[i] == NULL){
                    for (j=0; j<i; j++)
                        free(data[j]);
                    points_scan = 0;
                    _settextposition(10, 15);
                    rcoord = _gettextposition();
                    sprintf(buffer,"%s%i", error_string(11), i);
                    _outtext(buffer);
```

```
                    rcoord.row++;
                    _settextposition(rcoord.row, rcoord.col);
                    _outtext(odd_string(1));

                    while (k == 0)
                        k = kbhit();
                    k = getche();
                    _clearscreen(_GCLEARSCREEN);
                    }
                }
            }
        }

    else if (n == 9){
        for (i=0; i<4; i++)
            if (data[i] != NULL)
                free(data[i]);
        _clearscreen(_GCLEARSCREEN);
        exit(0);
        }
    else if (n < 1 || n > 9){
        _settextposition(7, 15);
        _settextcolor(ecolor);
        _outtext(error_string(5));
        _settextcolor(tcolor);
        }
    } while (n != 9);
exit(-1);
}


/*      readfile() reads in a file that contains a set of parameters
for data collection.  The type of setup file is determined by the
value of m, which is 1 for stage setup files and 2 for spectrum
files.
*/

void readfile(int m)
{
FILE  *stream;
char  inputs[10], inputs2[2], test[4];
char  *input, *input2, *d;
int  i, ch, dr;

/*      First a file name is obtained and checked to determine
whether or not it exists.  If not the user may try again or quit.
*/
```

```
do{
    do{
        _settextposition(10, 20);
        rcoord = _gettextposition();
        _settextcolor(tcolor);
        _outtext(prompt_string(1));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        input = gets(inputs);

        if ((stream = fopen(input, "rb")) == NULL){
            rcoord.row++;
            _settextcolor(ecolor);
            _settextposition(rcoord.row, rcoord.col);
            _outtext(prompt_string(4));

            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            _outtext(prompt_string(5));

            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            _outtext( prompt_string(7));

            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            input2 = gets(inputs2);
            if (inputs2[0] != 'y' && inputs2[0] != 'Y'){
                _settextcolor(tcolor);
                _clearscreen(_GCLEARSCREEN);
                return;
                }
            }
        } while ((inputs2[0] == 'Y' || inputs2[0] == 'y') &&
                                                stream == NULL);

/*      Then the file is checked to ensure that it is of the
appropriate type.  Setup files for data collection with the stage
begin with the three characters 's', 'e', and 'g'; setup files
for spectra begin with 's', 'p', and 'c'.  If the file is of the
wrong type, the user can try again or quit; if it is of the correct
type, the values are read into the appropriate parameters.
*/

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

```
        }
    test[i] = '\0';

    if (strcmp(test, "seg") != 0 && m == 1){
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _settextcolor(ecolor);
        _outtext(error_string(4));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(prompt_string(5));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(prompt_string(7));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        input2 = gets(inputs2);
        if (inputs2[0] != 'y' && inputs2[0] != 'Y'){
            _settextcolor(tcolor);
            _clearscreen(_GCLEARSCREEN);
            return;
            }
        }
    else if (strcmp(test, "spc") != 0 && m == 2){
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _settextcolor(ecolor);
        _outtext(error_string(7));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(prompt_string(5));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(prompt_string(7));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        input2 = gets(inputs2);
        if (inputs2[0] != 'y' && inputs2[0] != 'Y'){
            _settextcolor(tcolor);
            _clearscreen(_GCLEARSCREEN);
            return;
            }
        }
```

```
        else{
            if (m == 1)
                fscanf(stream, " %i %i", &points_scan, &stage_step);
            else if (m == 2)
                fscanf(stream, " %i %i", &nm_per_point, &stage_end);
            fscanf(stream, " %i %li ", &dr, &stage_beg);
            d = itoa(dr, dur, 10);

            if (m == 1)
                stage_end=(int)(stage_beg+(points_scan-1)*stage_step);
            else if (m == 2)
                points_scan=(stage_end-(int)stage_beg)/nm_per_point+1;

            for (i = 0; (i < 498) && (ch=fgetc(stream)) != '~'; i++)
                comment[i] = (char)ch;
            comment[i] = (char)ch;
            i++;
            ch = fgetc(stream);
            comment[i] = (char)ch;

            if (m == 1)
                fscanf(stream, " %s", stage);
            else if (m == 2)
                fscanf(stream, " %i", &steps_per_nm);
            fclose(stream);
            }
        }
    } while (strcmp(test, "seg") != 0 && strcmp(test, "spc") != 0);
_clearscreen(_GCLEARSCREEN);
return;
}


/*      setupfile() allows the user to set the values of parameters
to be used in taking data.  The values can also be saved in a set-up
file.  Whether a stage setup or a spectrum setup is being created is
determined by the value of m, which is 1 or 2 respectively.
*/

void setupfile(int m)
{
char  *input;
char  inputs[3];
int  is_set, n, i;

/*      A menu of options is printed from which the user provides a
choice.  This continues until the user provides the choice to return
to the main menu.  Options for setting parameter values are
accompanied on the right by their values if those values are valid.
```

```
*/

do{
    _settextcolor(tcolor);
    _settextposition(8, 15);
    rcoord = _gettextposition();
    _settextposition(rcoord.row, rcoord.col);
    if (m == 1)
        _outtext(setup_string(1));
    else if (m == 2)
        _outtext(specset_string(1));

    if (stage_beg != -11467){
        rcoord.col = rcoord.col + 40;
        _settextposition(rcoord.row, rcoord.col);
        sprintf(buffer, "%li", stage_beg);
        _settextcolor(ecolor);
        _outtext(buffer);
        _settextcolor(tcolor);
        rcoord.col = rcoord.col - 40;
        }

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    if (m == 1)
        _outtext(setup_string(2));
    else if (m == 2)
        _outtext(specset_string(2));

    if (points_scan != 0 && m == 1 || stage_end != -114 && m == 2){
        rcoord.col = rcoord.col + 40;
        _settextposition(rcoord.row, rcoord.col);
        if (m == 1)
            sprintf(buffer, "%i", points_scan);
        else if (m == 2)
            sprintf(buffer, "%i", stage_end);
        _settextcolor(ecolor);
        _outtext(buffer);
        _settextcolor(tcolor);
        rcoord.col = rcoord.col - 40;
        }

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    if (m == 1)
        _outtext(setup_string(3));
    else if (m == 2)
        _outtext(specset_string(3));
```

```
if (stage_step != -114 && m == 1 || nm_per_point != 0 && m == 2){
    rcoord.col = rcoord.col + 40;
    _settextposition(rcoord.row, rcoord.col);
    if (m == 1)
        sprintf(buffer, "%i", stage_step);
    else if (m == 2)
        sprintf(buffer, "%i", nm_per_point);
    _settextcolor(ecolor);
    _outtext(buffer);
    _settextcolor(tcolor);
    rcoord.col = rcoord.col - 40;
    }

rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
if (m == 1)
    _outtext(setup_string(4));
else if (m == 2)
    _outtext(specset_string(4));

if((strcmp(stage,"short") == 0 || strcmp(stage,"long") == 0 )
                    && m == 1 || steps_per_nm != 0 && m == 2){
    rcoord.col = rcoord.col + 40;
    _settextposition(rcoord.row, rcoord.col);
    if (m == 1)
        sprintf(buffer, stage);
    else if (m == 2)
        sprintf(buffer, "%i", steps_per_nm);
    _settextcolor(ecolor);
    _outtext(buffer);
    _settextcolor(tcolor);
    rcoord.col = rcoord.col - 40;
    }

rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext(setup_string(5));

if (atoi(dur) != 0){
    rcoord.col = rcoord.col + 40;
    _settextposition(rcoord.row, rcoord.col);
    _settextcolor(ecolor);
    _outtext(dur);
    _settextcolor(tcolor);
    rcoord.col = rcoord.col - 40;
    }

for (i=6; i<10; i++){
    rcoord.row++;
```

```
        _settextposition(rcoord.row, rcoord.col);
        _outtext(setup_string(i));
        }

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    input = gets(inputs);
    n = atoi(inputs);

    _clearscreen(_GCLEARSCREEN);
    if (n < 1 || n > 9){
        _settextcolor(ecolor);
        _settextposition(5, 15);
        rcoord = _gettextposition();
        _settextposition(rcoord.row, rcoord.col);
        _outtext(error_string(5));
        }
    else if (n == 1)
        stage_start(m);
    else if (n == 2 && m == 1)
        p_scan(1);
    else if (n == 2 && m == 2)
        stage_fin(1);
    else if (n == 3 && m == 1)
        s_step();
    else if (n == 3 && m == 2)
        interval();
    else if (n == 4 && m == 1)
        choose_stage();
    else if (n == 4 && m == 2)
        set_motor();
    else if (n == 5)
        dur_set();
    else if (n == 6)
        read_comment();
    else if (n == 7)
        display(m);
    else if (n == 8)
        storefile(m);
    } while (n != 9);
_clearscreen(_GCLEARSCREEN);
return;
}


/*      stage_start() gets the value of stage_beg from the user and
validates it.
*/
```

```
void stage_start(int m)
{
char  numbers[10], *result;
int  again;

do{
    again = 0;
    _settextcolor(tcolor);
    _settextposition(10, 15);
    rcoord = _gettextposition();
    if (m == 1)
        _outtext(stage_string(4));
    else if (m == 2)
        _outtext(specset_string(5));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    result = gets(numbers);
    stage_beg = atol(numbers);

    if (stage_beg == 0 && strcmp(numbers, "0") != 0 || m == 2 &&
             (stage_beg>(long)maxwave || stage_beg<(long) minwave)){
        again = 1;
        _clearscreen(_GCLEARSCREEN);
        _settextcolor(ecolor);
        _settextposition(9, 15);
        _outtext(error_string(5));
        }
    } while(again);
_clearscreen(_GCLEARSCREEN);
return;
}


/*      p_scan() gets the value of points_scan from the user and
validates it if n is 1; it calculates stage_end and allocates space
for the points_scan data points if n is 0.  Note that this is only
called for taking data with the stage.  For spectra points_scan is
strictly calculated from other parameters.
*/

void p_scan(int n)
{
int  i, j, k;
char  numbers[10];
char  *result;

for (i=0; i<4; i++)
    if(data[i] != NULL)
```

```
        free (data[i]);

/*      If n == 0, stage_end is calculated and space is allocated
for the data.  Note that this option must not be called until valid
values of points_scan, stage_step, and stage_beg have been set.
*/

if (n == 0){
    stage_end = (int)(stage_beg + (points_scan-1)*stage_step);
    for (i=0; i<4; i++){
        data[i] = (double *)malloc(points_scan * sizeof(double));
        if (data[i] == NULL){
            for (j=0; j<i; j++)
                free(data[j]);
            points_scan = 0;
            _settextposition(10, 15);
            _settextcolor(ecolor);
            rcoord = _gettextposition();
            sprintf(buffer,"%s%i", error_string(11), i);
            _outtext(buffer);

            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            _outtext(odd_string(1));

            while (k == 0)
                k = kbhit();
            k = getche();
            _settextcolor(tcolor);
            _clearscreen(_GCLEARSCREEN);
            return;
            }
        }
    _clearscreen(_GCLEARSCREEN);
    return;
    }

/*      If n != 0, the user is prompted for a value for points_scan,
which is accepted only if it is positive.
*/

do{
    _settextcolor(tcolor);
    _settextposition(10, 15);
    rcoord = _gettextposition();
    _outtext(stage_string(1));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
```

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

    if (points_scan <= 0){
        _clearscreen(_GCLEARSCREEN);
        _settextcolor(ecolor);
        _settextposition(9, 15);
        _outtext(error_string(5));
        }
    } while (points_scan <= 0);

_clearscreen(_GCLEARSCREEN);
return;
}


/*      stage_fin() gets the value of stage_end from the user and
validates it if n==1; it calculates points_scan and allocates space
for the points_scan data points if n==0.  Note that this is only
called for taking data with the motor (i.e. spectra).  For stage
data points_scan is strictly calculated from other parameters.
*/

void stage_fin(int n)
{
int  i, j, k, temp;
char  numbers[10];
char  *result;

for (i=0; i<4; i++)
    if(data[i] != NULL)
        free (data[i]);

/*      If n == 0, then points_scan is calculated and space is
allocated for the data.  This option must not be called until valid
values for stage_end, stage_beg, and nm_per_point have all been set.
*/

if (n == 0){
    if ((temp = (stage_end - (int)stage_beg) % nm_per_point) != 0)
        stage_end += nm_per_point - temp;
    points_scan = (stage_end - (int)stage_beg) / nm_per_point + 1;

    for (i=0; i<4; i++){
        data[i] = (double *)malloc(points_scan * sizeof(double));
        if (data[i] == NULL){
            stage_end = -114;
            for (j=0; j<i; j++)
                free(data[j]);
```

```
            _settextposition(10, 15);
            _settextcolor(ecolor);
            rcoord = _gettextposition();
            sprintf(buffer,"%s%i", error_string(11), i);
            _outtext(buffer);

            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            _outtext(odd_string(1));

            while (k == 0)
                k = kbhit();
            k = getche();
            _settextcolor(tcolor);
            _clearscreen(_GCLEARSCREEN);
            return;
            }
        }
    _clearscreen(_GCLEARSCREEN);
    return;
    }

/*      If n != 0, the user is prompted for a value for stage_end,
which is only accepted if it falls within the bounds set by minwave
and maxwave.
*/

do{
    _settextcolor(tcolor);
    _settextposition(10, 15);
    rcoord = _gettextposition();
    _outtext(specset_string(6));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    result = gets(numbers);
    stage_end = atoi(numbers);

    if (stage_end < minwave || stage_end > maxwave){
        _clearscreen(_GCLEARSCREEN);
        _settextcolor(ecolor);
        _settextposition(9, 15);
        _outtext(error_string(5));
        }
    } while (stage_end < minwave || stage_end > maxwave);

_clearscreen(_GCLEARSCREEN);
return;
```

```
}


/*      s_step() gets the value of stage_step from the user and
validates it.  This is called only for stage set up.
*/

void s_step(void)
{
char  numbers[10];
char  *result;

do{
    _settextcolor(tcolor);
    _settextposition(10, 15);
    rcoord = _gettextposition();
    _outtext(stage_string(2));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    result = gets(numbers);
    stage_step = atoi(numbers);

    if (stage_step == 0 && strcmp(numbers, "0") != 0 ||
                                                stage_step == -114){
        stage_step = -114;
        _clearscreen(_GCLEARSCREEN);
        _settextcolor(ecolor);
        _settextposition(9, 15);
        _outtext(error_string(5));
        }
    } while (stage_step == -114);
_clearscreen(_GCLEARSCREEN);
return;
}


/*      interval() gets the value of nm_per_point from the user and
validates it; thus, it is only called during set up for spectra.
*/

void interval(void)
{
char  numbers[10];
char  *result;

do{
    _settextcolor(tcolor);
    _settextposition(10, 15);
```

```
    rcoord = _gettextposition();
    _outtext(specset_string(7));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    result = gets(numbers);
    nm_per_point = atoi(numbers);

    if (nm_per_point <= 0){
        _clearscreen(_GCLEARSCREEN);
        _settextcolor(ecolor);
        _settextposition(9, 15);
        _outtext(error_string(5));
        }
    } while (nm_per_point <= 0);
_clearscreen(_GCLEARSCREEN);
return;
}


/*      choose_stage() allows the user to choose between the short
and long stages.  However, at present only the short stage is a valid
option.  Obviously this is called only when setting parameters for
stage data.
*/

void choose_stage(void)
{
int  i, ch;
char  *tmp;
char  inputs2[3];
char  *input2;

do{
    _settextcolor(ecolor);
    _settextposition(10, 15);
    rcoord = _gettextposition();

        strcpy(stage, "short");
        _outtext("Currently the only choice for stage is short");

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(odd_string(1));

        while (i = 0)
            i = kbhit();
        i = getche();
```

```
    } while(strcmp(stage, "short") != 0 &&
                                        strcmp(stage, "long") != 0);

return;
}


/*      set_motor() gets the value of steps_per_nm (needed when
taking spectra) from the user and validates it.
*/

void set_motor(void)
{
char  numbers[10];
char  *result;

do{
    _settextcolor(tcolor);
    _settextposition(10, 15);
    rcoord = _gettextposition();
    _outtext(specset_string(8));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    result = gets(numbers);
    steps_per_nm = atoi(numbers);

    if (steps_per_nm == 0){
        _clearscreen(_GCLEARSCREEN);
        _settextcolor(ecolor);
        _settextposition(9, 15);
        _outtext(error_string(5));
        }
    } while (steps_per_nm == 0);
_clearscreen(_GCLEARSCREEN);
return;
}


/*      dur_set() gets the string dur from the user and validates it.
*/

void dur_set(void)
{
int  i, dr;
char  *result;

do{
    _settextcolor(tcolor);
```

```
    _settextposition(10, 15);
    rcoord = _gettextposition();
    _outtext(stage_string(3));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    result = gets(dur);
    dr = atoi(dur);

    if (dr < 0 || dr > maxdur){
        dr = 0;
        _clearscreen(_GCLEARSCREEN);
        _settextcolor(ecolor);
        _settextposition(9, 15);
        _outtext(error_string(5));
        }
    } while (dr == 0);
_clearscreen(_GCLEARSCREEN);
return;
}


/*      read_comment() gets a comment string from the user.  Note
that the user must indicate the end of the comment with a '~'.
*/

void read_comment(void)
{
int  i, ch;

_settextcolor(tcolor);
_settextposition(10, 15);
rcoord = _gettextposition();
_settextposition(rcoord.row, rcoord.col);
_outtext(stage_string(5));

rcoord.row++;
_settextposition(rcoord.row, 0);
for (i = 0; (i < 498) && (ch = getchar()) != '~'; i++)
    comment[i] = (char)ch;
comment[i] = '~';
i++;
ch = getchar();
comment[i] = '\n';
i++;
comment[i] = '\0';
_clearscreen(_GCLEARSCREEN);
return;
}
```

```
/*      display() shows all of the validated parameters both input
and calculated as well as the comment.
*/

void display(int m)
{
int  i = 0;

_settextcolor(tcolor);
_settextposition(7, 15);
rcoord = _gettextposition();
if (m == 1)
    _outtext(set_string(1));
else if (m == 2)
    _outtext(set_string(4));

if (stage_beg != -11467){
    rcoord.col = rcoord.col + 40;
    _settextposition(rcoord.row, rcoord.col);
    sprintf(buffer,"%li", stage_beg);
    _settextcolor(ecolor);
    _outtext(buffer);
    _settextcolor(tcolor);
    rcoord.col = rcoord.col - 40;
    }

if (m == 1){
    if (stage_beg!=-11467 && points_scan!=0 && stage_step!=-114)
        stage_end = (int)(stage_beg + (points_scan-1)*stage_step);
    else
        stage_end = -114;
    }

rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
if (m == 1)
    _outtext(set_string(2));
else if (m == 2)
    _outtext(set_string(5));

if (stage_end != -114){
    rcoord.col = rcoord.col + 40;
    _settextposition(rcoord.row, rcoord.col);
    sprintf(buffer, "%i", stage_end);
    _settextcolor(ecolor);
    _outtext(buffer);
    _settextcolor(tcolor);
```

```
    rcoord.col = rcoord.col - 40;
    }

rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
if (m == 1)
    _outtext(set_string(3));
else if (m == 2)
    _outtext(set_string(6));

if (stage_step != -114 && m == 1 || nm_per_point != 0 && m == 2){
    rcoord.col = rcoord.col + 40;
    _settextposition(rcoord.row, rcoord.col);
    if (m == 1)
        sprintf(buffer, "%i", stage_step);
    else if (m == 2)
        sprintf(buffer, "%i", nm_per_point);
    _settextcolor(ecolor);
    _outtext(buffer);
    _settextcolor(tcolor);
    rcoord.col = rcoord.col - 40;
    }

rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext(set_string(7));

if (atoi(dur) != 0){
    rcoord.col = rcoord.col + 40;
    _settextposition(rcoord.row, rcoord.col);
    _settextcolor(ecolor);
    _outtext(dur);
    _settextcolor(tcolor);
    rcoord.col = rcoord.col - 40;
    }

if (m == 2){
    if (stage_end!=-114 && stage_beg!=-11467 && nm_per_point!=0)
        points_scan = (stage_end-(int)stage_beg) / nm_per_point + 1;
    else
        points_scan = 0;
    }

rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext(set_string(8));

if (points_scan != 0){
    rcoord.col = rcoord.col + 40;
```

```
    _settextposition(rcoord.row, rcoord.col);
    sprintf(buffer, "%i", points_scan);
    _settextcolor(ecolor);
    _outtext(buffer);
    _settextcolor(tcolor);
    rcoord.col = rcoord.col - 40;
    }

rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext(set_string(9));

if (low != -1){
    rcoord.col = rcoord.col + 40;
    _settextposition(rcoord.row, rcoord.col);
    sprintf(buffer,"%i", low);
    _settextcolor(ecolor);
    _outtext(buffer);
    _settextcolor(tcolor);
    rcoord.col = rcoord.col - 40;
    }

rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext(set_string(10));

rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext(set_string(11));
_wrapon(_GWRAPON);

if(comment[0] != '~'){
    rcoord.row++;
    _settextcolor(ecolor);
    _settextposition(rcoord.row, 5);
    _outtext(comment);
    _settextcolor(tcolor);
    }

rcoord = _gettextposition();
rcoord.row++;
_settextposition(rcoord.row, 15);
_outtext(odd_string(1));

while (i == 0)
    i = kbhit();
i = getche();
_clearscreen(_GCLEARSCREEN);
return;
```

```
}

/*      storefile() stores the current set of parameters.  The type
of setup file is determined by the value of m, with 1 indicating a
stage setup file and 2 indicating a spectrum setup file.
*/

void storefile(int m)
{
FILE  *stream;
int  i, dr;
char  inputs[10], inputs2[2];
char  *input, *input2;

/*      A file name is obtained from the user; and, if that file does
not exist, the current set of parameters is written thereto.  If the
file does exist, the user is given the opportunity to overwrite,
quit, or try again.  Note that for stage setup files the first three
characters in the file are ´s´, ´e´, and ´g´; for motor set-up files
they are ´s´, ´p´, and ´c´.
*/

do{
    _settextposition(10, 15);
    rcoord = _gettextposition();
    _outtext(prompt_string(1));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    input = gets(inputs);
    if (strcmp(input, "") == 0)
        return;

    if ((stream = fopen(input, "rb")) != NULL){
        fclose(stream);
        rcoord.row++;
        _settextcolor(ecolor);
        _settextposition(rcoord.row, rcoord.col);
        _outtext(prompt_string(2));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(prompt_string(3));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(prompt_string(6));
```

```
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        input2 = gets(inputs2);
        if (inputs2[0] == 'Q' || inputs2[0] == 'q'){
            _clearscreen(_GCLEARSCREEN);
            return;
            }
        }
    } while ((inputs2[0]=='n' || inputs2[0]=='N') && stream != NULL);

stream = fopen(input, "wb");
if (m == 1){
    fprintf(stream, "seg ");
    fprintf(stream, "%i %i ", points_scan, stage_step);
    }
else if (m == 2){
    fprintf(stream, "spc ");
    fprintf(stream, "%i %i ", nm_per_point, stage_end);
    }
dr = atoi(dur);
fprintf(stream, "%i %li ", dr, stage_beg);
fprintf(stream, "%s ", comment);
if (m == 1)
    fprintf(stream, "%s", stage);
else if (m == 2)
    fprintf(stream, "%i", steps_per_nm);

fclose(stream);
_clearscreen(_GCLEARSCREEN);
return;
}
```

## A.2 sdata.c

```
/*      This file contains the routines take_data_menu(),
set_multi(), getdata(), gp_talk(), and gp_read().
*/

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <graph.h>
#include <conio.h>
#include "sgadd.h"
#include "ieeeio.h"

extern struct rccoord  rcoord;
extern int  tcolor, ecolor;

extern int  points_scan, stage_step;
extern long int  stage_beg;
extern char  dur[], buffer[];
extern int  multi_count, reverse_flag;
extern char  comment[], stage[];
extern double  *data[];


/*      take_data_menu() offers the user a choice of operations
including changing the comment, setting multi-scan parameters,
setting the file name to which to save data, collecting a new
data set, looking at previously collected data, and returning
to the main menu.  The value of the parameter m determines whether
the user intends to take data with the stage (m==1) or the
monochromator (m==2).
*/

void take_data_menu(int m)
{
int  count, i=0, j, k;
char  inputs[3];
char  *input;

do{
    _settextcolor(tcolor);
    _settextposition(10, 25);
    rcoord = _gettextposition();
    if (m == 1)
        _outtext(data_string(0));
    else if (m == 2)
```

```
        _outtext(specset_string(0));

    for (i=1; i<7; i++){
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(data_string(i));
        }
    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    input = gets(inputs);
    count = atoi(inputs);

    _clearscreen(_GCLEARSCREEN);
    if (count < 1 || count > 6){
        _settextposition(7, 20);
        rcoord = _gettextposition();
        _settextcolor(ecolor);
        _outtext(error_string(5));
        }
    else if (count == 1)
        read_comment();
    else if (count == 2)
        set_multi();
    else if (count == 3)
        open_data_file(m);
    else if (count == 4)
        screen(m);
    else if (count == 5){

    /*      In order to look at another file data space must be
    freed for the new data.  Note that this new data file might not
    have the same number of points per scan as the current set-up
    has defined; therefore, even though points_scan is defined in
    look_data(), it is done so statically.  It is thus separate
    from the points_scan defined for the rest of the program.  The
    same holds true for the rest of the set-up parameters.  This
    allows the user to look at different data files without losing
    the current set-up.
    */

        for (i=0; i<4; i++)
            if (data[i] != NULL)
                free (data[i]);

        look_data(m);
        _setvideomode(_DEFAULTMODE);

        if (points_scan != 0 ){
            for (i=0; i<4; i++){
```

```
                data[i] =(double*)malloc(points_scan*sizeof(double));
                if (data[i] == NULL){
                    for (j=0; j<i; j++)
                        free(data[j]);

                    points_scan=0;
                    _settextposition(10, 15);
                    rcoord = _gettextposition();
                    sprintf(buffer,"%s%i", error_string(11), i);
                    _outtext(buffer);

                    rcoord.row++;
                    _settextposition(rcoord.row, rcoord.col);
                    _outtext(odd_string(1));

                    while (k == 0)
                        k = kbhit();
                    k = getche();
                    }
                }
            }
        }
    } while (count != 6);
_clearscreen(_GCLEARSCREEN);
return;
}


/*      set_multi() allows the user to set up the stage or motor to
take multiple scans in one or both directions.  If the user elects
to do so, he inputs the number of scans to be taken, which number
must be even and is assigned to multi_count.  For scans to be taken
in one direction only, reverse_flag==1; otherwise reverse_flag==2.
*/

void set_multi(void)
{
char  inputs[4];
char  *input;

do{
    _settextcolor(tcolor);
    _settextposition(10, 25);
    rcoord = _gettextposition();
    _outtext(multi_string(9));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext(multi_string(10));
```

```
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        input = gets(inputs);
        multi_count = atoi(inputs);

        if (multi_count == 1){
            reverse_flag = 1;
            _clearscreen(_GCLEARSCREEN);
            return;
            }
        else if (multi_count != 2){
            _clearscreen(_GCLEARSCREEN);
            _settextcolor(ecolor);
            _settextposition(9, 25);
            _outtext(multi_string(5));
            }
        } while(multi_count != 2);

    _clearscreen(_GCLEARSCREEN);
    do{
        _settextcolor(tcolor);
        _settextposition(10, 15);
        rcoord = _gettextposition();
        _outtext(multi_string(1));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        input = gets(inputs);
        multi_count = atoi(inputs);

        if (multi_count % 2 != 0 || multi_count <= 0){
            multi_count = 0;
            _clearscreen(_GCLEARSCREEN);
            _settextcolor(ecolor);
            _settextposition(9, 15);
            _outtext(multi_string(2));
            }
        } while (multi_count == 0);

    _clearscreen(_GCLEARSCREEN);
    do{
        _settextposition(10, 20);
        _settextcolor(tcolor);
        rcoord = _gettextposition();
        _outtext(multi_string(3));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
```

```
    _outtext(multi_string(4));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    input = gets(inputs);
    reverse_flag = atoi(inputs);

    if (reverse_flag != 2 && reverse_flag != 1){
        _clearscreen(_GCLEARSCREEN);
        _settextcolor(ecolor);
        _settextposition(9, 20);
        _outtext(multi_string(5));
        }
    } while(reverse_flag != 2 && reverse_flag != 1);
_clearscreen(_GCLEARSCREEN);
return;
}


/*      getdata() is the routine that handles the counter-timers.
On each call it configures them as follows (see also the IO-TECH
manual): CT3 is configured to output a single square pulse of width
dur milliseconds at a delay of 100 ms after being triggered; CT4 is
configured as a counter with input from external source 2 and is
gated by the output from CT3 which is connected externally (see also
notes in sgadd.h); CT5 is configured as a clock with a kHz source and
is zeroed. All three counter-timers are triggered at the same time.
When CT5 indicates that dur + 150 milliseconds have elapsed, the
counter is read. Errors are given by the pointer pe, which is
interpreted by timerror().
*/

void getdata(double dtemp[], int *pe)
{

char  acounts[10], clok[12];
int  i;

*pe = 0;
dtemp[0] = dtemp[1] = dtemp[2] = dtemp[3] = 0;

sprintf(buffer, "CONF:ONES3:SOUR 1KHZ;DEL 100;WIDT %s", dur);
if (gp_talk("timer", buffer) == -1){
    *pe = 1;
    return;
    }

if (gp_talk("timer", "CONF:COUN4:SOUR SOURCE2") == -1){
    *pe = 1;
```

```
    return;
    }
if (gp_talk("timer", "CONF:CLOC5:SOUR 1KHz;TIME 0") == -1){
    *pe = 1;
    return;
    }

if (gp_talk("timer", "START:ONES3;COUN4;CLOC5;:*TRG") == -1){
    *pe = 2;
    return;
    }

do{
    gp_talk("timer", "CAPT:CLOC5;:*TRG");
    gp_talk("timer", "READ:CLOCK5?");
    gp_read("timer", clok);
    } while (atol(clok) <= atol(dur) + 150);

if (gp_talk("timer", "READ:COUN4?") == -1){
    *pe = 4;
    return;
    }
if (gp_read("timer", acounts) == -1){
    *pe = 4;
    return;
    }

dtemp[0] = atof(acounts);

if (gp_talk("timer", "STOP:ONES3;COUN4;CLOC5;:*TRG") == -1){
    *pe = 6;
    return;
    }
}


/*      gp_talk() sends the string *data to *device via the IEEE/GPIB
interface and returns error info.
*/

int gp_talk(char *device, char *data)
{
return(ieeeprtf("output %s;%s\n", device, data));
}


/*      gp_read() reads in ASCII data from *device via the IEEE/GPIB
interface and places it into the string *data.
*/
```

```
int gp_read(char *device, char *data)
{
ieeeprtf("enter %s\n", device);
if (ieeescnf("%s", data) == 0)
    return(-1);
else
    return(0);
}
```

## A.3 snice.c

```
/*      This file contains the following subroutines:
screen(), replot(), top(), timerror(), stage_return(),
gp_stage_wait(), finderr(), and error().
*/

/*      Data is collected through an IOtech 488CT board via the
counter-timer input.  The actual data collection is handled by
the subroutine getdata() in SDATA.C.  At present it does not
use all of the available channels as proveded for in the original
incarnation.  Currently only a single data collection channel is
used and no data processing is required.  However, it may be
necessary to normalize the signal in the future, thereby making
use of the other channels.
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <graph.h>
#include <malloc.h>
#include <conio.h>
#include <time.h>
#include "ieeeio.h"
#include "sgadd.h"

/*      The following variables are used for plotting two of
the possible channels, although at present only the one channel
is actually used.
        The variables max, min, max1, and min1, are the maximum
and minimum of the two data channels that are being displayed to
allow appropriate scaling of the data in the data window.
*/

float  max, min, max1, min1;

/*      The following variables are values for experimental
conditions.
*/

/*      The variable stage_d keeps track of the direction that
the stage/motor should move next.  stage_direction is the string
sent to the stage controller to communicate that direction.
*/

int  stage_d;
```

```
char  *stage_direction;

/*      This a temporary place to store data in the code for
the subroutine getdata().
*/

double        dtemp[4];

/*      scan is the actual number of scans taken.
*/

int  scan;

/*      These variables are associated with communication
subroutines.  device and numdev are variables the GPIB-PC
software associates with the Klinger stage.
*/

char  *device, *numdev;

extern struct rccoord  rcoord;
extern int  channel_dis1, channel_dis2;
extern int  channel_color1, channel_color2, box_color;
extern int  ecolor, tcolor;
extern float  ymin_tick1, ymax_tick1, ymin_tick2, ymax_tick2;
extern long int  rect_xmin, rect_ymin, rect_xmax, rect_ymax;

extern int  points_scan, stage_step, nm_per_point, stage_end,
        steps_per_nm;
extern long int  stage_beg;
extern char  dur[];
extern int  multi_count, reverse_flag, scan;
extern int  initpos, minwave, maxwave;
extern double  *data[];

extern char  stage[], comment[], buffer[];
extern char  file_name[], file_num[], data_file[];

/*      screen() is the main data collection routine.  It controls
the stage and the output to the screen.  The value of m determines
whether the data is being taken with the stage (m==1) or with the
motor (m==2).
*/

void screen(int m)
{

/*      These are temporary variables to hold partial stage movements
in case the desired stage movement is a larger number than a 16 bit
```

```
integer can hold.  This is a constraint due to the stage controller.
This feature is currently unnecessary, but would be necessary for a
longer stage.
*/

unsigned int  step1, step2, step3, step4;

/*      These variables hold the lengths of character strings that
are sent to the stage controller.
*/

int  stage_beg_l, stage_step_length;

/*      These provide space for various strings used in
communication.
*/

char  inputs[3], stage_s[10], *input, present[5];
char  stage_beg_s[10];

/*      point_scan_tak and point_scan_tak2 are the number of
points in the scan that have been taken, although point_scan_tak
starts counting at 1 and point_scan_tak2 starts at 0. i is a
dummy counter for a loop, and e is an error flag from getdata().
flag is a variable to find out if there is a data point out of the
window bounds.  init is used in setting up the monochromator.
*/

int  point_scan_tak, point_scan_tak2, i;
int  e = 0, flag, init;

device = "CC1";
numdev = "07";

/*      If the data to be collected is a spectrum (m==2), the user
inputs the current monochromator position, which is validated as
falling between the minimum and maximum allowable values.
*/

if (m == 2){
    do{
        stage_step = steps_per_nm * nm_per_point;
        _settextcolor(tcolor);
        _settextposition(10, 15);
        rcoord = _gettextposition();
        _outtext(specset_string(9));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
```

```
        input = gets(inputs);
        initpos = atoi(inputs);

        if (initpos < minwave || initpos > maxwave){
            _clearscreen(_GCLEARSCREEN);
            _settextcolor(ecolor);
            _settextposition(9, 15);
            _outtext(error_string(5));
            }
        } while (initpos < minwave || initpos > maxwave);
    _clearscreen(_GCLEARSCREEN);
    init = (int)stage_beg - initpos;
    }

/*      The IEEE board is initialized and the presence of the device
CC1 (the Klinger controller) is verified.
*/

if (ieeeinit() == -1){
    _settextcolor(ecolor);
    _settextposition(10, 20);
    rcoord = _gettextposition();
    _outtext(odd_string(8));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext(odd_string(1));

    while (i = 0)
        i = kbhit();
    i = getche();
    _clearscreen(_GCLEARSCREEN);
    return;
    }
ieeewt("Error Off\n");

ieeeprtf("CheckListener %s\n", numdev);
ieeerd(present);
if (atoi(present) == 0){
    finderr();
    return;
    }

ieeeprtf("Remote %s\n", numdev);

/*      The stage speed/acceleration parameters, travel direction,
and position are set.  See the Klinger CC-1 manual for more details.
*/
```

```
stage_d = 1;

/*      First the parameters for the case of data collected with
a stage are set...
*/

if (m == 1){
    if (gp_talk(device, "R 253") < 0){
        error();
        return;
        }
    if (gp_talk(device, "S 2") < 0){
        error();
        return;
        }
    if (gp_talk(device, "F 20") < 0){
        error();
        return;
        }
    if (gp_talk(device, "A") < 0){
        error();
        return;
        }

    if (stage_beg > 0){
        if (stage_d == 1)
            stage_direction = "+";
        stage_d = -1;
        }
    else{
        stage_direction = "-";
        stage_d = 1;
        }
    if (gp_talk(device, stage_direction) < 0){
        error();
        return;
        }

    if (stage_beg < 65000 && stage_beg > -65000){
        stage_beg_l = sprintf(stage_beg_s, "N %li", labs(stage_beg));
        if (gp_talk(device, stage_beg_s) < 0){
            error();
            return;
            }
        if (gp_talk(device, "G") < 0){
            error();
            return;
            }
        }
```

```
/*      The following "else" is necessary only for the long
long stage, which is not used in the current set-up.
*/

else{
    step1 = abs((int)(stage_beg / 4));
    step2 = abs((int)((stage_beg - step1) / 3));
    step3 = abs((int)((stage_beg - step1 - step2) / 2));
    step4 = abs((int)(stage_beg - step1 - step2 - step3));

    stage_step_length = sprintf(stage_s, "N %u", abs(step1));

    if (gp_talk(device, stage_s) < 0){
        error();
        return;
        }
    if (gp_talk(device, "G") < 0){
        error();
        return;
        }

    gp_stage_wait();
    stage_step_length = sprintf(stage_s, "N %u", abs(step2));
    if (gp_talk(device, stage_s) < 0){
        error();
        return;
        }
    if (gp_talk(device, "G") < 0){
        error();
        return;
        }

    gp_stage_wait();
    stage_step_length = sprintf(stage_s, "N %u", abs(step3));
    if (gp_talk(device, stage_s) < 0){
        error();
        return;
        }
    if (gp_talk(device, "G") < 0){
        error();
        return;
        }

    gp_stage_wait();
    stage_step_length = sprintf(stage_s, "N %u", abs(step4));
    if (gp_talk(device, stage_s) < 0){
        error();
        return;
```

```
            }
        if (gp_talk(device, "G") < 0){
            error();
            return;
            }
        }

    gp_stage_wait();

    if (stage_step > 0){
        if (stage_d == 1)
            stage_direction = "+";
        stage_d = -1;
        }
    else{
        stage_direction = "-";
        stage_d =  1;
        }

    stage_step_length = sprintf(stage_s, "N %i", abs(stage_step));

    if (gp_talk(device, stage_s) < 0){
        error();
        return;
        }
    if (gp_talk(device, stage_direction) < 0){
        error();
        return;
        }
    }

/*      ...then the parameters for spectra, data collected with a
motor, are set.
*/

else if (m == 2){
    if (gp_talk(device, "R 196") < 0){
        error();
        return;
        }
    if (gp_talk(device, "S 2") < 0){
        error();
        return;
        }
    if (gp_talk(device, "F 20") < 0){
        error();
        return;
        }
```

```
    if (init * steps_per_nm > 0)
        stage_direction = "+";
    else
        stage_direction = "-";

    if (gp_talk(device, stage_direction) < 0){
        error();
        return;
        }

    stage_step_length =
                sprintf(stage_s, "N %i", abs(init * steps_per_nm));
    if (gp_talk(device, stage_s) < 0){
        error();
        return;
        }
    if (gp_talk(device, "G") < 0){
        error();
        return;
        }
    gp_stage_wait();
    if (gp_talk(device, "A") < 0){
        error();
        return;
        }
    if (((init = (stage_end - (int)stage_beg)) * steps_per_nm) > 0){
        stage_direction = "+";
        stage_d = -1;
        }
    else{
        stage_direction = "-";
        stage_d = 1;
        }
    if (gp_talk(device, stage_direction) < 0){
        error();
        return;
        }

    stage_step_length = sprintf(stage_s, "N %i", abs(stage_step));
    if (gp_talk(device, stage_s) < 0){
        error();
        return;
        }
    }

/*      The program now enters the main loop.  On each pass of the
loop, data is collected as the stage moves in the "forward" direction
as defined by stage_step, and if reverse_flag==2 in the opposite
direction as well.
```

```
*/

scan = 1;
inputs[0] = 'y';
while (scan <= multi_count){
    point_scan_tak = 1;
    point_scan_tak2 = 0;

/*      The following graphics routines prepare the data window.
*/

    top(scan - 1, m);

    _setcolor(0);
    _rectangle(_GFILLINTERIOR, 0, 50, 620, 330);
    _setcolor(box_color);
    _rectangle(_GBORDER, (short)rect_xmin, (short)rect_ymin,
                (short)rect_xmax, (short)rect_ymax);

/*      The following loop consists of getting a data point, plotting
it, and then advancing the stage to the next position.  The loop can
be exited prematurely by hitting any key on the keyboard.  The user
is then prompted to continue, save and quit, or quit without saving.
*/

    do{
        gp_stage_wait();

        getdata(dtemp,&e);

        if (e != 0){
            if (timerror(e, point_scan_tak, m) == 1){
                return;
                }
            }

        if (gp_talk(device, "G") < 0){
            error();
            return;
            }

/*      The window bounds for the first point are initialized.  Note
that the currently unused second channel is commented out.  Also at
present the minimum bound is set to zero; since the fluorescence data
consists of counts, it is strictly non-negative.
        Then the data are transferred from the temporary variable to
the dynamically allocated data arrays, and the data point is compared
to the previous maximum to see if it supersedes it.  If it does, the
window needs to be redrawn as signified by flag == 1.
```

```
*/

        if (point_scan_tak == 1){
            max = (float)dtemp[channel_dis1] + .01;
                /*max1 = (float)dtemp[channel_dis2] + .01;*/
            min = 0;
                /*min = (float) dtemp[channel_dis1] - .01;*/
                /*min1 = (float) dtemp[channel_dis2] - .01;*/
            rbound1((float)stage_beg, min, (float)stage_end, max);
                /*rbound2(min1, max1);*/
            }

        for (i=0; i!=4; i++){
            *(data[i] + point_scan_tak2) = dtemp[i];
            }
        flag = 0;
        if ((float)dtemp[channel_dis1] > ymax_tick1){
            flag = 1;
            max = (float) dtemp[channel_dis1];
            }

        /*if ((float)dtemp[channel_dis1] < ymin_tick1){
            flag = 1;
            min = (float)dtemp[channel_dis1];
            }
        if ((float)dtemp[channel_dis2] > ymax_tick2){
            flag = 1;
            max1 = (float)dtemp[channel_dis2];
            }
        if ((float)dtemp[channel_dis2] < ymin_tick2){
            flag = 1;
            min1 = (float)dtemp[channel_dis2];
            }*/

/*      If the new data point is out of the window bounds, the window
is redrawn and new bounds are determined.  Then the new point is
plotted.
*/

        if (flag == 1)
            replot(point_scan_tak, m);

        if (m == 1)
            rpoint1((float)(stage_beg + point_scan_tak2*stage_step),
                                    (float)dtemp[channel_dis1]);
        else if (m == 2)
            rpoint1((float)(stage_beg+point_scan_tak2*nm_per_point),
                                        (float)dtemp[channel_dis1]);
        _settextposition(25, 65);
```

```
        _settextcolor(box_color);
        sprintf(buffer, "%i %li  ", point_scan_tak,
                                        (long)dtemp[channel_dis1]);
        _outtext(buffer);

        /*rpoint2((float)point_scan_tak,(float)dtemp[channel_dis2]);*/

        point_scan_tak++;
        point_scan_tak2++;

/*      The keyboard is checked; and, if it was hit, the following
allows the user to continue, quit, or quit while saving the data
collected up to this point.
*/

        if (kbhit() != 0){
            i = getche();
            _setcolor(0);
            _rectangle(_GFILLINTERIOR, 0, 30, 620, 330);

            _setcolor(tcolor);
            _settextposition(10, 15);
            rcoord = _gettextposition();
            _outtext(window_string(9));

            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            _outtext(window_string(10));

            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            _outtext(window_string(11));

            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            _outtext(window_string(12));

            rcoord.row++;
            _settextposition(rcoord.row, rcoord.col);
            input = gets(inputs);

            if (inputs[0] == 'S' || inputs[0] == 's'){
                gp_stage_wait();
                gp_talk(device, "P 0");
                _setvideomode(_DEFAULTMODE);

                open_data_file(m);
                save(scan - 1, point_scan_tak - 1, m);
                return;
```

```
                }
            else if (inputs[0] == 'Y' || inputs[0] == 'y')
                replot(point_scan_tak, m);
            else{
                gp_stage_wait();
                gp_talk(device, "P 0");
                _setvideomode(_DEFAULTMODE);
                return;
                }
            }
        } while (point_scan_tak <= points_scan &&
                        (inputs[0] == 'Y' || inputs[0] == 'y'));

/*      Having collected a data set for one complete pass of the
stage (or motor), the data are saved.  Then the stage (or motor)
is moved back one position to that the of last collected data point.
*/

    _setvideomode(_DEFAULTMODE);
    save(scan - 1, points_scan, m);
    scan++;
    point_scan_tak--;
    point_scan_tak2--;

    if (stage_d == 1){
        stage_direction = "+";
        stage_d = -1;
        }
    else{
        stage_direction = "-";
        stage_d = 1;
        }
    gp_stage_wait();
    if (gp_talk(device, stage_direction) < 0){
        error();
        return;
        }
    if (gp_talk(device, "G") < 0){
        error();
        return;
        }

/*      The next section of code takes data in the "reverse"
direction if reverse_flag == 2.  In all other respects, it is
essentially the same as the previous section.  If reverse_flag == 1,
the stage returns to the starting position.
*/

    if (reverse_flag == 2){
```

```
top(scan - 1, m);

_setcolor(0);
_rectangle(_GFILLINTERIOR, 0, 45, 620, 330);
_setcolor(box_color);
_rectangle(_GBORDER, (short)rect_xmin, (short)rect_ymin,
                    (short)rect_xmax, (short)rect_ymax);

do{
    gp_stage_wait();

    getdata(dtemp,&e);

    if (e != 0){
        if(timerror(e, point_scan_tak, m) == 1){
            return;
            }
        }
    if(gp_talk(device, "G") < 0){
        error();
        return;
        }

    if (point_scan_tak == points_scan){
        max = (float)dtemp[channel_dis1] + .01;
            /*max1 = (float)dtemp[channel_dis2] + .01;*/
        min = 0;
            /*min = (float)dtemp[channel_dis1] - .1;*/
            /*min1 = (float)dtemp[channel_dis2] - .01;*/
        rbound1((float)stage_beg, min, (float)stage_end,max);
            /*rbound2( min1, max1);*/
        }
    for (i=0; i!=4; i++){
        *(data[i] + point_scan_tak2) = dtemp[i];
        }
    flag = 0;
    if ((float)dtemp[channel_dis1] > ymax_tick1){
        flag = 1;
        max = (float)dtemp[channel_dis1];
        }

    /*if ((float)dtemp[channel_dis1] < ymin_tick1){
        flag = 1;
        min = (float)dtemp[channel_dis1];
        }
    if ((float)dtemp[channel_dis2] > ymax_tick2){
        flag = 1;
        max1 = (float)dtemp[channel_dis2];
        }
```

```
if ((float)dtemp[channel_dis2] < ymin_tick2){
    flag = 1;
    min1 = (float)dtemp[channel_dis2];
    }*/

if (flag == 1)
    replot(point_scan_tak, m);

if (m == 1)
    rpoint1((float)(stage_beg+point_scan_tak2*stage_step),
                                (float)dtemp[channel_dis1]);
else if (m == 2)
  rpoint1((float)(stage_beg+point_scan_tak2*nm_per_point),
                                (float)dtemp[channel_dis1]);
_settextposition(25, 65);
_settextcolor(box_color);
sprintf(buffer, "%i %li  ", point_scan_tak,
                                (long)dtemp[channel_dis1]);
_outtext(buffer);

/*rpoint2((float)point_scan_tak,
                                (float)dtemp[channel_dis2]);*/

point_scan_tak--;
point_scan_tak2--;

if (kbhit() != 0){
    i = getche();
    _setcolor(0);
    _rectangle(_GFILLINTERIOR, 0, 30, 620, 330);

    _settextcolor(tcolor);
    _settextposition(10, 15);
    rcoord = _gettextposition();
    _outtext(window_string(9));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext(window_string(10));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext(window_string(11));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext(window_string(12));

    rcoord.row++;
```

```
                _settextposition(rcoord.row, rcoord.col);
                input = gets(inputs);

                if (inputs[0] == 'S' || inputs[0] == 's'){
                    gp_stage_wait();
                    gp_talk(device, "P 0");
                    _setvideomode(_DEFAULTMODE);

                    open_data_file(m);
                    save(scan-1, points_scan-point_scan_tak+1, m);
                    return;
                    }
                else if (inputs[0] == 'Y' || inputs[0] == 'y')
                    replot(point_scan_tak, m);
                else{
                    gp_stage_wait();
                    gp_talk(device, "P 0");
                    _setvideomode(_DEFAULTMODE);
                    return;
                    }
                }
            } while (point_scan_tak>=1 && (inputs[0]=='Y' ||
                                            inputs[0]=='y'));
        _setvideomode(_DEFAULTMODE);
        save(scan - 1, points_scan, m);
        scan++;
        point_scan_tak++;
        point_scan_tak2++;

        gp_stage_wait();
        if (stage_d == 1){
            if(gp_talk(device, "+") < 0){
                error();
                return;
                }
            }
        else{
            if(gp_talk(device, "-") < 0){
                error();
                return;
                }
            }
        if (gp_talk(device, "G") < 0){
            error();
            return;
            }
        }
    else{
        stage_return((long int)point_scan_tak2*(long int)stage_step);
```

```
        if (gp_talk(device, stage_s) < 0){
            error();
            return;
            }
        }

    if (stage_d == 1){
        stage_direction = "+";
        stage_d = -1;
        }
    else{
        stage_direction = "-";
        stage_d = 1;
        }
    gp_stage_wait();
    if (gp_talk(device, stage_direction) < 0){
        error();
        return;
        }
    }
gp_stage_wait();
gp_talk(device, "P 0");
}


/*      replot() redraws the data window and replots all the points
up to but not including the point_scan_tak-"th".
*/

void replot(int point_scan_tak, int m)
{
int i;

_setcolor(0);
_rectangle(_GFILLINTERIOR, 0, 45, 620, 330);
_setcolor(box_color);
_rectangle(_GBORDER, (short)rect_xmin, (short)rect_ymin,
                            (short)rect_xmax, (short)rect_ymax);
rbound1((float)stage_beg, min, (float)stage_end, max);
/*rbound2( min1,  max1);*/

if ((scan%2) == 0 && reverse_flag == 2){
    for (i=points_scan; i>point_scan_tak; i--){
        if (m == 1)
            rpoint1((float)(stage_beg + (i - 1) * stage_step),
                                (float)*(data[channel_dis1]+i-1));
        else if (m == 2)
            rpoint1((float)(stage_beg + (i - 1) * nm_per_point),
                                (float)*(data[channel_dis1]+i-1));
```

```
        /*rpoint2((float)i, (float)*(data[channel_dis2]+i-1));*/
        }
    }
else{
    for (i=1; i<point_scan_tak; i++){
        if (m == 1)
            rpoint1((float)(stage_beg + (i - 1) * stage_step),
                                (float)*(data[channel_dis1]+i-1));
        else if (m == 2)
            rpoint1((float)(stage_beg + (i - 1) * nm_per_point),
                                (float)*(data[channel_dis1]+i-1));
        /*rpoint2((float)i, (float)*(data[channel_dis2]+i-1));*/
        }
    }
}


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

void top(int n, int m)
{
char  *p, file_numt[3];
int  num1, i;
time_t        ltime;

strcpy(data_file, file_name);
if (m == 1)
    strcat(data_file, ".");
else if (m == 2)
    strcat(data_file, ".S");
num1 = atoi(file_num);
num1 = num1 + n;
p = itoa(num1, file_numt, 10);
strcat(data_file, file_numt);

if (_setvideomode(_ERESCOLOR))
    ;
else{
    _setvideomode(_DEFAULTMODE);
    _settextposition(10, 15);
    rcoord = _gettextposition();
    _settextcolor(ecolor);
    _outtext(odd_string(6));

    rcoord.row++;
```

```
    _settextposition(rcoord.row, rcoord.col);
    _outtext(odd_string(1));

    while (i == 0)
        i = kbhit();
    i = getche();
    _clearscreen(_GCLEARSCREEN);
    return;
    }

_settextcolor(tcolor);
_settextposition(1, 1);
rcoord = _gettextposition();
_outtext(window_string(1));
_outtext(data_file);

rcoord.col = rcoord.col + 25;
_settextposition(rcoord.row, rcoord.col);
time(&ltime);
_outtext((ctime(&ltime)));

rcoord.col = rcoord.col + 35;
_settextposition(rcoord.row, rcoord.col);
sprintf(buffer, "Display %i  ", channel_dis1);
_settextcolor(channel_color1);
_outtext(buffer);

_settextcolor(tcolor);
_settextposition(2, 1);
rcoord = _gettextposition();
_outtext(window_string(4));
if (m == 1)
    sprintf(buffer, "%li", stage_beg);
else if (m == 2)
    sprintf(buffer, "%li nm", stage_beg);
_outtext(buffer);

rcoord.col = rcoord.col + 20;
_settextposition(rcoord.row, rcoord.col);
_outtext(window_string(5));
sprintf(buffer, "%i",  points_scan);
_outtext(buffer);

rcoord.col = rcoord.col + 20;
_settextposition(rcoord.row, rcoord.col);
_outtext(window_string(3));
sprintf(buffer, dur);
_outtext(buffer);
```

```
/*rcoord.col = rcoord.col + 20;
_settextposition(rcoord.row, rcoord.col);
sprintf(buffer, "Display %i  ", channel_dis2);
_settextcolor(channel_color2);
_outtext(buffer);
_settextcolor(tcolor);*/

if (multi_count != 1){
    _settextposition(3, 1);
    rcoord = _gettextposition();
    _outtext(window_string(6));

    rcoord.col = rcoord.col + 20;
    _settextposition(rcoord.row, rcoord.col);
    sprintf(buffer, "Scan %i out of %i scans", n + 1, multi_count);
    _outtext(buffer);

    if (reverse_flag == 2){
        rcoord.col = rcoord.col + 30;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(window_string(8));
        }
    }
}


/*      timerror() handles errors involved with the counter-timer
portion of the Power 488CT board.
*/

timerror(e, point_scan_tak, m)
int  e, point_scan_tak, m;
{
char  inputs[3];
char  *input;
int  i;

while (e != 0){
    _setcolor(0);
    _rectangle(_GFILLINTERIOR, 0, 30, 620, 330);
    _setcolor(ecolor);
    _settextposition(10, 15);
    rcoord = _gettextposition();
    if (e == 1)
      _outtext("An error was encountered during TIMER configuration");
    else if (e == 2)
        _outtext("An error occurred while starting counter-timers");
    else if (e == 4)
        _outtext("An error occurred while reading the data");
```

```
    else if (e == 6)
        _outtext("An error occurred while stopping counter-timers");

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext(odd_string(5));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    input = gets(inputs);

    if (inputs[0] != 'Y' && inputs[0] != 'y'){

/*      If the user wants to quit, the stage should be returned to
the place where it began.
*/

        gp_stage_wait();
        gp_talk(device, "P 0");

        return(1);
        }

/*      Here data collection continues and there are two cases:
either the data window doesn't need to be redrawn because no data
has been taken yet, or the window must be redrawn because data
already exist.
*/

    else if ((inputs[0]=='Y'||inputs[0]=='y') && point_scan_tak==1)
        getdata(dtemp,&e);
    else if ((inputs[0]=='Y'||inputs[0]=='y') && point_scan_tak!=1){
        getdata(dtemp,&e);
        replot(point_scan_tak, m);
        }
    }
return(0);
}


void stage_return(back)
long int  back;
{
char  stage_s[10];
int  stage_step_length;
unsigned int  step1, step2, step3, step4;

gp_stage_wait();
if ((back) > 0){
```

```
    stage_direction = "-";
    gp_talk(device, stage_direction);
    }
else{
    stage_direction = "+";
    gp_talk(device, stage_direction);
    }

if (labs(back) < 65000){
    stage_step_length = sprintf(stage_s, "N %li", labs(back));
    gp_talk(device, stage_s);
    gp_talk(device, "G");
    }

/*      As in screen() the following is only needed for long stages.
*/

else{
    step1 = abs((int)(stage_beg / 4));
    step2 = abs((int)((stage_beg - step1) / 3));
    step3 = abs((int)((stage_beg - step1 - step2) / 2));
    step4 = abs((int)(stage_beg - step1 - step2 - step3));

    stage_step_length = sprintf(stage_s, "N %u", step1);
    gp_talk(device, stage_s);
    gp_talk(device, "G");

    gp_stage_wait();

    stage_step_length = sprintf(stage_s, "N %u", step2);
    gp_talk(device, stage_s);
    gp_talk(device, "G");

    gp_stage_wait();

    stage_step_length = sprintf(stage_s, "N %u", step3);
    gp_talk(device, stage_s);
    gp_talk(device, "G");

    gp_stage_wait();

    stage_step_length = sprintf(stage_s, "N %u", step4);
    gp_talk(device, stage_s);
    gp_talk(device, "G");
    }
}


/*      gp_stage_wait() is nominally written to cause the program
```

```
to wait while the stage controller is busy; however, I do not believe
that it works correctly.  After the time out interval set up
in the driver´s install program, the driver itself crashes.
*/

gp_stage_wait()
{
int  dummy = 0;
int  ierrno = 0, i;
char  errnum[4], buffer[70];

do{
    gp_talk(device, stage_direction);
    ieeewt("status\n");
    ieeerd(buffer);
    for(i=0; i<3; i++)
        errnum[i] = buffer[13 + i];
    errnum[i] = ´\0´;
    ierrno = atoi(errnum);
    } while(ierrno != 0);
}


/*      This routine indicates a CheckListener failure.
*/

void finderr()
{
int  i=0;

_settextcolor(ecolor);
_settextposition(10, 15);
rcoord = _gettextposition();
_outtext(odd_string(2));

rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext(odd_string(3));

rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext(odd_string(1));

while (i == 0)
    i = kbhit();
i = getche();
_clearscreen(_GCLEARSCREEN);
return;
}
```

```
/*      This error checking routine will upon the occurrence of
an error during communication with the Klinger controller check
the error number and the error message to determine the exact
cause of the error condition.
*/

void error()
{
char  buffer[70];
int  i;
char  errmes[41], errnum[4];

_clearscreen(_GCLEARSCREEN);
_settextcolor(4);
_settextposition(10, 15);
rcoord = _gettextposition();
_outtext(odd_string(4));

ieeewt("status\n");
ieeerd(buffer);
for(i=0; i<3; i++)
    errnum[i] = buffer[13+i];
errnum[i] = '\0';
for(i=0; i<40; i++)
    errmes[i] = buffer[26+i];
errmes[i] = '\0';
sprintf(buffer, "Error %s: %s", errnum, errmes);

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

rcoord.row+=2;
_settextposition(rcoord.row, rcoord.col);
_outtext(odd_string(1));

while (i == 0)
    i = kbhit();
i = getche();
_clearscreen(_GCLEARSCREEN);
return;
}
```

## A.4 splot.c

```
/*      This file contains the following subroutines:
rbound1(), rtickxy1(), rpoint1(), text_point(), rbound2(),
rtickxy2(), rpoint2(), csave(), rpointc(), and cend().
*/

#include <graph.h>
#include <conio.h>
#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include "sgadd.h"

/*      cimage stores the portion of a plot occupied by the cursor
so that it may be restored when the cursor is moved elsewhere.
ixold and iyold store the logical coordinates where this image is
to be put.
*/

static char far  *cimage;
int  ixold, iyold;

/*      xycoord stores physical pixel coordinates.
*/

struct xycoord        xycoord;

/*      rect_xdiff and rect_ydiff store the full width and height
of the data window in terms of pixels.  xdiff, ydiff1, and ydiff2
store the full spread of the data in the appropriate directions in
terms of the input units, in the present case wavelength or time in
the x direction and counts in the y direction.  xmin, ymin1, and
ymin2 are the minima of the data in the appropriate directions.
xs, ys1, and ys2 are the scale factors used to convert from input
units to pixels.
*/

float  rect_xdiff, rect_ydiff;
float  xdiff, ydiff1, ydiff2;
float  xmin, ymin1, ymin2;
float  xs, ys1, ys2;

/*      The variables xmin_tick, xmax_tick, ymin_tick1, etc., are
the real minima and maxima for the data window.  These ensure that
the tick marks on the axes are reasonable and all the data points
can be seen.
*/
```

```
float  xmin_tick, xmax_tick;
float  ymin_tick1, ymax_tick1, ymin_tick2, ymax_tick2;

extern long int  rect_xmin, rect_ymin, rect_xmax, rect_ymax;

/*      rbound1() sets up the boundaries of the data in terms of the
spread of the values as passed down through the arguments; it thus
begins the mapping of data to the screen´s coordinates.  rbound1()
does this for the first set of data points, both x and y coordinates.
*/

void rbound1(float a, float b, float xmax, float ymax1)
{
xmin = a;
ymin1 = b;
xdiff = xmax - xmin;
ydiff1 = ymax1 - ymin1;
rect_xdiff = (float)(rect_xmax - rect_xmin);
rect_ydiff = (float)(rect_ymax - rect_ymin);
rtickxy1();
}


/*      rtickxy1() finishes the mapping (but not the plotting) of
data and plots and labels the tick marks (but not the axes) for the
first set of data points, both x and y coordinates.
*/

void rtickxy1(void)
{
extern int  box_color, channel_color1;
extern char  buffer[];

float  temp, temp2, temp_tick;

xmin_tick = floor(xmin);
xmax_tick = ceil(xmin + xdiff);
temp_tick = xmin_tick;

xdiff = xmax_tick - xmin_tick;
temp = (xdiff) / 5;
xs = rect_xdiff / xdiff;
xmin = xmin_tick;

_settextcolor(box_color);
_setcolor(box_color);
while (temp_tick <= xmax_tick){
    temp2 = (temp_tick - xmin_tick) * xs + rect_xmin;
```

```
    _moveto((int)temp2, (int)rect_ymax);
    _lineto((int)temp2, (int)rect_ymax - 10);

    text_point((long int)temp2, rect_ymax + 10);

    if (temp > (float)1.)
        sprintf(buffer, "%.0f", temp_tick);
    else if (temp > (float) .1)
        sprintf(buffer, "%.1f", temp_tick);
    else if (temp < (float) .1)
        sprintf(buffer, "%.2f", temp_tick);

    _outtext(buffer);
    temp_tick = temp_tick + temp;
    }

ymin_tick1 = floor(ymin1);
ymax_tick1 = ceil(ymin1 + ydiff1);
temp_tick = ymin_tick1;

ydiff1 = ymax_tick1 - ymin_tick1;
temp = ydiff1 / 5;
ys1 = rect_ydiff / ydiff1;
ymin1 = ymin_tick1;

_setcolor(channel_color1);
_settextcolor(channel_color1);
while (temp_tick <= ymax_tick1){
    temp2 = rect_ymax - (temp_tick - ymin_tick1) * ys1;
    _moveto((int)rect_xmin, (int)temp2);
    _lineto((int)rect_xmin + 10, (int)temp2);

    text_point(rect_xmin - 30, (long int)temp2);

    if (temp > (float)1.)
        sprintf(buffer, "%.0f", temp_tick);
    else if (temp > (float) .1)
        sprintf(buffer, "%.1f", temp_tick);
    else if (temp > (float) .01)
        sprintf(buffer, "%.2f", temp_tick);
    else if (temp < (float) .01)
        sprintf(buffer, "%.3f", temp_tick);

    _outtext(buffer);
    temp_tick = temp_tick + temp;
    }
}
```

```
/*      rpoint1() plots one point per call using the mapping set up
in rbound1() and rtickxy1().
*/

void rpoint1(float rx0, float ry0)
{
extern int  channel_color1;

int  ix, iy;

ix = (int)((rx0 - xmin_tick) * xs + rect_xmin);
iy = (int)(rect_ymax - (ry0 - ymin_tick1) * ys1);
_setcolor(channel_color1);
_setpixel(ix, iy);
}


/*      text_point() moves the text coordinate to the location where
a label for a tick mark should be.
*/

void text_point(ix, iy)
long int  ix, iy;
{
extern struct videoconfig  vc;

_getvideoconfig(&vc);

xycoord = _getphyscoord((int)ix, (int)iy);
ix = (long int)((float)xycoord.xcoord /
        (float)vc.numxpixels * ((float)vc.numtextcols));
iy = (long int)(1 + (float)xycoord.ycoord /
        (float)vc.numypixels * ((float)vc.numtextrows));
_settextposition((int)iy, (int)ix);
}


/*      rbound2() is the same as rbound1() except the mapping is for
the second set of data points (currently not used).  Note that the
x-axis mapping is already defined from rbound1().
*/

void rbound2(float b, float ymax2)
{
ymin2 = b;
ydiff2 = ymax2 - ymin2;
rtickxy2();
}
```

```
/*      rtickxy2() is for channel 2 as rtickxy1() is for channel 1
except that the x-axis tick marks have already been drawn by
rtickxy1().
*/

void rtickxy2(void)
{
extern int  channel_color2;
extern char  buffer[];

float  temp, temp2, temp_tick;

ymin_tick2 = floor(ymin2);
ymax_tick2 = ceil(ymin2 + ydiff2);
temp_tick = ymin_tick2;

ydiff2 = ymax_tick2 - ymin_tick2;
temp = ydiff2 / 5;
ys2 = rect_ydiff / ydiff2;
ymin2 = ymin_tick2;

_setcolor(channel_color2);
_settextcolor(channel_color2);
while (temp_tick <= ymax_tick2){
    temp2 = rect_ymax - (temp_tick - ymin_tick2) * ys2;
    _moveto((int)rect_xmax, (int)temp2);
    _lineto((int)rect_xmax - 10, (int)temp2);
    text_point(rect_xmax + 20, (long int)temp2);

    if (temp > (float)1.)
        sprintf(buffer, "%.0f", temp_tick);
    else if (temp > (float) .1)
        sprintf(buffer, "%.1f", temp_tick);
    else if (temp > (float).01)
        sprintf(buffer, "%.2f", temp_tick);
    else if (temp < (float).01)
        sprintf(buffer, "%.3f", temp_tick);

    _outtext(buffer);
    temp_tick = temp_tick + temp;
    }
}


/*      rpoint2() is the channel 2 analog of rpoint1() but maps to
the screen according to the mapping set up in rbound2() and
rtickxy2().
*/
```

```
void rpoint2(float rx0, float ry0)
{
extern int  channel_color2;

int  ix, iy;

ix = (int)((rx0 - xmin_tick) * xs + rect_xmin);
iy = (int)(rect_ymax - (ry0 - ymin_tick2) * ys2);
_setcolor(channel_color2);
_setpixel(ix, iy);
}


/*      csave() initializes cursor movements by saving space
for the original image and then writes the cursor image.
*/

void csave(float rx0, float ry0)
{
extern int  ecolor;

int  ix, iy, i;

ix = (int)((rx0 - xmin_tick) * xs + rect_xmin);
iy = (int)(rect_ymax - (ry0 - ymin_tick1) * ys1);
ixold = ix;
iyold = iy;
cimage = _fmalloc((unsigned int)_imagesize(ix-5, iy-5, ix+5, iy+5));

if (cimage == (char far *)NULL){
    _settextposition(24, 1);
    _settextcolor(ecolor);
    printf("     _fmalloc failed     \n");
    printf("     %s     ", odd_string(1));
    while (i == 0)
        i = kbhit();
    i = getche();
    return;
    }

_getimage(ix - 5, iy - 5, ix + 5, iy + 5, cimage);

_setlinestyle(0xaaaa);
}


/*      rpointc() plots the cursor at the new position.
*/
```

```
void rpointc(float rx0, float ry0)
{
extern int  box_color;
extern char  buffer[];

int  ix, iy;

_putimage(ixold - 5, iyold - 5, cimage, _GPSET);

ix = (int)((rx0 - xmin_tick) * xs + rect_xmin);
iy = (int)(rect_ymax - (ry0 - ymin_tick1) * ys1);
_getimage(ix - 5, iy - 5, ix + 5, iy + 5, cimage);

_setcolor(15);
_moveto(ix, iy + 5);
_lineto(ix, iy - 5);
_moveto(ix - 5, iy);
_lineto(ix + 5, iy);

ixold = ix;
iyold = iy;

_setcolor(0);
_rectangle(_GFILLINTERIOR, 520, 330, 640, 350);

_settextposition(25, 65);
_settextcolor(box_color);
sprintf(buffer, "%i %li", (int)rx0, (long)ry0);
_outtext(buffer);
}


/*      cend() frees up the space used for cursor movements.
*/

void cend()
{
_ffree(cimage);
}
```

## A.5 ssave.c

```
/*      This file contains the routines open_data_file() and save().
*/

#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <graph.h>
#include <conio.h>
#include <math.h>
#include "sgadd.h"
#include <string.h>

extern struct rccoord  rcoord;
extern int  tcolor, ecolor;
extern int  channel_dis1, channel_dis2;
extern int  channel_color1, channel_color2, box_color;

extern int  points_scan, stage_step, nm_per_point, stage_end,
        steps_per_nm;
extern int  multi_count, reverse_flag;
extern long int  stage_beg;
extern char  dur[];
extern double  *data[];

extern char  file_name[], file_num[], data_file[];
extern char  comment[], stage[], buffer[];

/*      open_data_file() obtains a file name from the user for
data storage.  If that file already exists, the user has the options
of overwriting or trying again.  Choosing to overwrite a file results
in that file´s being deleted.  Note that spectral files are indicated
by the insertion of ´S´ after the ´.´ and before the file number.
*/

void open_data_file(int m)
{

char  inputs2[2];
char  *input, *input2;
FILE  *stream;

do{
    _settextposition(8, 15);
    rcoord = _gettextposition();
    if (strcmp(data_file, "0") != 0){
        sprintf(buffer, "The last file was %s", data_file);
```

```
        _settextcolor(ecolor);
        _outtext(buffer);
        }

    _settextcolor(tcolor);
    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext(multi_string(7));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    input = gets(file_name);
    if (!strcmp("", file_name)){
        _clearscreen(_GCLEARSCREEN);
        return;
        }
    strcpy(data_file, file_name);
    if (m == 1)
        strcat(data_file, ".");
    else if (m == 2)
        strcat(data_file, ".S");

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext(multi_string(8));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    input = gets(file_num);
    strcat(data_file, file_num);

    if ((stream = fopen(data_file, "rb")) != NULL){
        fclose(stream);
        rcoord.row++;
        _settextcolor(ecolor);
        _settextposition(rcoord.row, rcoord.col);
        _outtext(prompt_string(2));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(prompt_string(3));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(prompt_string(7));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        input2 = gets(inputs2);
```

```
        if (inputs2[0] == 'y' || inputs2[0] == 'Y')
            unlink(data_file);
        }
    } while(inputs2[0]!='y' && inputs2[0]!='Y' && stream!=NULL);
_clearscreen(_GCLEARSCREEN);
return;
}


/*      save() saves the current data set in the file given by
file_name and file_num with file_num incremented by n (the number
of scans-1, non-zero only in multi-mode). Note that the variable
point_scan_tak is passed to save because the data collection may
have been stopped prematurely and points_scan may not be equal to
the number of data points that were actually taken. If the file
already exists, open_data_file() is called. Note that for stage
files the first three characters in the header are 'g', 'a', and
'd', while for spectra they are 't', 'r', and 'm'.
*/

void save(int n, int point_scan_tak, int m)
{
time_t        ltime;
int  i;
int  num1, numwritten;
char  *p, file_numt[3], *input, inputs[2];
FILE  *stream;

if (file_name[0] == '\0')
    return;

strcpy(data_file, file_name);
if (m == 1)
    strcat(data_file, ".");
else if (m == 2)
    strcat(data_file, ".S");
num1 = atoi(file_num);
num1 = num1 + n;
p = itoa(num1, file_numt, 10);
strcat(data_file, file_numt);

if ((stream = fopen(data_file, "rb")) != NULL){
    fclose(stream);
    _settextcolor(ecolor);
    _settextposition(10, 15);
    rcoord = _gettextposition();
    _outtext(data_file);

    rcoord.row++;
```

```
    _settextposition(rcoord.row, rcoord.col);
    _outtext(prompt_string(2));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext(prompt_string(3));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext(prompt_string(7));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    input = gets(inputs);
    _clearscreen(_GCLEARSCREEN);

    if (inputs[0] != 'y' && inputs[0] != 'Y')
        open_data_file(m);
    }

stream = fopen(data_file, "wb");

if (m == 1)
    fprintf(stream, "gad");
else if (m == 2)
    fprintf(stream, "trm");
fprintf(stream, " %li ", ltime);
if (m == 1)
    fprintf(stream, "%i %i ", point_scan_tak, stage_step);
else if (m == 2){
    stage_end = (point_scan_tak - 1) * nm_per_point + (int)stage_beg;
    fprintf(stream, "%i %i ", nm_per_point, stage_end);
    }
fprintf(stream, "%s %li ", dur, stage_beg);
fprintf(stream, "%i %i %i ", reverse_flag, multi_count, n+1);
fprintf(stream, "%s", comment);
for (i=0; i!=4; i++)
    numwritten=fwrite(data[i],sizeof(double),point_scan_tak,stream);
if (m == 1)
    fprintf(stream, " %s", stage);
else if (m == 2)
    fprintf(stream, " %i", steps_per_nm);
fclose(stream);
}
```

## A.6 slook.c

```
/*      This file contains the following routines:  look_data(),
plot(), dat_plot(), plottop(), edit_comment(), and resave().
*/

#include <malloc.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <graph.h>
#include <conio.h>
#include <bios.h>
#include <math.h>
#include "sgadd.h"
#include <string.h>

/*      Note the use of static variables in this file.  This is to
allow the user to look at data with different set-up parameters
without affecting the set-up parameters used in taking the data.
*/

static int  points_scan, stage_step, nm_per_point, stage_end,
        steps_per_nm;
static int  multi_count, reverse_flag, scan;
static long int  stage_beg;
static double  *data[4];
static char  file_name[20], file_num[3], data_file[32];

static int  point_scan_tak, multi_flag;
static char  comment[500], stage[50];
static char  dur[8];
static float  dmax, dmin, dmax1, dmin1;

extern int  tcolor, ecolor, box_color, channel_color1, channel_dis1;
extern long int  rect_xmin, rect_ymin, rect_xmax, rect_ymax;
extern char  buffer[];
extern struct rccoord        rcoord;

time_t        ltime;
int  commnum;


/*      look_data() allows the user to view previously collected
data.
*/

void look_data(int m)
```

```
{
int  i, chan, j, k;
int  numwritten, ch;
FILE  *stream;
char  inputs2[2], test[4], inputs[2];
char  *input, *input2;

inputs2[0] = 'a';
stream = NULL;
input2 = inputs2;
input = inputs;

/*      First, the user inputs a file name.  If it does not exist or
is not of the correct type as indicated by its first three header
characters the user may quit or try again.  Note also that after
viewing one file, the user may then view another, continue to the
next file in the number series, view or edit the comment, or quit.
The string in inputs2[] indicates which option will be pursued having
respective values 'a', 'c', 'e', and 'q'.
*/

do{
    _settextcolor(tcolor);
    _settextposition(9, 20);
    rcoord = _gettextposition();

    if (inputs2[0] == 'a'){
        _outtext(multi_string(7));
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        input = gets(file_name);
        if (strcmp(file_name, "") == 0){
            _setvideomode(_DEFAULTMODE);
            return;
            }
        }

    strcpy(data_file, file_name);
    if (m == 1)
        strcat(data_file, ".");
    else if (m == 2)
        strcat(data_file, ".S");

    if (inputs2[0] == 'a'){
        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(multi_string(8));

        rcoord.row++;
```

```
        _settextposition(rcoord.row, rcoord.col);
        input = gets(file_num);
        }

/*      If ´C´ or ´c´ has been hit the next data file with the same
name but incremented by one number is read in.  For example, if the
user was looking at "steve.1" and  the  ´C´ or ´c´ key was hit,
"steve.2" would be read in.
*/

    if (inputs2[0] == ´c´){
        i = atoi(file_num);
        i++;
        itoa(i, file_num, 10);
        }

    strcat(data_file, file_num);
    if (inputs2[0] == ´e´)
        edit_comment(m);

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

    if ((stream = fopen(data_file, "rb")) == NULL){
        rcoord.row++;
        _settextcolor(ecolor);
        _settextposition(rcoord.row, rcoord.col);
        _outtext(prompt_string(4));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(prompt_string(5));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(prompt_string(7));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        input2 = gets(inputs2);
        }
    _clearscreen(_GCLEARSCREEN);

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

```
test[i] = '\0';
if (strcmp(test,"gad")!=0 && m==1 || strcmp(test,"trm")!=0 &&
                                                    m == 2){
    _settextcolor(ecolor);
    _settextposition(9, 20);
    rcoord = _gettextposition();
    _outtext(data_file);

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    if (m == 1)
        _outtext(error_string(9));
    else if (m == 2)
        _outtext(error_string(10));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext(prompt_string(5));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext(prompt_string(7));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    input2 = gets(inputs2);
    _clearscreen(_GCLEARSCREEN);
    }
else{
    fscanf(stream, " %li ", &ltime);
    if (m == 1)
        fscanf(stream, "%i %i ", &points_scan, &stage_step);
    else if (m == 2)
        fscanf(stream, "%i %i ", &nm_per_point, &stage_end);
    fscanf(stream, "%s %li ", dur, &stage_beg);

    if (m == 1)
        stage_end = (int)(stage_beg +
                                (points_scan-1)*stage_step);
    else if (m == 2)
        points_scan = (stage_end - (int)stage_beg) /
                                        nm_per_point + 1;

    fscanf(stream, "%i %i %i ", &reverse_flag, &multi_count,
                                        &scan);

    for (i = 0; (i < 498) && (ch=fgetc(stream)) != '~'; i++)
        comment[i] = (char) ch;
    comment[i] = (char) ch;
```

```
            i++;
            ch = fgetc(stream);
            comment[i] = (char)ch;
            i++;
            comment[i] = (char) '\0';
            commnum = i;

            for (i=0; i<4; i++)
                if (data[i] != NULL)
                    free (data[i]);

            for (i=0; i<4; i++){
                data[i]=(double*)malloc(points_scan*sizeof(double));
                if (data[i] == NULL){
                    for (j=0; j<i; j++)
                        free(data[j]);
                    _settextposition(10, 15);
                    rcoord = _gettextposition();
                    _settextcolor(ecolor);
                    sprintf(buffer,"%s%i", error_string(11), i);
                    _outtext(buffer);

                    rcoord.row++;
                    _settextposition(rcoord.row, rcoord.col);
                    _outtext(odd_string(1));

                    while (k == 0)
                        k = kbhit();
                    k = getche();
                    _settextcolor(tcolor);
                    _setvideomode(_DEFAULTMODE);
                    return;
                    }
                }

            for (i=0; i!=4; i++)
                numwritten=fread(data[i],sizeof(double),points_scan,
                    stream);

            if (m == 1)
                fscanf(stream, " %s", stage);
            else if (m == 2)
                fscanf(stream, " %i", &steps_per_nm);
            fclose(stream);

/*      With the current data collection system, only one channel
is utilized, namely channel 0.  Thus, this next section is commented
out but is saved for possible future use.
*/
```

```
            chan = 0;

            /*if (inputs2[0] == 'a'){
                do{
                    _settextposition(10, 20);
                    _settextcolor(tcolor);
                    rcoord = _gettextposition();
                    _outtext(look_string(1));

                    rcoord.row++;
                    _settextposition(rcoord.row, rcoord.col);
                    input2 = gets(inputs2);
                    chan = atoi(inputs2);

                    if (chan < 0 || chan > 3){
                        _clearscreen(_GCLEARSCREEN);
                        _settextcolor(ecolor);
                        _settextposition(10, 20);
                        _outtext(error_string(5));
                        }
                    } while (chan < 0 || chan > 3);
                _clearscreen(_GCLEARSCREEN);
                }*/

/*      The data having been read from the input file, plot() is
then called to display the data and retrieve from the user the
aforementioned options regarding further viewing.
*/

            inputs2[0] = (char)(plot(chan, m) & 0x0FF);

            /*while (inputs2[0] == '0' || inputs2[0] == '1'||
                        inputs2[0] == '2' || inputs2[0] == '3'){
                chan = atoi(inputs2);
                inputs2[0] = (char)(plot(chan, m) & 0x0FF);
                }*/
            }
        }
    } while (inputs2[0]=='c' || inputs2[0]=='a' || inputs2[0]=='e');

for (i=0; i<4; i++)
    if (data[i] != NULL)
        free (data[i]);

/*      cend() frees space that was allocated for some of
the plotting routines.
*/
```

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


/*      plot() displays the data and retrieves options from the user
regarding future viewing.  It also allows the user to move a cursor
from point to point and see the actual data value.  This can be done
singly, in steps of ten, or by jumping directly to the first, last,
maximal, or minimal points.  The screen can also be toggled between
a full scale plot with 0 minimum and an expanded view scaled to the
minimum of the data.
*/

int  plot(int chan, int m)
{
int  i, ch, j, imin, imax, scale = 1;
int  imin1, imax1, fitflag = 0;
float  max, min, max1, min1;

/*      The top of the screen is plotted by plottop() and the
boxes are drawn.
*/

ch = 'n';
plottop(m);

_settextcolor(tcolor);
_settextposition(24, 1);
rcoord = _gettextposition();
_outtext(look_string(2));

rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
_outtext(look_string(3));

_setcolor(box_color);
_rectangle(_GBORDER, (short)rect_xmin, (short)rect_ymin,
        (short)rect_xmax, (short)rect_ymax);

/*      If there is data in the second array as the result of some
other program, for example a fitting program, it will be displayed
in this routine as well.  It is shown as the second channel.
*/

if (*data[1] > .01)
    fitflag = 1;
```

```
dmax = (float)*(data[chan]) + .01;
dmin = (float)*(data[chan]) - .01;
imin = 0;
imax = 0;

dmax1 = (float)*(data[1]) + .01;
    dmin1 = (float)*(data[1]) - .01;

/*      The max and min values of the data are determined, and the
data are drawn by dat_plot().
*/

for (i=0; i<points_scan; i++){
    if (*(data[chan] + i) > dmax){
        dmax = (float) *(data[chan] + i);
        imax = i;
        }
    else if (*(data[chan] + i) < dmin){
        dmin = (float) *(data[chan] + i);
        imin = i;
        }
    if (fitflag == 1){
        if (*(data[1] + i) > dmax1){
            dmax1 = (float) *(data[1] + i);
            imax1 = i;
            }
        else if (*(data[1] + i) < dmin1){
            dmin1 = (float) *(data[1] + i);
            imin1 = i;
            }
        }
    }
max = dmax;
min = dmin;
if (fitflag == 1){
    if (dmax < dmax1)
        max = max1 = dmax1;
    else
        max1 = max = dmax;

    if (dmin > dmin1)
        min = min1 = dmin1;
    else
        min1 = min = dmin;
    }
dat_plot(chan, m, max, 0, fitflag, max1, 0);

i = 0;
j = 1;
```

```
/*      csave() and rpointc() deal with cursor allocation, placement,
and movement.  See also splot.c.
*/

if (m == 1){
    csave((float)(stage_beg+i*stage_step), (float)*(data[chan]+ i));
    rpointc((float)(stage_beg+i*stage_step), (float)*(data[chan]+i));
    }
else if (m == 2){
    csave((float)(stage_beg+i*nm_per_point), (float)*(data[chan]+i));
    rpointc((float)(stage_beg+i*nm_per_point),(float)*(data[chan]+i));
    }

/*      The next section of code moves the cursor around the
screen and allows the user to provide other input.  The values
of ch returned by _bios_keybrd correspond as follows:
        exit characters:
´q´ == 0x1071, ´0´ == 0x0b30, ´1´ == 0x0231, ´2´ == 0x0322,
´3´ == 0x0433, ´c´ == 0x2e63, ´a´ == 0x1e61, and ´e´ == 0x1265;
        cursor/rescale characters:
0x4700 == HOME    : moves cursor to the first point;
0x4f00 == END     : moves cursor to the last point;
0x4900 == PGUP    : moves cursor to the maximal point;
0x5100 == PGDN    : moves cursor to the minimal point;
0x4800 == UPARROW : moves cursor forward 10 points (if possible);
0x5000 == DNARROW : moves cursor backward 10 points (if possible);
0x4d00 == RTARROW : moves cursor forward 1 point (if possible);
0x4b00 == LFARROW : moves cursor backward 1 point (if possible);
0x1372 == ´r´     : rescales the y-axis.
*/

ch = _bios_keybrd(_KEYBRD_READ);
while (ch!=0x1071 && /*ch!=0x0b30 && ch!=0x0231 && ch!=0x0332 &&
        ch!=0x0433 &&*/ ch!=0x2e63 && ch!=0x1e61 && ch!=0x1265){
    if (ch == 0x4700 && j != 1){
        j = 1;
        i = 0;
        if (m == 1)
            rpointc((float)(stage_beg + i * stage_step),
                                        (float)*(data[chan] + i));
        else if (m == 2)
            rpointc((float)(stage_beg + i * nm_per_point),
                                        (float)*(data[chan] + i));
        }
    if (ch == 0x4f00 && j < points_scan){
        j = points_scan;
        i = points_scan - 1;
        if (m == 1)
```

```
            rpointc((float)(stage_beg + i * stage_step),
                                        (float)*(data[chan] + i));
        else if (m == 2)
            rpointc((float)(stage_beg + i * nm_per_point),
                                        (float)*(data[chan] + i));
        }
    if (ch == 0x4900){
        j = imax + 1;
        i = imax;
        if (m == 1)
            rpointc((float)(stage_beg + i * stage_step),
                                        (float)*(data[chan] + i));
        else if (m == 2)
            rpointc((float)(stage_beg + i * nm_per_point),
                                        (float)*(data[chan] + i));
        }
    if (ch == 0x5100){
        j = imin + 1;
        i = imin;
        if (m == 1)
            rpointc((float)(stage_beg + i * stage_step),
                                        (float)*(data[chan] + i));
        else if (m == 2)
            rpointc((float)(stage_beg + i * nm_per_point),
                                        (float)*(data[chan] + i));
        }
    if (ch == 0x4800 && j  <= points_scan - 10){
        j = j + 10;
        i = i + 10;
        if (m == 1)
            rpointc((float)(stage_beg + i * stage_step),
                                        (float)*(data[chan] + i));
        else if (m == 2)
            rpointc((float)(stage_beg + i * nm_per_point),
                                        (float)*(data[chan] + i));
        }
    if (ch == 0x5000 && j  > 10){
        j = j - 10;
        i = i - 10;
        if (m == 1)
            rpointc((float)(stage_beg + i * stage_step),
                                        (float)*(data[chan] + i));
        else if (m == 2)
            rpointc((float)(stage_beg + i * nm_per_point),
                                        (float)*(data[chan] + i));
        }
    if (ch == 0x4b00 && j != 1){
        j--;
        i--;
```

```
        if (m == 1)
            rpointc((float)(stage_beg + i * stage_step),
                                        (float)*(data[chan] + i));
        else if (m == 2)
            rpointc((float)(stage_beg + i * nm_per_point),
                                        (float)*(data[chan] + i));
        }
    else if (ch == 0x4d00 && j < points_scan){
        i++;
        j++;
        if (m == 1)
            rpointc((float)(stage_beg + i * stage_step),
                                        (float)*(data[chan] + i));
        else if (m == 2)
            rpointc((float)(stage_beg + i * nm_per_point),
                                        (float)*(data[chan] + i));
        }
    else if (ch == 0x1372){
        scale = -scale;

        cend();
        plottop(m);
        _settextcolor(tcolor);
        _settextposition(24, 1);
        rcoord = _gettextposition();
        _outtext(look_string(2));

        rcoord.row++;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(look_string(3));

        _setcolor(box_color);
        _rectangle(_GBORDER, (short)rect_xmin, (short)rect_ymin,
                (short)rect_xmax, (short) rect_ymax);

        if (scale == 1)
            dat_plot(chan, m, max, 0, fitflag, max1, 0);
        else
            dat_plot(chan, m, max, min, fitflag, max1, min1);
        i = 0;
        j = 1;
        if (m == 1){
            csave((float)(stage_beg + i * stage_step),
                                        (float)*(data[chan] + i));
            rpointc((float)(stage_beg + i * stage_step),
                                        (float)*(data[chan] + i));
            }
        else if (m == 2){
            csave((float)(stage_beg + i * nm_per_point),
```

```
                                        (float)*(data[chan] + i));
            rpointc((float)(stage_beg + i * nm_per_point),
                                        (float)*(data[chan] + i));
            }
        }

    ch = _bios_keybrd(_KEYBRD_READ);
    }
_setvideomode(_DEFAULTMODE);
return(ch);
}


/*      dat_plot() takes care of plotting out the axes and data.
*/

dat_plot(int chan, int m, float max, float min,
                    int fitflag, float max1, float min1)
{
int  i;

rbound1((float)stage_beg, min, (float)stage_end, max);

if (fitflag == 1)
    rbound2(min1, max1);

for (i=0; i < points_scan; i++){
    if (m == 1){
        rpoint1((float)(stage_beg + i * stage_step),
                                    (float)*(data[chan] + i));

        if (fitflag == 1)
            rpoint2((float)i + 1, (float)*(data[1] + i));

        }
    else if (m == 2){
        rpoint1((float)(stage_beg + i * nm_per_point),
                                    (float)*(data[chan] + i));

         if (fitflag == 1)
                rpoint2((float)(stage_beg + i * nm_per_point),
                                    (float)*(data[1] + i));

        }
    }
}
```

```
/*      plottop() plots the information that is above the window
of data.  Note that this is a different subroutine than top()
because of the use of the static variables for looking at data.
*/

void plottop(m)
int  m;
{
char  *p, file_numt[3];
int  num1, i;

if (_setvideomode(_ERESCOLOR))
    ;
else{
    _setvideomode(_DEFAULTMODE);
    _settextposition(10, 20);
    rcoord = _gettextposition();
    _settextcolor(ecolor);
    _outtext(odd_string(6));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    _outtext(odd_string(1));

    while (i == 0)
        i = kbhit();
    i = getche();
    _clearscreen(_GCLEARSCREEN);
    return;
    }

_settextcolor(tcolor);
_settextposition(1, 1);
rcoord = _gettextposition();
_outtext(window_string(1));
_outtext(data_file);

rcoord.col = rcoord.col + 25;
_settextposition(rcoord.row, rcoord.col);
time(&ltime);
_outtext((ctime(&ltime)));

rcoord.col = rcoord.col + 35;
_settextposition(rcoord.row, rcoord.col);
sprintf(buffer, "Display %i  ", channel_dis1);
_settextcolor(channel_color1);
_outtext(buffer);

_settextcolor(tcolor);
```

```
_settextposition(2, 1);
rcoord = _gettextposition();
_outtext(window_string(4));
if (m == 1)
    sprintf(buffer, "%li", stage_beg);
else if (m == 2)
    sprintf(buffer, "%li nm", stage_beg);
_outtext(buffer);

rcoord.col = rcoord.col + 20;
_settextposition(rcoord.row, rcoord.col);
_outtext(window_string(5));
sprintf(buffer, "%i", points_scan);
_outtext(buffer);

rcoord.col = rcoord.col + 20;
_settextposition(rcoord.row, rcoord.col);
_outtext(window_string(3));
_outtext(dur);

if (multi_count != 1 && multi_count != 0){
    _settextposition(3, 1);
    rcoord = _gettextposition();
    _outtext(window_string(6));

    rcoord.col = rcoord.col + 20;
    _settextposition(rcoord.row, rcoord.col);
    sprintf(buffer, "Scan %i out of %i scans", scan, multi_count);
    _outtext(buffer);

    if (reverse_flag == 2){
        rcoord.col = rcoord.col + 30;
        _settextposition(rcoord.row, rcoord.col);
        _outtext(window_string(8));
        }
    }
}


/*      edit_comment() allows the user to change the comment in a
safer manner than directly editing the file via the OS.
*/

void edit_comment(int m)
{
int  i, ch;
char  *input, inputs[2];

_settextcolor(tcolor);
```

```
_settextposition(5, 5);
rcoord = _gettextposition();
_settextposition(rcoord.row, rcoord.col);
_outtext(set_string(11));
_wrapon(_GWRAPON);

if(comment[0] != '~'){
    rcoord.row++;
    _settextcolor(ecolor);
    _settextposition(rcoord.row, rcoord.col);
    _outtext(comment);
    _settextcolor(tcolor);
    }

rcoord.row+=2;
_settextposition(rcoord.row, rcoord.col);
_outtext("Hit e to change the comment.");

rcoord.row++;
_settextposition(rcoord.row, rcoord.col);
input = gets(inputs);
if (inputs[0] == 'e' || inputs[0] == 'E'){
    _clearscreen(_GCLEARSCREEN);
    _settextposition(15, 5);
    rcoord = _gettextposition();
    _settextposition(rcoord.row, rcoord.col);
    _outtext(stage_string(5));

    rcoord.row++;
    _settextposition(rcoord.row, rcoord.col);
    for (i=0; (i<498) && (ch=getchar()) != '~'; i++)
        comment[i] = (char)ch;
    comment[i] = (char)ch;
    i++;
    ch = getchar();
    comment[i] = (char)ch;
    i++;
    comment[i] = '\0';

    resave(m);
    }
_clearscreen(_GCLEARSCREEN);
return;
}


/*      If the user edited the comment, resave() is called to save
the new comment as well as to resave all the old data in the same
file.
```

```
*/

void resave(int m)
{
int  i;
int  numwritten;
FILE  *stream;

stream = fopen(data_file, "wb");

if (m == 1)
    fprintf(stream, "gad");
else if (m == 2)
    fprintf(stream, "trm");
fprintf(stream, " %li ", ltime);
if (m == 1)
    fprintf(stream, "%i %i ", points_scan, stage_step);
else if (m == 2)
    fprintf(stream, "%i %i ", nm_per_point, stage_end);
fprintf(stream, "%s %li ", dur, stage_beg);
fprintf(stream, "%i %i %i ", reverse_flag, multi_count, scan);
fprintf(stream, "%s", comment);
for (i=0; i!=4; i++)
    numwritten = fwrite(data[i],sizeof(double),points_scan,stream);
if (m == 1)
    fprintf(stream, " %s", stage);
else if (m == 2)
    fprintf(stream, " %i", steps_per_nm);
fclose(stream);
}
```

## A.7 sconst.c

```
/*      This file contains the following routines:
main_string(), error_string(), setup_string(), stage_string(),
specset_string(), set_string(), prompt_string(), data_string(),
multi_string(), window_string(), look_string(), and odd_string().
        This file contains most of the string constants used in
the program.  This was done for two reasons: it saves data space
for strings that are used more than once; and it allows some
menus to be output with for loops instead of having one line
for each output.  Basically, these are subroutines that return
a string.  The first part is the definition of the static char
which is an array of character strings.  The function then returns
the string element indexed by the input argument if that argument
is of an acceptable value; otherwise it returns the 0th string.
*/

#include "sgadd.h"


/*      main_string() is used for the main menu.
*/

char *main_string(n)
int  n;
{
static char  *string[] = {
        "error",
        "1: Read in stage setup file",
        "2: Read in spectrum setup file",
        "3: Create stage setup file",
        "4: Create spectrum setup file",
        "5: Take DATA with stage",
        "6: Take a SPECTRUM",
        "7: Look at DATA taken with stage",
        "8: Look at a SPECTRUM",
        "9: Quit",
        };
return((n < 1 || n > 9) ? string[0] : string[n]);
}


/*      error_string() is used for a variety of errors.
*/

char *error_string(n)
int  n;
{
static char  *string[] = {
        "error",
        "Stage parameters not set",
        "Monochromator not set",
        "Duration not set",
        "This is not a stage setup file",
        "Not an acceptable number",
        "Bounds not set",
```

```
        "This is not a spectrum setup file",
        "You must type either long or short",
        "This is not a data file",
        "This is not a spectrum file",
        "Malloc failed on "
        };
return((n < 1 || n > 11) ? string[0] : string[n]);
}


/*      setup_string() is used for the set-up menu for stage data;
strings 5 - 9 are also used in the set-up menu for spectra.
*/

char *setup_string(n)
int  n;
{
static char  *string[] = {
        "error",
        " 1: Stage starting position",
        " 2: # of points in scan",
        " 3: # of stage steps between points",
        " 4: Choose stage",
        " 5: Duration (in ms)",
        " 6: Comment of 498 characters or less - end with ~",
        " 7: Display all ",
        " 8: Save setup",
        " 9: Return to main menu"
        };
return((n < 1 || n > 9) ? string[0] : string[n]);
}


/*      stage_string() is used in the routines in which the user
enters the parameters for data taken with the stage or for parameters
used the same for both stage data and spectra.
*/

char *stage_string(n)
int  n;
{
static char  *string[] = {
        "Invalid string",
        "Input total # of points in scan",
        "Input # of stage steps between points",
        "Input duration (in ms) - maximum 10000",
        "Input stage position where scan should begin",
        "Input comment of 498 characters or less and end with a ~",
        "Input stage name (short or long)",
        "Input lower bound",
        "Input upper bound"
        };
return((n < 1 || n > 8) ? string[0] : string[n]);
}


/*      specset_string() is used for the set-up menu for spectra and
```

```
in the routines in which the user enters the parameters for spectra.
*/

char *specset_string(n)
int n;
{
static char  *string[] = {
        "Take a SPECTRUM menu:",
        " 1: Initial wavelength in nm",
        " 2: Final wavelength in nm",
        " 3: # nanometers between points",
        " 4: # steps per nanometer",
        "Input initial wavelength (in nm)",
        "Input final wavelength (in nm)",
        "Input # nanometers between points",
        "Input # motor steps per nanometer (2 or -8(surf))",
        "Input the current monochromator position (in nm)"
        };
return((n < 1 || n > 9) ? string[0] : string[n]);
}


/*      set_string() is used by the routine display().
*/

char *set_string(n)
int  n;
{
static char  *string[] = {
        "error",
        "Stage start",
        "Stage end",
        "Stage steps between points",
        "Initial wavelength in nm",
        "Final wavelength in nm",
        "Nanometers between points",
        "Duration per point",
        "Points in scan",
        "Lower bound on data",
        "Upper bound on data",
        "Comment"
        };
return((n < 1 || n > 11) ? string[0] : string[n]);
}


/*      prompt_string() is used in a variety of routines to prompt
the user for information regarding file management and to display
warnings for the same.
*/

char *prompt_string(n)
int  n;
{
static char  *string[] = {
        "error",
        "Enter a file name ",
```

```
        "File Already Exists!!!",
        "Do you want to OVERWRITE?",
        "File Does NOT Exist!!!",
        "Do you want try another?",
        "Answer Y or N or Q",
        "Answer Y or N"
        };
return((n < 1 || n > 7) ? string[0] : string[n]);
}


/*      data_string() is used for the data taking menu.
*/

char *data_string(n)
int  n;
{
static char  *string[] = {
        "Take DATA with stage menu:",
        "1: Change comment",
        "2: Set up multi",
        "3: Open data files",
        "4: Take DATA",
        "5: Look at DATA",
        "6: Return to main menu"
        };
return((n < 1 || n > 6) ? string[0] : string[n]);
}


/*      multi_string() is used by the routine set_multi().
*/

char *multi_string(n)
int  n;
{
static char  *string[] = {
        "error",
        "Input number of scans (must be even number)",
        "Must be an even, positive number",
        "1: Stage takes data in one direction",
        "2: Stage takes data in both directions",
        "Input 1 or 2",
        "5: Return to main menu",
        "Input data file name (maximum 8 characters)",
        "Input data file number (maximum 3 digits)",
        "1: Turn multi off",
        "2: Turn multi on",
        };
return((n < 1 || n > 10) ? string[0] : string[n]);
}


/*      window_string() is used by top() and plottop() for strings
appearing at the top of data display windows.
*/
```

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


/*      look_string() is used for strings to be displayed below
the data display window when viewing previously collected data.
*/

char *look_string(n)
int  n;
{
static char  *string[] = {
        "error",
        "Input channel: 0, 1, 2, or 3",
      "Input q=quit, a=do another file, c=continue to the next file,",
        "r=rescale, e=edit comment, or new channel in current file"
        };
return((n < 1 || n > 3) ? string[0] : string[n]);
}


/*      odd_string() contains a wide variety of strings used
throughout the program.
*/

char *odd_string(n)
int  n;
{
static char  *string[] = {
        "error",
        "Hit a key to continue",
        "CheckListener error; does device or board",
        "   name match given configuration name?",
        "GPIB function call error:",
        "Input a y to continue or a q to quit",
        "Graphics not supported.",
        "Input q to quit y for another",
        "Cannot initialize IEEE system"
        };
return((n < 1 || n > 8) ? string[0] : string[n]);
```

```
}
```

# Appendix B

# Data Fitting Program

## B.1 fitX.c

```
/*      The following files are necessary to compile this program:
fitaux.c, fitgraph.c, calc.c.  Also required are the numerical
recipes routines convlv.c, covsrt.c, four1.c, gaussj.c, mrqcof.c,
mrqmin.c, nrutil.c, realft.c, and twofft.c.  See also Makefile for
more information on compiling.
        This file contains the following routines:
main(), create_control_panel(), create_file_popups(),
create_set_popups(), mc_proc(), file_open_proc(),
file_save_proc(), file_close_proc(), func_proc(), conv_proc(),
flag_proc(), ibase_proc(), mc_set_proc(), file_saveas(),
func_choose(), conv_choose(), flag_set(), ibase_set(), mc_set(),
and quit_proc().
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/notice.h>
#include <xview/cms.h>
#include <xview/svrimage.h>
#include <xview/icon.h>
```

```
#include "fitX.h"
#include "/home/xenon/steve/Bitmaps/bill.bmp"
#include "/home/xenon/steve/Bitmaps/bill_mask.bmp"

/*      The structure dat_set is used to hold all of the important
data needed by each window.  This includes all of the data imported
from the data file and the parameters of the fit.  The variables
loaded from the data files are detailed in the data collection
program.  The remainder are described below.
        file_name is the name of the data file whose data is being
fit by the current window.
        func indicates the analytical function being convolved with
the response:  1:biexponential, 2:Kohlrausch.
        conv indicates the response function:  1:Gaussian,
2:sech squared, 3:Lorentzian, 4:input response data.
        time_per_point is the conversion for data points to time in
picoseconds.
        data is the array of pointers to the data.
        datmax and datmin hold the maximum and minimum data values
after correction for the calculated baseline value.
        param stores the values of the fitting parameters, starting
with the FWHM in the zeroth element and progressing in the order a1,
a2, b1, b2, c1, c2, c3, and c4.
        tzero holds the position of the time zero point.  The default
value is 1.
        left_bnd and right_bnd indicate the points between which
the squared error is calculated.  Default values are 1 and
data.points_scan, respectively.
        flags indicates the type of data being examined in the first
three bits:  1: 0 => fs data, 1 => ps data; 2: 0 => absorption,
1 => fluorescence; 3: 0 => not a bleach, 1 => a bleach.  Bit four is
used to let the program know that autocorrelation data is going to
be fit.  This data will be fit by one of the bare response functions
(that is unconvolved).  This also changes the effect of the Fit
button.  The fifth bit indicates whether or not the contribution of
each residual to the squared errors is weighted by dividing it by
the corresponding data value divided by 100 or not (1 or 0).  The
default value is 3, that is non-bleach ps fluorescence with
unweighted errors.
*/

struct dat_set  data;

/*      lstsqer is the squared error and ibase is the first point of
the baseline.
*/

double  lstsqer = 0;
int  ibase;
```

```
/*      The array buffer is a generic char buffer.  Its size ensures
that it can accept the comment from opened data files; the comment is
not saved.
*/

char  buffer[500];

/*      The array pdelta holds the incremental values for changing
the parameters by the keypad.  The arrays pmax and pmin hold upper
and lower bounds on the values of the parameters.  pdelta is set up
in set_pdelta_mat().  The values depend upon the type of data as
indicated by data.flags.  pmax and pmin are set up in
set_pextrema_mats().  Note that their values depend on the function
selected.
*/

float  pdelta[12], pmin[9], pmax[9];

/*      saved stores the save status of the data corresponding to the
identically indexed canvas in dat_wins.  Values are -1:no data yet;
0:saved; 1:never saved; and 2:saved, but changed since last save.
*/

int  saved;

/*      The nine least significant bits in mc_flags indicate which of
the parameters are to be randomly varied when the "Fit" button is
selected.  The bits correspond in increasing order to the parameters
FWHM and a1-c4.  If the bit is "on", the corresponding parameter is
active; otherwise it´s not.  flag_mask indicates which parameters are
valid for the current decay function.
*/

int  mc_flags, flag_mask;

/*      def_event_proc is a pointer to a function returning a type
void.  It is used to store the location of the default event handler
for a text item.  See also keypad().
*/

void (*def_event_proc)();

/*      The following variables are used for input response data.
Note that only _one_ set of response data can be used by the program
at a time.  fcnv_name is the array containing the name of the file
containing the response data.  cnvpts is the number of points of
said data.  resp_step is the number of stage steps per point in
gathering the response data.  icnvmax is the maximum value of the
```

```
response data set.  cnv is a pointer to the dynamically allocated
array of response data.
*/

char  fcnv_name[51];
int  cnvpts, resp_step, icnvmax;
double  *cnv;

/*      The following are the indices into the colormap segment used
in this program for both plotting and the control panel.  Their
values are set in setup_graphics().
*/

int  data_color, fit_color, resid_color, t0_color, box_color;
int  label_color;
int  white, black;

/*      The following are many of the xview structures used in
constructing the user interface for this program.  main_frame and
control_panel are the frame and panel of the control panel
respectively.
        file_menu and set_menu are the menu structures invoked by the
"File" and "Set" buttons respectively.
        file_open_frame and file_saveas_frame are the frames for the
windows invoked by selecting the "Open" and "Save As" options in
file_menu respectively.
        sd_text, t0_text, left_bnd_text, and right_bnd_text are the
panel items, in this case text items, corresponding to the displayed
items for FWHM, t0, left, and right on the control panel.  error_item
is the panel item (a panel message) used to display the value of the
squared error.
        func_frame, conv_frame, flag_frame, ibase_frame, and
mc_set_frame are the frames for the windows displayed upon selecting
the corresponding item from the set_menu.  fcnv_open_frame is
displayed upon selecting to use response data in place of an
analytical response function.
        fname_item, sname_item, and fcnv_name_item are the panel
items for the file names entered in the file_open_frame,
file_saveas_frame, and fcnv_open_frame respectively.  comment_item
is the item for the comment that can be saved in the header.
        func_item and conv_item are the panel (choice) items on the
panels owned by func_frame and conv_frame respectively.  flag_item
and mc_set_item are the panel (toggle and checkbox) items on the
panels owned by flag_frame and mc_set_frame respectively.  ibase_item
is the panel (text) item on the panel owned by ibase_frame.
        cms is the colormap segment created in setup_graphics() and
used by both control panel and canvases alike.
        dat_win is the data window canvas.
        Note that for all of the above there is always and exactly
```

```
one such item created.
*/

Frame  main_frame;
Panel  control_panel;
Menu  file_menu, set_menu;
Panel_item  sd_text, t0_text, left_bnd_text, right_bnd_text;
Panel_item  error_item;

Frame  file_open_frame, file_saveas_frame, fcnv_open_frame;
Frame  func_frame, conv_frame, flag_frame, ibase_frame, mc_set_frame;

Panel_item  fname_item, sname_item, comment_item, fcnv_name_item;
Panel_item  func_item, conv_item, flag_item, ibase_item, mc_set_item;
Cms  cms;
Canvas  dat_win;

/*      The main routine first presets several variables.  Note that
the value of flag_mask corresponds to the default value of
data.func = 1, which corresponds to the biexponential.
        xv_init strips and applies any command line arguments
intended for the server and initializes the connections thereto.  At
most there should remain one argument (beside the program name), a
file to be opened initially.  If there are too many arguments, the
program returns after printing an error message.
        Next the main frame is created and a routine is interposed on
the destruction to ensure proper cleanup of dynamically allocated
variables.  Then the remainder of the setup is carried out in several
routines and the frame is fit around its panel.
        If an initial file was indicated, it is opened.  In any case
the the main loop is entered.
*/

static Notify_value fit_destroy_func();

main(int argc, char *argv[])
{
Server_image  bill_image, bill_mask_image;
Icon  bill_icon;

data.tzero = 1;
data.left_bnd = 1;
data.right_bnd = 1;
data.func = 1;
data.conv = 1;
data.flags = (unsigned) 3;
mc_flags = 0;
flag_mask = 367;
saved = -1;
```

```
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
if (argc > 2){
    fprintf(stderr, "Too many command line arguments.\n");
    exit(-2);
    }

setup_colors();

main_frame = (Frame)xv_create((int)NULL, FRAME,
    FRAME_LABEL,            "fitX",
    WIN_CMS,                cms,
    NULL);
if (main_frame == 0){
    fprintf(stderr, "Allocation error for the main frame.\n");
    exit(-1);
    }
notify_interpose_destroy_func(main_frame, fit_destroy_func);

setup_graphics();

create_control_panel();
create_file_popups();
create_set_popups();
window_fit(main_frame);

bill_image = (Server_image)xv_create((int)NULL, SERVER_IMAGE,
    XV_WIDTH,               64,
    XV_HEIGHT,              64,
    SERVER_IMAGE_X_BITS,    bill_bits,
    NULL);
bill_mask_image = (Server_image)xv_create((int)NULL, SERVER_IMAGE,
    XV_WIDTH,               64,
    XV_HEIGHT,              64,
    XV_DEPTH,               1,
    SERVER_IMAGE_X_BITS,    bill_mask_bits,
    NULL);
bill_icon = (Icon)xv_create(main_frame, ICON,
    ICON_IMAGE,             bill_image,
    ICON_MASK_IMAGE,        bill_mask_image,
    NULL);
xv_set(main_frame, FRAME_ICON, bill_icon, NULL);

if (argc == 2){
    xv_set(fname_item, PANEL_VALUE, argv[1], NULL);
    file_open(fname_item, (Event *)NULL);
    }
window_main_loop(main_frame);
exit(0);
```

```
}


/*      create_control_panel() creates all of the panel items and
menus associated with the control panel.
*/

void create_control_panel()
{
int  i;
char  error[18];
char  alerstr[18];
Attr_avlist  list;
Panel_item  item, text_item;

dat_win = (Canvas)xv_create(main_frame, CANVAS,
        CANVAS_X_PAINT_WINDOW,      TRUE,
        CANVAS_AUTO_SHRINK,         FALSE,
        CANVAS_AUTO_EXPAND,         FALSE,
        CANVAS_WIDTH,               600,
        CANVAS_HEIGHT,              600,
        XV_WIDTH,                   600,
        XV_HEIGHT,                  600,
        NULL);
if (!dat_win){
        allocation_error("the data canvas.");
        xv_destroy(main_frame);
        exit(-1);
        }
xv_set(canvas_paint_window(dat_win),
        WIN_BACKGROUND_COLOR,       white,
        NULL);

strcpy(alerstr, "the control panel");

control_panel = (Panel)xv_create(main_frame, PANEL,
    PANEL_LAYOUT,           PANEL_VERTICAL,
    XV_X,                   600,
    XV_HEIGHT,              600,
    NULL);
if (control_panel == 0){
    sprintf(buffer, "%s.", alerstr);
    allocation_error(buffer);
    xv_destroy(main_frame);
    exit(-1);
    }

file_menu = (Menu)xv_create((int)NULL, MENU,
    MENU_ITEM,
```

```
        MENU_STRING,        "Open",
        MENU_NOTIFY_PROC,   file_open_proc,
        MENU_VALUE,         1,
        NULL,
    MENU_ITEM,
        MENU_STRING,        "Save",
        MENU_NOTIFY_PROC,   file_save_proc,
        MENU_VALUE,         2,
        MENU_INACTIVE,      TRUE,
        NULL,
    MENU_ITEM,
        MENU_STRING,        "Save As",
        MENU_NOTIFY_PROC,   file_saveas_proc,
        MENU_VALUE,         3,
        MENU_INACTIVE,      TRUE,
        NULL,
    MENU_ITEM,
        MENU_STRING,        "Close",
        MENU_NOTIFY_PROC,   file_close_proc,
        MENU_VALUE,         4,
        MENU_INACTIVE,      TRUE,
        NULL,
    MENU_ITEM,
        MENU_STRING,        "Quit",
        MENU_NOTIFY_PROC,   quit_proc,
        MENU_VALUE,         5,
        NULL,
    NULL);
if (file_menu == 0){
    allocation_error("the file menu.");
    xv_destroy(main_frame);
    exit(-1);
    }

set_menu = (Menu)xv_create((int)NULL, MENU,
    MENU_ITEM,
        MENU_STRING,        "Choose Fitting Function",
        MENU_NOTIFY_PROC,   func_proc,
        MENU_VALUE,         1,
        NULL,
    MENU_ITEM,
        MENU_STRING,        "Choose Response Function",
        MENU_NOTIFY_PROC,   conv_proc,
        MENU_VALUE,         2,
        NULL,
    MENU_ITEM,
        MENU_STRING,        "Set Parameters",
        MENU_NOTIFY_PROC,   flag_proc,
        MENU_VALUE,         3,
```

```
        NULL,
    MENU_ITEM,
        MENU_STRING,        "Set Baseline",
        MENU_NOTIFY_PROC,   ibase_proc,
        MENU_VALUE,         4,
        NULL,
    MENU_ITEM,
        MENU_STRING,        "Fit Setting",
        MENU_NOTIFY_PROC,   mc_set_proc,
        MENU_VALUE,         5,
        NULL,
    NULL);
if (set_menu == 0){
    allocation_error("the set menu.");
    xv_destroy(main_frame);
    exit(-1);
    }

item = (Panel_item)xv_create(control_panel, PANEL_BUTTON,
    PANEL_ITEM_COLOR,       black,
    PANEL_LABEL_STRING,     "File",
    PANEL_LABEL_WIDTH,      35,
    PANEL_ITEM_MENU,        file_menu,
    XV_X,                   20,
    NULL);
if (item == 0){
    sprintf(buffer, "%s file button.", alerstr);
    allocation_error(buffer);
    xv_destroy(main_frame);
    exit(-1);
    }

item = (Panel_item)xv_create(control_panel, PANEL_BUTTON,
    PANEL_ITEM_COLOR,       black,
    PANEL_LABEL_STRING,     "Set",
    PANEL_LABEL_WIDTH,      35,
    PANEL_ITEM_MENU,        set_menu,
    XV_X,                   20,
    NULL);
if (item == 0){
    sprintf(buffer, "%s set button.", alerstr);
    allocation_error(buffer);
    xv_destroy(main_frame);
    exit(-1);
    }

list = attr_create_list(PANEL_VALUE,    "0.0000",
    PANEL_VALUE_DISPLAY_LENGTH, 6,
    PANEL_VALUE_STORED_LENGTH,  6,
```

```
    PANEL_NOTIFY_PROC,          param_proc,
    WIN_CONSUME_EVENTS,         WIN_RIGHT_KEYS,
                                NULL,
    NULL);
if (list == 0){
    sprintf(buffer, "%s attribute list.", alerstr);
    allocation_error(buffer);
    xv_destroy(main_frame);
    exit(-1);
    }

sd_text = (Panel_item)xv_create(control_panel, PANEL_TEXT,
    ATTR_LIST,              list,
    PANEL_ITEM_COLOR,       fit_color,
    PANEL_LABEL_STRING,     param_string(0),
    PANEL_CLIENT_DATA,      (caddr_t)0,
    NULL);
if (sd_text == 0){
    sprintf(buffer, "%s sd item.", alerstr);
    allocation_error(buffer);
    free(list);
    xv_destroy(main_frame);
    exit(-1);
    }
def_event_proc = (void (*)())xv_get(sd_text, PANEL_EVENT_PROC);
xv_set(sd_text, PANEL_EVENT_PROC, keypad, NULL);

for(i=1; i<9; i++){
    text_item = (Panel_item)xv_create(control_panel, PANEL_TEXT,
        ATTR_LIST,          list,
        PANEL_ITEM_COLOR,   fit_color,
        PANEL_LABEL_STRING, param_string(i),
        PANEL_CLIENT_DATA,  (caddr_t)i,
        NULL);
    if (text_item == 0){
        sprintf(buffer, "%s text_item %d.", alerstr, i);
        allocation_error(buffer);
        free(list);
        xv_destroy(main_frame);
        exit(-1);
        }
    xv_set(text_item, PANEL_EVENT_PROC, keypad, NULL);
    }

t0_text = (Panel_item)xv_create(control_panel, PANEL_TEXT,
    ATTR_LIST,              list,
    PANEL_ITEM_COLOR,       t0_color,
    PANEL_LABEL_STRING,     param_string(9),
    PANEL_VALUE,            "1",
```

```
    PANEL_CLIENT_DATA,      (caddr_t) 9,
    NULL);
if (t0_text == 0){
    sprintf(buffer, "%s t0 item.", alerstr);
    allocation_error(buffer);
    free(list);
    xv_destroy(main_frame);
    exit(-1);
    }
xv_set(t0_text, PANEL_EVENT_PROC, keypad, NULL);

left_bnd_text = (Panel_item)xv_create(control_panel, PANEL_TEXT,
    ATTR_LIST,              list,
    PANEL_ITEM_COLOR,       resid_color,
    PANEL_LABEL_STRING,     param_string(10),
    PANEL_VALUE,            "1",
    PANEL_CLIENT_DATA,      (caddr_t) 10,
    NULL);
if (left_bnd_text == 0){
    sprintf(buffer, "%s left bound item.", alerstr);
    allocation_error(buffer);
    free(list);
    xv_destroy(main_frame);
    exit(-1);
    }
xv_set(left_bnd_text, PANEL_EVENT_PROC, keypad, NULL);

right_bnd_text = (Panel_item)xv_create(control_panel, PANEL_TEXT,
    ATTR_LIST,              list,
    PANEL_ITEM_COLOR,       resid_color,
    PANEL_LABEL_STRING,     param_string(11),
    PANEL_VALUE,            "1",
    PANEL_CLIENT_DATA,      (caddr_t) 11,
    NULL);
if (right_bnd_text == 0){
    sprintf(buffer, "%s right bound item.", alerstr);
    allocation_error(buffer);
    free(list);
    xv_destroy(main_frame);
    exit(-1);
    }
xv_set(right_bnd_text, PANEL_EVENT_PROC, keypad, NULL);
free(list);

sprintf(error, "%s = %.3E", param_string(12), lstsqer);
error_item = (Panel_item)xv_create(control_panel, PANEL_MESSAGE,
    PANEL_LABEL_STRING,     error,
    PANEL_ITEM_COLOR,       resid_color,
    NULL);
```

```
if (error_item == 0){
    sprintf(buffer, "%s least squares item.", alerstr);
    allocation_error(buffer);
    xv_destroy(main_frame);
    exit(-1);
    }

item = (Panel_item)xv_create(control_panel, PANEL_BUTTON,
    PANEL_LABEL_STRING,     " Fit ",
    PANEL_LABEL_WIDTH,      35,
    PANEL_NOTIFY_PROC,      mc_proc,
    XV_X,                   35,
    NULL);
if (item == 0){
    sprintf(buffer, "%s fit button.", alerstr);
    allocation_error(buffer);
    xv_destroy(main_frame);
    exit(-1);
    }

window_fit(control_panel);
}


/*      create_file_popups() creates the frames, panels, and items
pertaining to the file_menu choices.
*/

void create_file_popups()
{
Panel  file_open_panel, file_saveas_panel;
Panel_item  item;

file_open_frame = (Frame)xv_create(main_frame, FRAME_CMD,
    FRAME_LABEL,            "Open a Data File",
    NULL);
if (file_open_frame == 0){
    allocation_error("the file open frame.");
    xv_destroy(main_frame);
    exit(-1);
    }

file_open_panel = (Panel)xv_get(file_open_frame, FRAME_CMD_PANEL);
xv_set(file_open_panel, PANEL_LAYOUT, PANEL_VERTICAL, NULL);

item = (Panel_item)xv_create(file_open_panel, PANEL_MESSAGE,
    PANEL_LABEL_STRING,     "Enter the data file to be opened.",
    NULL);
if (item == 0){
```

```
    allocation_error("the file open message.");
    xv_destroy(main_frame);
    exit(-1);
    }

fname_item = (Panel_item)xv_create(file_open_panel, PANEL_TEXT,
    PANEL_VALUE_STORED_LENGTH,  50,
    PANEL_VALUE_DISPLAY_LENGTH, 50,
    PANEL_LABEL_STRING,         "Open: ",
    PANEL_NOTIFY_PROC,          file_open,
    NULL);
if (fname_item == 0){
    allocation_error("the file open text item.");
    xv_destroy(main_frame);
    exit(-1);
    }

window_fit(file_open_panel);
window_fit(file_open_frame);


file_saveas_frame = (Frame)xv_create(main_frame, FRAME_CMD,
    FRAME_LABEL,            "Save to a File",
    NULL);
if (file_saveas_frame == 0){
    allocation_error("the file save as frame.");
    xv_destroy(main_frame);
    exit(-1);
    }

file_saveas_panel = (Panel)xv_get(file_saveas_frame,FRAME_CMD_PANEL);
xv_set(file_saveas_panel, PANEL_LAYOUT, PANEL_VERTICAL,    NULL);

item = (Panel_item)xv_create(file_saveas_panel, PANEL_MESSAGE,
    PANEL_LABEL_STRING,     "Enter the file to which to save.",
    NULL);
if (item == 0){
    allocation_error("the file save as message.");
    xv_destroy(main_frame);
    exit(-1);
    }

comment_item = (Panel_item)xv_create(file_saveas_panel, PANEL_TEXT,
    PANEL_VALUE_STORED_LENGTH,  120,
    PANEL_VALUE_DISPLAY_LENGTH, 50,
    PANEL_LABEL_STRING,         "Comment: ",
    PANEL_VALUE,                "",
    NULL);
if (comment_item == 0){
```

```
    allocation_error("the comment text item.");
    xv_destroy(main_frame);
    exit(-1);
    }

sname_item = (Panel_item)xv_create(file_saveas_panel, PANEL_TEXT,
    PANEL_VALUE_STORED_LENGTH,  50,
    PANEL_VALUE_DISPLAY_LENGTH, 50,
    PANEL_LABEL_STRING,         "Save as: ",
    PANEL_NOTIFY_PROC,          file_saveas,
    PANEL_VALUE,                "",
    NULL);
if (sname_item == 0){
    allocation_error("the file save as text item.");
    xv_destroy(main_frame);
    exit(-1);
    }

window_fit(file_saveas_panel);
window_fit(file_saveas_frame);
}


/*      create_set_popups() creates the frames, panels, and items
pertaining to the set_menu choices.  This includes the frame, panel,
and item associated with electing to use response data.
*/

void create_set_popups()
{
Panel  func_panel, conv_panel, fcnv_open_panel, flag_panel;
Panel  ibase_panel, mc_set_panel;
Panel_item  item;

func_frame = (Frame)xv_create(main_frame, FRAME_CMD,
    FRAME_LABEL,                    "",
    NULL);
if (func_frame == 0){
    allocation_error("the function frame.");
    xv_destroy(main_frame);
    exit(-1);
    }

func_panel = (Panel)xv_get(func_frame, FRAME_CMD_PANEL);
xv_set(func_panel, PANEL_LAYOUT, PANEL_VERTICAL, NULL);

func_item = (Panel_item)xv_create(func_panel, PANEL_CHOICE,
    PANEL_LAYOUT,           PANEL_VERTICAL,
```

```
    PANEL_LABEL_STRING,     "Fitting Function:",
    PANEL_CHOICE_STRINGS,   function_string(1),
                            function_string(2),
                            NULL,
    PANEL_NOTIFY_PROC,      func_choose,
    XV_X,                   15,
    NULL);
if (func_item == 0){
    allocation_error("the function item.");
    xv_destroy(main_frame);
    exit(-1);
    }

item = (Panel_item)xv_create(func_panel, PANEL_BUTTON,
    PANEL_LABEL_STRING,     "Done",
    PANEL_LABEL_WIDTH,      35,
    PANEL_NOTIFY_PROC,      func_done_proc,
    XV_X,                   40,
    NULL);
if (item == 0){
    allocation_error("the function done button.");
    xv_destroy(main_frame);
    exit(-1);
    }

window_fit(func_panel);
window_fit(func_frame);

conv_frame = (Frame)xv_create(main_frame, FRAME_CMD,
    FRAME_LABEL,                    "",
    NULL);
if (conv_frame == 0){
    allocation_error("the convolution frame.");
    xv_destroy(main_frame);
    exit(-1);
    }

conv_panel = (Panel)xv_get(conv_frame, FRAME_CMD_PANEL);
xv_set(conv_panel, PANEL_LAYOUT, PANEL_VERTICAL, NULL);

conv_item = (Panel_item)xv_create(conv_panel, PANEL_CHOICE,
    PANEL_LAYOUT,           PANEL_VERTICAL,
    PANEL_LABEL_STRING,     "Convolving Function:",
    PANEL_CHOICE_STRINGS,   convolution_string(1),
                            convolution_string(2),
                            convolution_string(3),
                            convolution_string(4),
                            NULL,
    PANEL_NOTIFY_PROC,      conv_choose,
```

```
    XV_X,                   15,
    NULL);
if (conv_item == 0){
    allocation_error("the convolution item.");
    xv_destroy(main_frame);
    exit(-1);
    }

item = (Panel_item)xv_create(conv_panel, PANEL_BUTTON,
    PANEL_LABEL_STRING,     "Done",
    PANEL_LABEL_WIDTH,      35,
    PANEL_NOTIFY_PROC,      conv_done_proc,
    XV_X,                   55,
    NULL);
if (item == 0){
    allocation_error("the convolution done button.");
    xv_destroy(main_frame);
    exit(-1);
    }

window_fit(conv_panel);
window_fit(conv_frame);


fcnv_open_frame = (Frame)xv_create(conv_frame, FRAME_CMD,
    FRAME_LABEL,            "Open a Response File",
    NULL);
if (fcnv_open_frame == 0){
    allocation_error("the convolution file open frame.");
    xv_destroy(main_frame);
    exit(-1);
    }

fcnv_open_panel = (Panel)xv_get(fcnv_open_frame, FRAME_CMD_PANEL);
xv_set(fcnv_open_panel, PANEL_LAYOUT, PANEL_VERTICAL, NULL);

item = (Panel_item)xv_create(fcnv_open_panel, PANEL_MESSAGE,
    PANEL_LABEL_STRING, "Enter the convolution file to be opened.",
    NULL);
if (item == 0){
    allocation_error("the convolution file open message.");
    xv_destroy(main_frame);
    exit(-1);
    }

fcnv_name_item = (Panel_item)xv_create(fcnv_open_panel, PANEL_TEXT,
    PANEL_VALUE_STORED_LENGTH,  50,
    PANEL_VALUE_DISPLAY_LENGTH, 50,
    PANEL_LABEL_STRING,         "Open: ",
```

```
    PANEL_NOTIFY_PROC,          fcnv_open,
    NULL);
if (fcnv_name_item == 0){
    allocation_error("the convolution file open text item.");
    xv_destroy(main_frame);
    exit(-1);
    }

window_fit(fcnv_open_panel);
window_fit(fcnv_open_frame);


flag_frame = (Frame)xv_create(main_frame, FRAME_CMD,
    FRAME_LABEL,                    "",
    NULL);
if (flag_frame == 0){
    allocation_error("the flag frame.");
    xv_destroy(main_frame);
    exit(-1);
    }

flag_panel = (Panel)xv_get(flag_frame, FRAME_CMD_PANEL);
xv_set(flag_panel, PANEL_LAYOUT, PANEL_VERTICAL, NULL);

flag_item = (Panel_item)xv_create(flag_panel, PANEL_TOGGLE,
    PANEL_LAYOUT,           PANEL_VERTICAL,
    PANEL_LABEL_STRING,     "            Data Parameters:      ",
    PANEL_CHOICE_STRINGS,   "      picosecond data (else fs)   ",
                            "   fluorescence (else absorption) ",
                            "          bleach (else no)        ",
                            "autocorrelation data (else decays)",
                            "     weighted errors (else not)   ",
                            NULL,
    PANEL_TOGGLE_VALUE,     0,      TRUE,
    PANEL_TOGGLE_VALUE,     1,      TRUE,
    PANEL_NOTIFY_PROC,      flag_set,
    XV_X,                   10,
    NULL);
if (flag_item == 0){
    allocation_error("the flag item.");
    xv_destroy(main_frame);
    exit(-1);
    }

item = (Panel_item)xv_create(flag_panel, PANEL_BUTTON,
    PANEL_LABEL_STRING,     "Done",
    PANEL_LABEL_WIDTH,      35,
    PANEL_NOTIFY_PROC,      flag_done_proc,
    XV_X,                   90,
```

```
    NULL);
if (item == 0){
    allocation_error("the flag done button.");
    xv_destroy(main_frame);
    exit(-1);
    }

window_fit(flag_panel);
window_fit(flag_frame);


ibase_frame = (Frame)xv_create(main_frame, FRAME_CMD,
    FRAME_LABEL,                    "",
    NULL);
if (ibase_frame == 0){
    allocation_error("the baseline frame.");
    xv_destroy(main_frame);
    exit(-1);
    }

ibase_panel = (Panel)xv_get(ibase_frame, FRAME_CMD_PANEL);
xv_set(ibase_panel, PANEL_LAYOUT, PANEL_VERTICAL, NULL);

ibase_item = (Panel_item)xv_create(ibase_panel, PANEL_TEXT,
    PANEL_LAYOUT,               PANEL_VERTICAL,
    PANEL_LABEL_STRING,         "Last Point of Baseline =",
    PANEL_VALUE_STORED_LENGTH,  5,
    PANEL_VALUE_DISPLAY_LENGTH, 5,
    PANEL_NOTIFY_PROC,          ibase_set,
    NULL);
if (ibase_item == 0){
    allocation_error("the baseline text item.");
    xv_destroy(main_frame);
    exit(-1);
    }

window_fit(ibase_panel);
window_fit(ibase_frame);


mc_set_frame = (Frame)xv_create(main_frame, FRAME_CMD,
    FRAME_LABEL,                    "",
    NULL);
if (mc_set_frame == 0){
    allocation_error("the mc_set frame.");
    xv_destroy(main_frame);
    exit(-1);
    }
```

```
mc_set_panel = (Panel)xv_get(mc_set_frame, FRAME_CMD_PANEL);
xv_set(mc_set_panel, PANEL_LAYOUT, PANEL_VERTICAL, NULL);

mc_set_item = xv_create(mc_set_panel, PANEL_CHECK_BOX,
    PANEL_LAYOUT,           PANEL_VERTICAL,
    PANEL_LABEL_STRING,     "Parameters to Fit:",
    PANEL_CHOICE_STRINGS,   param_string(0),
                            param_string(1),
                            param_string(2),
                            param_string(3),
                            param_string(4),
                            param_string(5),
                            param_string(6),
                            param_string(7),
                            param_string(8),
                            NULL,
    PANEL_NOTIFY_PROC,      mc_set,
    PANEL_VALUE,            0,
    NULL);
if (mc_set_item == 0){
    allocation_error("the mc_set item.");
    xv_destroy(main_frame);
    exit(-1);
    }

item = (Panel_item)xv_create(mc_set_panel, PANEL_BUTTON,
    PANEL_LABEL_STRING,     "Done",
    PANEL_LABEL_WIDTH,      35,
    PANEL_NOTIFY_PROC,      mc_set_done_proc,
    XV_X,                   40,
    NULL);
if (item == 0){
    allocation_error("the mc_set done button.");
    xv_destroy(main_frame);
    exit(-1);
    }

window_fit(mc_set_panel);
window_fit(mc_set_frame);
}


/*      mc_proc() is the event procedure invoked by selecting the
"Fit" button on the control panel.
*/

void mc_proc(Panel_item item, Event *event)
{
Xv_notice  notice;
```

```
if (saved >= 0){
    if (!(data.flags & 8)){
        mc_flags = mc_flags & flag_mask;
        if(monte()){
            notice = (Xv_notice)xv_create(main_frame, NOTICE,
                NOTICE_MESSAGE_STRINGS, "Set parameters to fit",
                                        "and try again.",
                                        NULL,
                NOTICE_BUTTON_YES,      "Confirm",
                XV_SHOW,                TRUE,
                NULL);
            xv_destroy_safe(notice);
            }
        }

    else if (data.param[0]!=0.0 && data.param[5]!=0.0 &&
                                        data.param[7]!=0.0)
        fitt();
    else{
        notice = (Xv_notice)xv_create(conv_frame, NOTICE,
            NOTICE_MESSAGE_STRINGS, "Please enter initial estimates",
                                    "for FWHM, c1, and c3 and try again.",
                                    NULL,
            NOTICE_BUTTON_YES,      "Confirm",
            XV_SHOW,                TRUE,
            NULL);
        xv_destroy_safe(notice);
        }
    }
}


/*      file_open_proc() is the procedure invoked by choosing the
"Open" option from the file_menu.
*/

void file_open_proc(Menu menu, Menu_item menu_item)
{
Xv_notice  notice;
int  notice_stat;

/*      If a data set is already opened, close it.
*/

if (saved > 0){
    notice = (Xv_notice)xv_create(main_frame, NOTICE,
        NOTICE_MESSAGE_STRINGS, "File has changed since last save.",
                                "Do you want to save before opening?",
```

```
                                NULL,
        NOTICE_BUTTON_YES,      "Cancel open",
        NOTICE_BUTTON_NO,       "Open new data without saving old",
        NOTICE_STATUS,          &notice_stat,
        XV_SHOW,                TRUE,
        NULL);
    if (notice_stat == NOTICE_YES){
        xv_destroy_safe(notice);
        return;
        }
    xv_destroy_safe(notice);
    }
clear_window();
xv_set(file_open_frame, XV_SHOW, TRUE, NULL);
}


/*      file_saveas_proc() is the procedure invoked by choosing the
"Save As" option from the file_menu.
*/

void file_saveas_proc(Menu menu, Menu_item menu_item)
{
xv_set(file_saveas_frame, XV_SHOW, TRUE, NULL);
}


/*      file_save_proc() is the procedure invoked by choosing the
"Save" option from the file_menu.
*/

void file_save_proc(Menu menu, Menu_item menu_item)
{
file_save(1);
}


/*      file_close_proc() is the procedure invoked by choosing the
"Close" option from the file_menu.
*/

void file_close_proc(Menu menu, Menu_item menu_item)
{
file_close();
}


/*      func_proc() is the procedure invoked by choosing the "Choose
Fitting Function" option from the set_menu.
```

```
*/

void func_proc(Menu menu, Menu_item menu_item)
{
xv_set(func_frame, XV_SHOW, TRUE, NULL);
}


/*      conv_proc() is the procedure invoked by choosing the "Choose
Response Function" option from the set_menu.
*/

void conv_proc(Menu menu, Menu_item menu_item)
{
xv_set(conv_frame, XV_SHOW, TRUE, NULL);
}


/*      flag_proc() is the procedure invoked by choosing the "Set
Parameters" option from the set_menu.
*/

void flag_proc(Menu menu, Menu_item menu_item)
{
xv_set(flag_frame, XV_SHOW, TRUE, NULL);
}


/*      ibase_proc() is the procedure invoked by choosing the "Set
Baseline" option from the set_menu.
*/

void ibase_proc(Menu menu, Menu_item menu_item)
{
char  *cibase;

sprintf(cibase, "%d", ibase);
xv_set(ibase_item, PANEL_VALUE, cibase, NULL);
xv_set(ibase_frame, XV_SHOW, TRUE, NULL);
}


/*      mc_set_proc() is the procedure invoked by choosing the "Fit
Setting" option from the set_menu.
*/

void mc_set_proc(Menu menu, Menu_item menu_item)
{
mc_flags = mc_flags & flag_mask;
```

```
xv_set(mc_set_item, PANEL_VALUE, mc_flags, NULL);
xv_set(mc_set_frame, XV_SHOW, TRUE, NULL);
}


/*      file_saveas() is the notify procedure for the sname_item.
*/

Panel_setting file_saveas(Panel_setting item, Event *event)
{
file_save(0);
return(PANEL_NONE);
}


/*      func_choose() is the notify procedure for the func_item.  If
there is at least one open data window, it updates the window, the
pseudo-data, and all associated parameters and variables necessary to
reflect the newly selected fitting function.
*/

void func_choose(Panel_item item, int value, Event *event)
{
Xv_notice  notice;
char  temp_name[50];

if (data.flags & 8){
    xv_set(func_item, PANEL_VALUE, data.func-1, NULL);
    notice = (Xv_notice)xv_create(func_frame, NOTICE,
        NOTICE_MESSAGE_STRINGS, "For autocorrelations fit",
                                "with a convolution function.",
                                NULL,
        NOTICE_BUTTON_YES,      "Confirm",
        XV_SHOW,                TRUE,
        NULL);
    xv_destroy_safe(notice);
    return;
    }

data.func = value + 1;

if (saved >= 0){
    if (data.conv != 4)
        strcpy(temp_name, convolution_string(data.conv));
    else
        strcpy(temp_name, fcnv_name);

    sprintf(buffer, "%s - %s w/ %s", data.file_name,
            function_string(data.func), temp_name);
```

```
    xv_set(main_frame, XV_LABEL, buffer, NULL);

    set_pextrema_mats();
    calc();
    plot();
    }
}


/*      conv_choose() is the notify procedure for the conv_item.  If
there is an open data set, it updates the window, the pseudo-data,
and all associated parameters and variables necessary to reflect the
newly selected response function/data.
*/

void conv_choose(Panel_item item, int value, Event *event)
{
Xv_notice  notice;
char  *temp_name;

if (!(data.flags & 8) || value != 3)
    data.conv = value + 1;
else{
    xv_set(conv_item, PANEL_VALUE, data.conv-1, NULL);
    notice = (Xv_notice)xv_create(conv_frame, NOTICE,
        NOTICE_MESSAGE_STRINGS, "Autocorrelations cannot be",
                                "fit by input functions.",
                                NULL,
        NOTICE_BUTTON_YES,      "Confirm",
        XV_SHOW,                TRUE,
        NULL);
    xv_destroy_safe(notice);
    return;
    }

if (data.conv == 4)
    xv_set(fcnv_open_frame, XV_SHOW, TRUE, NULL);

if (saved >= 0){
    if (data.conv != 4)
        temp_name = convolution_string(data.conv);
    else if(!strcmp(fcnv_name, "")){
        data.conv = 1;
        temp_name = convolution_string(data.conv);
        }
    else
        strcpy(temp_name, fcnv_name);

    if (data.flags & 8)
```

```
        sprintf(buffer, "%s - %s", data.file_name, temp_name);
    else
        sprintf(buffer, "%s - %s w/ %s", data.file_name,
                  function_string(data.func), temp_name);
    xv_set(main_frame, XV_LABEL, buffer, NULL);

    if (data.flags & 8)
        calct0();
    else
        calc();
    plot();
    }
}


/*      flag_set() is the notify procedure for the flag_item.  If
there is an open data set, it updates the window, the pseudo-data,
and all associated parameters and variables necessary to reflect the
newly selected data parameters.
*/

void flag_set(Panel_item item, int value, Event *event)
{
if ((data.flags & 8) && !(value & 8)){
   xv_set((Menu_item)xv_find(set_menu, MENUITEM, MENU_VALUE, 1,NULL),
                    MENU_INACTIVE, FALSE, NULL);
   xv_set((Menu_item)xv_find(set_menu, MENUITEM, MENU_VALUE, 4,NULL),
                    MENU_INACTIVE, FALSE, NULL);
   }
if (!(data.flags & 1) && (value & 1))
   data.time_per_point *= 10.;
else if ((data.flags & 1) && !(value & 1))
   data.time_per_point /= 10.;
data.flags = value;

if (data.flags & 8){
   xv_set((Menu_item)xv_find(set_menu, MENUITEM, MENU_VALUE, 1,NULL),
                    MENU_INACTIVE, TRUE, NULL);
   xv_set((Menu_item)xv_find(set_menu, MENUITEM, MENU_VALUE, 4,NULL),
                    MENU_INACTIVE, TRUE, NULL);
   if (data.conv == 4)
        data.conv = 1;
   set_flag_mask();
   mc_flags = 99;
   }

if (saved >= 0){
   set_pextrema_mats();
   set_pdelta_mat();
```

```
   if (data.flags & 8)
        calct0();
   else
        calc();
   plot();
   }
}


/*      ibase_set() is the notify procedure for the ibase_item.
*/

Panel_setting ibase_set(Panel_item item, Event *event)
{
Xv_notice  notice;

if (!(data.flags & 2)){
    sprintf(buffer, "%d", ibase);
    xv_set(ibase_item, PANEL_VALUE, buffer, NULL);
    notice = (Xv_notice)xv_create(conv_frame, NOTICE,
        NOTICE_MESSAGE_STRINGS, "Baseline adjusted only",
                                "for fluorescence data.",
                                NULL,
        NOTICE_BUTTON_YES,      "Confirm",
        XV_SHOW,                TRUE,
        NULL);
    xv_destroy_safe(notice);
    return(PANEL_NONE);
    }
ibase = atoi((char *)xv_get(ibase_item, PANEL_VALUE));
if (ibase <= 0 || ibase > data.points_scan && saved >= 0){
    strcpy(buffer, "1");
    xv_set(ibase_item, PANEL_VALUE, buffer, NULL);
    return(PANEL_NONE);
    }
if (saved >= 0){
    baseline_adjust();
    calc_err();
    plot();
    }
return(PANEL_NONE);
}


/*      mc_set() is the notify procedure for the mc_set_item.
*/

void mc_set(Panel_item item, unsigned value, Event *event)
{
```

```
if (value != (value & flag_mask)){
    value = value & flag_mask;
    xv_set(mc_set_item, PANEL_VALUE, value, NULL);
    }
mc_flags = value;
}


/*     func_done_proc() closes the func_frame when depressed.
*/

void func_done_proc(Panel_item item, Event *event)
{
xv_set(func_frame, XV_SHOW, FALSE, NULL);
}


/*     conv_done_proc() closes the func_frame when depressed.
*/

void conv_done_proc(Panel_item item, Event *event)
{
xv_set(conv_frame, XV_SHOW, FALSE, NULL);
}


/*     flag_done_proc() closes the func_frame when depressed.
*/

void flag_done_proc(Panel_item item, Event *event)
{
xv_set(flag_frame, XV_SHOW, FALSE, NULL);
}


/*     mc_set_done_proc() closes the func_frame when depressed.
*/

void mc_set_done_proc(Panel_item item, Event *event)
{
xv_set(mc_set_frame, XV_SHOW, FALSE, NULL);
}


/*      quit_proc is called to terminate the program.
*/

void quit_proc(Menu menu, Menu_item menu_item)
{
```

```
xv_destroy_safe(main_frame);
}


/*      fit_destroy_func is the function interposed on the
destruction of the main_frame.  It frees as necessary any dynamically
allocated dat_set structs and any dynamically allocated .data
elements of dat_set structs, including that of the struct data.  It
also frees any allocated memory associated with the pointer cnv.
*/

static Notify_value fit_destroy_func(Notify_client client,
                        Destroy_status status)
{
int  i, j;

/*if (status == DESTROY_CHECKING && confirm_quit() == 0){
    notify_veto_destroy(client);
    return(NOTIFY_DONE);
    }
else*/
if (status == DESTROY_PROCESS_DEATH || status == DESTROY_CLEANUP){
    for (i=0; i<4; i++)
        if (data.data[i] != NULL)
            free(data.data[i]);
    }
return(notify_next_destroy_func(client, status));
}
```

## B.2 fitaux.c

```
/*      This file contains the following routines:
allocation_error(), param_string(), function_string(),
convolution_string(), keypad(), adjust_value(), param_proc(),
file_open(), fcnv_open(), file_save(), set_menu_greyed(),
reset_control_panel(), set_flag_mask(), and file_close().
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/notice.h>
#include <xview/cms.h>
#include <X11/Xlib.h>

#include "fitX.h"

extern Frame  main_frame, file_open_frame, file_saveas_frame,
        fcnv_open_frame;
extern Panel  control_panel;
extern Menu  file_menu, set_menu;
extern Panel_item  fname_item, sname_item, comment_item;
extern Panel_item  fcnv_name_item, sd_text, t0_text;
extern Panel_item  left_bnd_text, right_bnd_text, error_item;
extern Panel_item  func_item, conv_item, flag_item;
extern Canvas  dat_win;
extern Cms  cms;

extern struct dat_set  data;
extern int  saved, flag_mask, mc_flags, cnvpts, resp_step, white;
extern float  pdelta[], pmax[], pmin[];
extern char  buffer[], fcnv_name[], names[];
extern double  *cnv, lstsqer;

extern void  (*def_event_proc)();

/*      allocation_error() is called if and when a dynamic memory
allocation error occurs.  It is passed a string indicating the item
for which allocation failed and activates an appropriate alert
window.
*/

void allocation_error(char *serror)
{
Xv_notice  notice;
```

```
notice = (Xv_notice)xv_create(main_frame, NOTICE,
    NOTICE_MESSAGE_STRINGS, "Allocation error for:",
                            serror,
                            "",
                            "Exiting",
                            NULL,
    NOTICE_BUTTON_YES,      "Confirm",
    XV_SHOW,                TRUE,
    NULL);

xv_destroy_safe(notice);
}


/*      The following three routines, param_string(),
function_string(), and convolution_string(), return a constant
string as indicated by the integer argument n.  Values of n that
are too great or too small result in the return of some default
string.
*/

char *param_string(int n)
{
static char  *string[] = {
        "FWHM = ",
        "a1 = ",
        "a2 = ",
        "b1 = ",
        "b2 = ",
        "c1 = ",
        "c2 = ",
        "c3 = ",
        "c4 = ",
        "t0 = ",
        "left = ",
        "right = ",
        "error"
        };
return((n < 0 || n > 11) ? string[12] : string[n]);
}


char *function_string(int n)
{
static char  *string[] = {
        "error",
        " Biexponential ",
        "  Kohlrausch   "
```

```
        };
return((n < 1 || n > 2) ? string[0] : string[n]);
}


char *convolution_string(int n)
{
static char  *string[] = {
        "error",
        "Gaussian",
        "Sech Squared",
        "Lorentzian",
        "Input File"
        };
return((n < 1 || n > 4) ? string[0] : string[n]);
}


/*      keypad() is the event procedure for the text items on the
control panel.  It is designed to handle the events associated with
the right-hand keys, that is those on the keypad with num-lock off.
In particular, the following actions should occur.  Arrow up (down)
should move the caret up (down) one item.  Right (left) arrow should
increase (decrease) the current item by an amount as indicated in the
array pdelta, which is indexed identically to the parameters.  The
page up (page down) and home (end) increase (decrease) the values by
10 or 100 times the base value in pdelta.  Note that in order to
handle other events, the default event procedure for text items had
to be obtained in create_control_panel(); a pointer to this function
is stored in def_event_proc().
*/

void keypad(Panel_item pitem, Event *event)
{
int  p, id, keycode;
char  disp[10];
XEvent  *xev;

if (saved == -1)
    return(def_event_proc(pitem, event));
id = event_id(event);
xev = event_xevent(event);

if (id==46 || id<32 && id>0 || id>=48 && id<=57 || id==127)
    return(def_event_proc(pitem, event));
else if (xev->type == KeyPress){
    p = (int)xv_get(pitem, PANEL_CLIENT_DATA);

    keycode = ((XKeyEvent*)xev)->keycode;
```

```
if (id == 0){
    strncpy(disp, DisplayString(((XKeyEvent *)xev)->display), 4);
    disp[4] = '\0';

    if (!strcmp(disp, "hoek")){
        switch(keycode){
        case 98:
            id = 32596;
            break;
        case 104:
            id = 32602;
            break;
        case 102:
            id = 32600;
            break;
        case 100:
            id = 32598;
            break;
        case 99:
            id = 32597;
            break;
        case 105:
            id = 32603;
            break;
        case 97:
            id = 32595;
            break;
        case 103:
            id = 32601;
            break;
            }
        }
    else{
        switch(keycode){
        case 76:
            id = 32596;
            break;
        case 120:
            id = 32602;
            break;
        case 100:
            id = 32600;
            break;
        case 98:
            id = 32598;
            break;
        case 77:
            id = 32597;
            break;
```

```
            case 121:
                id = 32603;
                break;
            case 75:
                id = 32595;
                break;
            case 119:
                id = 32601;
                break;
                }
            }
        }

    switch(id){
    case 77:
    case 109:
    case KEY_RIGHT(14):
        panel_advance_caret(control_panel);
        return;
    case 85:
    case 117:
    case KEY_RIGHT(8):
        panel_backup_caret(control_panel);
        return;
    case 75:
    case 107:
    case KEY_RIGHT(12):
        adjust_value(p, pitem, pdelta[p]);
        break;
    case 72:
    case 104:
    case KEY_RIGHT(10):
        adjust_value(p, pitem, -pdelta[p]);
        break;
    case 73:
    case 105:
    case KEY_RIGHT(9):
        adjust_value(p, pitem, 10*pdelta[p]);
        break;
    case 44:
    case 60:
    case KEY_RIGHT(15):
        adjust_value(p, pitem, -10*pdelta[p]);
        break;
    case 89:
    case 121:
    case KEY_RIGHT(7):
        adjust_value(p, pitem, 100*pdelta[p]);
        break;
```

```
    case 78:
    case 110:
    case KEY_RIGHT(13):
        adjust_value(p, pitem, -100*pdelta[p]);
        break;
    default:
        return;
        }
    if (saved == 0)
        saved = 2;
    if (data.flags & 8)
        calct0();
    else
        calc();
    plot();
    return;
    }
}


/*      adjust_value() is called by keypad() to change the value of
the parameter associated with the control panel item pitem, which
has an integer p associated with it.  That parameter is adjusted by
adjustment provided such a change does not place its value outside
of the acceptable limits (indicated by the identically indexed arrays
pmax[] and pmin[] for FWHM, a1-c4).
*/

void adjust_value(p, pitem, adjustment)
int p;
Panel_item pitem;
float adjustment;
{

if (p == 9){
    data.tzero += (int)adjustment;
    if (data.tzero > data.points_scan)
        data.tzero = data.points_scan;
    else if (data.tzero < 1)
        data.tzero = 1;
    sprintf(buffer, "%d", data.tzero);
    xv_set(pitem, PANEL_VALUE, buffer, NULL);
    return;
    }
else if (p == 10){
    data.left_bnd += (int)adjustment;
    if (data.left_bnd > data.right_bnd)
        data.left_bnd = data.right_bnd;
    else if (data.left_bnd < 1)
```

```
        data.left_bnd = 1;
    sprintf(buffer, "%d", data.left_bnd);
    xv_set(pitem, PANEL_VALUE, buffer, NULL);
    return;
    }
else if (p == 11){
    data.right_bnd += (int)adjustment;
    if (data.right_bnd > data.points_scan)
        data.right_bnd = data.points_scan;
    else if (data.right_bnd < data.left_bnd)
        data.right_bnd = data.left_bnd;
    sprintf(buffer, "%d", data.right_bnd);
    xv_set(pitem, PANEL_VALUE, buffer, NULL);
    return;
    }
else if (p >= 0 && p < 9){
    data.param[p] += adjustment;
    if (data.param[p] > pmax[p])
        data.param[p] = pmax[p];
    else if (data.param[p] < pmin[p])
        data.param[p] = pmin[p];
    sprintf(buffer, "%f", data.param[p]);
    xv_set(pitem, PANEL_VALUE, buffer, NULL);
    return;
    }
}


/*      param_proc() is the notify procedure for the panel text
items.  It handles keyboard entry of values for those items and
ensures the input values fall within the acceptable limits.
*/

Panel_setting param_proc(Panel_item item, Event *event)
{
int  p, itemp;
float  param_temp;

if (saved >= 0){
    p = (int)xv_get(item, PANEL_CLIENT_DATA);
    if (p == 9){
        itemp = atoi((char *)xv_get(item, PANEL_VALUE));
        if (itemp < 1 || itemp > data.points_scan){
            sprintf(buffer, "%d", data.tzero);
            xv_set(item, PANEL_VALUE, (caddr_t)buffer, NULL);
            return(PANEL_NONE);
            }
        data.tzero = itemp;
        }
```

```
    else if (p == 10){
        itemp = atoi((char *)xv_get(item, PANEL_VALUE));
        if (itemp < 1 || itemp > data.right_bnd){
            sprintf(buffer, "%d", data.left_bnd);
            xv_set(item, PANEL_VALUE, (caddr_t)buffer, NULL);
            return(PANEL_NONE);
            }
        data.left_bnd = itemp;
        }
    else if (p == 11){
        itemp = atoi((char *)xv_get(item, PANEL_VALUE));
        if (itemp < data.left_bnd || itemp > data.points_scan){
            sprintf(buffer, "%d", data.right_bnd);
            xv_set(item, PANEL_VALUE, (caddr_t)buffer, NULL);
            return(PANEL_NONE);
            }
        data.right_bnd = itemp;
        }
    else if (p >= 0 && p < 9){
        param_temp = atof((char *)xv_get(item, PANEL_VALUE));
        if (param_temp > pmax[p] || param_temp < pmin[p]){
            sprintf(buffer, "%f", data.param[p]);
            xv_set(item, PANEL_VALUE, (caddr_t)buffer, NULL);
            return(PANEL_NONE);
            }
        data.param[p] = param_temp;
        }
    if (data.flags & 8)
        calct0();
    else
        calc();
    plot();

    if (saved == 0)
        saved = 2;
    }
return(panel_text_notify(item, event));
}


/*      file_open() is the notify procedure for the text item
fname_item.  It carries out many important tasks as indicated below.
*/

Panel_setting file_open(Panel_item item, Event *event)
{
Xv_notice  notice;

FILE  *stream;
```

```
int  i, j, idum;
time_t  ltime;
long  ldum;
char  ch, *temp_name;
double  dtemp;

/*      The entered file name is copied into a buffer to check
its validity.  If the string is empty, the window is dismissed;
and file_open() returns.  Otherwise, the buffer is copied into
the file_name field of the working data structure data; and the
the file is checked to ensure its existence as a readable file.
If it does not exist, a notice is sent after the confirmation of
which the window is dismissed and file_open() returns.
*/

strcpy(buffer, (char *)xv_get(fname_item, PANEL_VALUE));
if (!strcmp(buffer, "")){
    xv_set(file_open_frame, XV_SHOW, FALSE, NULL);
    return(PANEL_NONE);
    }
strcpy(data.file_name, buffer);

if ((stream = fopen(data.file_name, "rb")) == NULL){
    notice = (Xv_notice)xv_create(file_open_frame, NOTICE,
        NOTICE_MESSAGE_STRINGS, "File does not exist!",
                                NULL,
        NOTICE_BUTTON_YES,      "Confirm",
        XV_SHOW,                TRUE,
        NULL);
    xv_set(file_open_frame, XV_SHOW, FALSE, NULL);
    xv_destroy_safe(notice);
    return(PANEL_NONE);
    }

/*      At this point the file input via the fname_item in the
file_open_frame is known to exist as a readable file.  The first
three characters are then read to verify its nature as a data file
appropriate for this program as well as a few specifics about its
nature as such.  Acceptable strings and their meanings are:
"gad" -> picosecond fluorescence; "djr" -> picosecond absorption;
"bjs" -> femtosecond absorption.  Other acceptable parameters
(bleach or not, weighted or not, a response function instead of a
decay) are masked to remain on if they are valid for the new data
set.  Note, however, that this does not ensure their correctness.
        If the file is not an appropriate data file, the file is
closed; a notice is sent; and, after the confirmation of the notice,
the window is dismissed; and file_open() returns.
        Otherwise the window is dismissed and data are read into the
appropriate variables.  Note that some of the values are not needed
```

```
and are therefore read into dummy variables *dum or buffer[]. The
time_per_point element of data is calculated here. Also the memory
for the data is allocated. This will require freeing upon later
closing the data window and/or destruction. After the data are all
read in, the file is closed.
*/

for (i=0; i!=3; i++)
    data.test[i] = (char)fgetc(stream);
data.test[i] = '\0';

if (strcmp(data.test, "gad") == 0)
    data.flags = (data.flags | 3) & 27;
else if (strcmp(data.test, "djr") == 0)
    data.flags = (data.flags | 1) & 13;
else if (strcmp(data.test, "bjs") == 0)
    data.flags = data.flags & 12;
else{
    fclose(stream);
    notice = (Xv_notice)xv_create(file_open_frame, NOTICE,
        NOTICE_MESSAGE_STRINGS, "This is not a proper data file!",
                                NULL,
        NOTICE_BUTTON_YES,      "Confirm",
        XV_SHOW,                TRUE,
        NULL);
    xv_set(file_open_frame, XV_SHOW, FALSE, NULL);
    xv_destroy_safe(notice);
    return(PANEL_NONE);
    }

xv_set(flag_item, PANEL_VALUE, data.flags, NULL);

xv_set(file_open_frame, XV_SHOW, FALSE, NULL);

fscanf(stream, " %li ", &ltime);
fscanf(stream, "%i %i ", &data.points_scan, &data.stage_step);
data.time_per_point = (float)abs(data.stage_step) * 20.0/299.8;
if (!(data.flags & 1))
    data.time_per_point /= 10;
fscanf(stream, "%s %li ", buffer, &ldum);
fscanf(stream, "%i %i %i ", &idum, &idum, &idum);

for (i=0; (i<498) && (ch=fgetc(stream)) != '~'; i++)
    buffer[i] = (char) ch;
ch = fgetc(stream);
buffer[i] = (char) '\0';

for (i=0; i<4; i++){
    data.data[i] = (double *)malloc(data.points_scan*sizeof(double));
```

```
    if (data.data[i] == NULL){
        allocation_error("the loaded data.");
        xv_destroy_safe(main_frame);
        exit(-1);
        }
    }

for (i=0; i<data.points_scan; i++){
    fscanf(stream, "%lf\n", &dtemp);
    *(data.data[0] + i) = dtemp;
    for (j=1; j<4; j++)
        *(data.data[j] + i) = 0.0;
    }

fscanf(stream, " %s", buffer);
fclose(stream);

/*      Having all been read in, the data are initialized (including
setting the values of the arrays pdelta, pmax, and pmin - see also
initialize()).  Then the appropriate label for the new data is
determined and the saveas and close options in the file menu are
activated if inactive.
*/

initialize();

if (data.conv != 4)
    temp_name = convolution_string(data.conv);
else
    temp_name = fcnv_name;

if (data.flags & 8)
    sprintf(buffer, "%s - %s", data.file_name, temp_name);
else
    sprintf(buffer, "%s - %s w/ %s", data.file_name,
                function_string(data.func), temp_name);

xv_set(main_frame, XV_LABEL, buffer, NULL);

if (saved == -1){
    xv_set((Menu_item)xv_find(file_menu,MENUITEM,MENU_VALUE,3,NULL),
            MENU_INACTIVE,      FALSE,
            NULL);
    xv_set((Menu_item)xv_find(file_menu,MENUITEM,MENU_VALUE,4,NULL),
            MENU_INACTIVE,      FALSE,
            NULL);
    }
saved = 1;
```

```
/*      Finally an initial plot of the data is made before
file_open() returns.
*/

initial_plot();
return(PANEL_NONE);
}


/*      fcnv_open() plays the same role for opening an input
response function for convolution with theoretical decay functions
as file_open() does for decay data.
*/

Panel_setting fcnv_open(Panel_item item, Event *event)
{
Xv_notice  notice;
char  ch, rtest[4], *temp_name;
int  i, j, dumi;
long int  duml;
FILE  *stream;
time_t  ltime;
double  dtemp;

/*      First fcnv_open ensures that the current data are not
intended to be fit as response functions themselves.  Such data are
not fit by the same techniques and require the analytic functions
provided.  If the current data are indicated to be response functions
(the fourth bit of data.flags on), a notice is sent after the
confirmation of which the window is dismissed if unpinned and
fcnv_open() returns.
*/

if (data.flags & 8){
    notice = (Xv_notice)xv_create(fcnv_open_frame, NOTICE,
        NOTICE_MESSAGE_STRINGS,        "Autocorrelations cannot be",
                        "fit by input functions.",
                        NULL,
        NOTICE_BUTTON_YES,        "Confirm",
        XV_SHOW,            TRUE,
        NULL);
    xv_destroy_safe(notice);
    xv_set(fcnv_open_frame, XV_SHOW, FALSE, NULL);
    return(PANEL_NONE);
    }

/*      As in file_open(), fcnv_open() returns if given an empty
string via the fcnv_name item or if the file does not exist as a
readable file.  Also as in file_open(), in the latter case, a notice
```

```
is sent; and in both cases the window is dismissed if unpinned.
*/

strcpy(fcnv_name, (char *)xv_get(fcnv_name_item, PANEL_VALUE));
if (!strcmp(fcnv_name, "")){
    xv_set(fcnv_open_frame, XV_SHOW, FALSE, NULL);
    return(PANEL_NONE);
    }

if ((stream = fopen(fcnv_name, "rb")) == NULL){
    notice = (Xv_notice)xv_create(fcnv_open_frame, NOTICE,
        NOTICE_MESSAGE_STRINGS, "File does not exist!",
                                NULL,
        NOTICE_BUTTON_YES,      "Confirm",
        XV_SHOW,                TRUE,
        NULL);
    strcpy(fcnv_name, "");
    xv_set(fcnv_open_frame, XV_SHOW, FALSE, NULL);
    xv_destroy_safe(notice);
    return(PANEL_NONE);
    }

/*      Given that the file is indeed a readable file, the first
three characters are checked to ensure that the file is acceptable.
At the present time, only files beginning with "gad" are acceptable.
All others cause the file to close and cause fcnv_open() to send a
notice following confirmation of which the window is dismissed if
unpinned after which fcnv_open() returns.
        Otherwise, the window is dismissed; and the data are read
into the appropriate variables.  Some are read into dummy variables
*dum and buffer, as they are not needed.  Memory allocation is
carried out for the response data here as well.  The file is then
closed.
*/

for (i=0; i!=3; i++)
    rtest[i] = (char)fgetc(stream);
rtest[i] = '\0';

if (strcmp(rtest, "gad") != 0){
    fclose(stream);
    notice = (Xv_notice)xv_create(fcnv_open_frame, NOTICE,
        NOTICE_MESSAGE_STRINGS,"This is not a proper response file!",
                                NULL,
        NOTICE_BUTTON_YES,      "Confirm",
        XV_SHOW,                TRUE,
        NULL);
    strcpy(fcnv_name, "");
    xv_set(fcnv_open_frame, XV_SHOW, FALSE, NULL);
```

```
    xv_destroy_safe(notice);
    return(PANEL_NONE);
    }
xv_set(fcnv_open_frame, XV_SHOW, FALSE, NULL);

fscanf(stream, " %li ", &ltime);
fscanf(stream, "%i %i ", &cnvpts, &resp_step);
fscanf(stream, "%s %li ", buffer, &duml);
fscanf(stream, "%i ", &dumi);
fscanf(stream, "%i ", &dumi);
fscanf(stream, "%i ", &dumi);

for (i=0; (i<498) && (ch=fgetc(stream)) != '~'; i++)
    buffer[i] = (char) ch;
ch = fgetc(stream);
buffer[i] = (char) '\0';

for (i=0; i<4; i++){
    if (cnv != NULL)
        free(cnv);
    cnv = (double *)malloc(cnvpts * sizeof(double));
    if (cnv == NULL){
        allocation_error("the loaded response.");
        xv_destroy_safe(main_frame);
        exit(-1);
        }
    }

for (i=0; i<cnvpts; i++){
    fscanf(stream, "%lf\n", &dtemp);
    *(cnv + i) = dtemp;
    }

fscanf(stream, " %s", buffer);
fclose(stream);

/*      Finally, prior to returning, fcnv_name item is truncated to
remove the path if present and the response data are preprocessed by
resp_prep() prior to their use in calculations.
*/

temp_name = strrchr(fcnv_name, (int)'/');
if (temp_name != NULL){
    strcpy(fcnv_name, temp_name+1);
    xv_set(main_frame, XV_LABEL, buffer, NULL);
    }

resp_prep();
```

```
return(PANEL_NONE);
}


/*      File save is the routine called to save files to new file
names as well as old ones as indicated by the value of the parameter
again.  If again = 0, a new file name is provided by the sname_item;
on the other hand, if again = 1, the data are saved to
data.file_name.
*/

void file_save(int again)
{
Xv_notice  notice;
char  ch, hdr[54], *temp_name;
int  i, notice_stat;
float fpt;
FILE  *stream;

/*      For new file names, file_save() first ensures a non-null
string, returning otherwise after dismissing the window.  Then it
checks to see if the file has changed since its last save.  If not,
it sends a notice, obtaining from the user instructions to cancel or
to continue.  If cancelling, the window is first dismissed.  Then the
file is checked to see if it already exists as a readable file
(assuming that if it exists, it is readable).  If it does, a notice
is sent to the user requesting confirmation to cancel or overwrite.
Again, if cancelling, the window is first dismissed.
        If at this point the file does not exist (at least not as a
readable file) or overwriting has been confirmed, the window is
dismissed; and data.file_name is updated to the new file name.
*/

if (!again){
    strcpy(buffer, (char *)xv_get(sname_item, PANEL_VALUE));
    if (!strcmp(buffer, "")){
        xv_set(file_saveas_frame, XV_SHOW, FALSE, NULL);
        return;
        }

    if (saved == 0){
        notice = (Xv_notice)xv_create(file_saveas_frame, NOTICE,
            NOTICE_MESSAGE_STRINGS, "File has not changed.",
                                    "Do you want to save anyway?",
                                    NULL,
            NOTICE_BUTTON_YES,      "No",
            NOTICE_BUTTON_NO,       "Yes",
            NOTICE_STATUS,          &notice_stat,
            XV_SHOW,                TRUE,
```

```
            NULL);
        if (notice_stat == NOTICE_YES){
            xv_destroy_safe(notice);
            xv_set(file_saveas_frame, XV_SHOW, FALSE, NULL);
            return;
            }
        xv_destroy_safe(notice);
        }

    if ((stream = fopen(buffer, "rb")) != NULL){
        fclose(stream);
        notice = (Xv_notice)xv_create(file_saveas_frame, NOTICE,
            NOTICE_MESSAGE_STRINGS, "File already exists!",
                                    "Do you want to overwrite?",
                                    NULL,
            NOTICE_BUTTON_YES,      "No",
            NOTICE_BUTTON_NO,       "Yes",
            NOTICE_STATUS,          &notice_stat,
            XV_SHOW,                TRUE,
            NULL);
        if (notice_stat == NOTICE_YES){
            xv_set(file_saveas_frame, XV_SHOW, FALSE, NULL);
            xv_destroy_safe(notice);
            return;
            }
        xv_destroy_safe(notice);
        }
    xv_set(file_saveas_frame, XV_SHOW, FALSE, NULL);
    strcpy(data.file_name, buffer);
    }

/*      If the data are to be saved to the same file as previously
and no changes have occurred since the last save, a notice is sent
and no save takes place.
*/

else if (saved == 0){
    notice = (Xv_notice)xv_create(main_frame, NOTICE,
        NOTICE_MESSAGE_STRINGS, "File has not changed.",
                                "No new save will occur.",
                                NULL,
        NOTICE_BUTTON_YES,      "Confirm",
        XV_SHOW,                TRUE,
        NULL);
    xv_destroy_safe(notice);
    return;
    }

/*      Since saving is to take place, the data are written to the
```

```
file indicated (after opening for writing).  Also some of the
important information such as the parameters of the fit are saved to
a separate header file with the same file name except for an
additional .hdr extension.  At the conclusion of writing each file,
it is closed.
*/

fpt = (float)(1 - data.tzero) * data.time_per_point;

if((stream = fopen(data.file_name, "wb")) == NULL){
    notice = (Xv_notice)xv_create(file_saveas_frame, NOTICE,
        NOTICE_MESSAGE_STRINGS, "File is not writeable!",
                                "Try another file name.",
                                NULL,
        NOTICE_BUTTON_YES,      "Confirm",
        XV_SHOW,                TRUE,
        NULL);
    xv_destroy_safe(notice);
    return;
    }

for (i=0; i<data.points_scan; i++){
    fprintf(stream, "%f ", fpt + i*data.time_per_point);
    fprintf(stream, "%lf %lf\n",*(data.data[0]+i),*(data.data[1]+i));
    }
fclose(stream);

strcpy(hdr, data.file_name);
strcat(hdr, ".hdr");
stream = fopen(hdr, "wb");

temp_name=(char *)xv_get((Frame)xv_get(dat_win,XV_OWNER),FRAME_LABEL);
fprintf(stream, "%s\n", temp_name);

fprintf(stream, "number of points = %i\n", data.points_scan);

fprintf(stream, "time per point = %f ps, first point = %f ps\n",
        data.time_per_point, fpt);
fprintf(stream, "zero time point = %i, FWHM = %f ps\n",
        data.tzero, data.param[0]);
fprintf(stream, "a1 = %f, a2 = %f, b1 = %f, b2 = %f\n",
        data.param[1], data.param[2], data.param[3], data.param[4]);
fprintf(stream, "c1 = %f, c2 = %f, c3 = %f, c4 = %f\n",
        data.param[5], data.param[6], data.param[7], data.param[8]);
fprintf(stream, "error = %lf from %d through %d\n",
        lstsqer, data.left_bnd, data.right_bnd);

strcpy(buffer, (char *)xv_get(comment_item, PANEL_VALUE));
fprintf(stream, "%s\n", buffer);
```

```
fclose(stream);

/*      Finally, the apporpriate entry in saved[] is updated to
reflect the nature of the current data as unchanged since saving.
Also the save menu option is inactivated, and the data frame label
is updated.
*/

saved = 0;
if (!again)
    xv_set((Menu_item)xv_find(file_menu,MENUITEM,MENU_VALUE,2,NULL),
        MENU_INACTIVE,      FALSE,
        NULL);


if (data.conv != 4)
    temp_name = convolution_string(data.conv);
else
    temp_name = fcnv_name;

sprintf(buffer, "%s - %s w/ %s", data.file_name,
                            function_string(data.func), temp_name);
xv_set(main_frame, XV_LABEL, buffer, NULL);
return;
}


/*      set_menu_greyed() is used to activate or deactivate the
"Save" and "Save As" options of the file_menu as dictated by its
save status as stored in save.
*/

void set_menu_greyed()
{
if (saved == 1){
    xv_set((Menu_item)xv_find(file_menu,MENUITEM,MENU_VALUE,2,NULL),
        MENU_INACTIVE,      TRUE,
        NULL);
    xv_set((Menu_item)xv_find(file_menu,MENUITEM,MENU_VALUE,3,NULL),
        MENU_INACTIVE,      FALSE,
        NULL);
    }
else{
    xv_set((Menu_item)xv_find(file_menu,MENUITEM,MENU_VALUE,2,NULL),
        MENU_INACTIVE,      FALSE,
        NULL);
    xv_set((Menu_item)xv_find(file_menu,MENUITEM,MENU_VALUE,3,NULL),
        MENU_INACTIVE,      FALSE,
```

```
        NULL);
    }
}


/*      reset_control_panel() updates the values displayed on the
control panel to ensure agreement with the current values of the
associated variables.
*/

void reset_control_panel()
{
int  i;
Panel_item  item, next_item;

sprintf(buffer, "%f", data.param[0]);

xv_set(sd_text, PANEL_VALUE, (caddr_t)buffer, NULL);
item = sd_text;

for (i=1; i<9; i++){
    next_item = (Panel_item)xv_get(item, PANEL_NEXT_ITEM);
    item = next_item;
    sprintf(buffer, "%f", data.param[i]);
    xv_set(item, PANEL_VALUE, (caddr_t)buffer, NULL);
    }
sprintf(buffer, "%d", data.tzero);
xv_set(t0_text, PANEL_VALUE, (caddr_t)buffer, NULL);

sprintf(buffer, "%d", data.left_bnd);
xv_set(left_bnd_text, PANEL_VALUE, (caddr_t)buffer, NULL);

sprintf(buffer, "%d", data.right_bnd);
xv_set(right_bnd_text, PANEL_VALUE, (caddr_t)buffer, NULL);

sprintf(buffer, "error = %.3E", lstsqer);
xv_set(error_item, PANEL_LABEL_STRING, (caddr_t)buffer, NULL);
}


/*      set_flag_mask sets the value of flag_mask for the current
value of data.func.  Note that if the data are actually a response
and not a decay, the flag_mask is preemptively set to 99.  Also,
mc_flags is masked by the new value of flag_mask.
*/

void set_flag_mask()
{
if (data.flags & 8)
```

```
    flag_mask = 99;
else if (data.func == 1)
    flag_mask = 367;
else if (data.func == 2)
    flag_mask = 315;
else
    flag_mask = 0;
mc_flags &= flag_mask;
}


/*      file_close() is called as a result of choosing the close
option under the file button.  It first gets the private data
structure for the data set, freeing both the data memory and then
the structure itself as well as the memory allocated for the
path-removed filename.  Then the frame is destroyed.  Finally, the
selected menu options are inactivated if appropriate.
*/

void file_close()
{
Xv_notice  notice;

int  i, notice_stat;

if (saved > 0){
    notice = (Xv_notice)xv_create(main_frame, NOTICE,
        NOTICE_MESSAGE_STRINGS, "File has changed since last save.",
                                "Do you want to close without saving?",
                                NULL,
        NOTICE_BUTTON_YES,  "Cancel close",
        NOTICE_BUTTON_NO,   "Close without saving",
        NOTICE_STATUS,      &notice_stat,
        XV_SHOW,            TRUE,
        NULL);
    if (notice_stat == NOTICE_YES){
        xv_destroy_safe(notice);
        return;
        }
    xv_destroy_safe(notice);
    }

for(i=0; i<4; i++)
    if (data.data[i] != NULL)
        free(data.data[i]);

xv_set((Menu_item)xv_find(file_menu, MENUITEM, MENU_VALUE, 2, NULL),
        MENU_INACTIVE,      TRUE,
        NULL);
```

```
xv_set((Menu_item)xv_find(file_menu, MENUITEM, MENU_VALUE, 3, NULL),
        MENU_INACTIVE,      TRUE,
        NULL);
xv_set((Menu_item)xv_find(file_menu, MENUITEM, MENU_VALUE, 4, NULL),
        MENU_INACTIVE,      TRUE,
        NULL);

clear_window();
saved = -1;
return;
}
```

## B.3 fitgraph.c

```
/*      This file contains the following routines:
setup_graphics(), initial_plot(), clear_window(), plot(),
print_function_text(), fit_fn_string(), draw_boxes(), rboundd(),
rtickxyd(), draw_data(), draw_fits(), and draw_resids().
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/panel.h>
#include <xview/cms.h>
#include <X11/Xlib.h>
#include <xview/xv_xrect.h>
#include <xview/font.h>

#include "fitX.h"

extern Frame  main_frame;
extern Canvas  dat_win;
extern Cms  cms;
extern Panel_item  error_item;

extern struct dat_set  data;
extern double  lstsqer;
extern char  buffer[];
extern int  data_color, fit_color, resid_color, t0_color;
extern int  box_color, label_color, white, black;


/*      Following are the plotting variables used solely in this
file.
*/

/*      *pixel_table is used to hold the real indices into the
colormap for the colormap segment created in setup_graphics().
*/

unsigned long  *pixel_table;

/*      The rect_?min? and rect_max? variables mark the boundaries
for the boxes.  The *d variables apply to the data box; *r, to the
residual box.
```

```
*/

short  rect_xmind = 60, rect_ymind = 40;
short  rect_xmaxd = 560, rect_ymaxd = 390;
short  rect_xminr = 60, rect_yminr = 430;
short  rect_xmaxr = 560, rect_ymaxr = 580;

/*      The func_* variables mark the location of the string
indicating the fuctional form chosen to convolve.
*/

int  func_x = 110, func_y = 20;

/*      The dat_box[] and resid_box[] arrays are used to pass the box
boundaries for the data box and the residual box respectively to some
of the graphics functions.
*/

XPoint  dat_box[5], resid_box[5];

/*      The rect_xdiff and rect_ydiffd variables are for the width
and height of the data box respectively.
        The xdiff and ydiffd variables are used to hold the
differences in the maximum and minimum values of the abscissa and
ordinate respectively of the actual data.
        The xs and ysd variables hold scaling factors for conversion
to pixel locations from x (point number) and y values respectively.
        rect_y0 is for the y-position in pixels of the y==0 line for
the residual box.
*/

float  rect_xdiff, rect_ydiffd;
float  xdiff, ydiffd;
float  xs, ysd;
short  rect_y0;

/*      The *_tick? variables are used in determining the labels for
the tick marks for the boxes.
*/

float  xmin_tick, xmax_tick;
float  ymin_tickd, ymax_tickd;

/*      dpy is the default display.  gc is the graphics context used
throughout this program except for with the pseudo-data.  Note that
the foreground color is nevertheless changed often.  gc_fits, which
is used with the pseudo-data, has a different line width as well as a
different color.
*/
```

```
Display  *dpy;
GC  gc, gc_fits;


/*      setup_colors() makes the colormap segment and defines the
colors.
*/

void setup_graphics()
{
static Xv_singlecolor  colors[] = {
    {255, 255, 255},    /*white*/
    {60, 190, 60},      /*data color - green*/
    {0, 0, 230},        /*fit color - blue*/
    {230, 0, 230},      /*t0 color - purple*/
    {230, 0, 0},        /*resid color - red*/
    {0, 230, 255},      /*box color - bright cyan*/
    {0, 0, 230},        /*label color - blue*/
    {0, 0, 0},          /*black*/
    };


cms = (Cms)xv_create((int)NULL, CMS,
    CMS_SIZE,           8 + CMS_CONTROL_COLORS,
    CMS_COLORS,         colors,
    CMS_CONTROL_CMS,    TRUE,
    NULL);
xv_set(main_frame, WIN_CMS, cms);

pixel_table = (unsigned long *)xv_get(cms, CMS_INDEX_TABLE);

white = CMS_CONTROL_COLORS;
data_color = 1 + CMS_CONTROL_COLORS;
fit_color = 2 + CMS_CONTROL_COLORS;
t0_color = 3 + CMS_CONTROL_COLORS;
resid_color = 4 + CMS_CONTROL_COLORS;
box_color = 5 + CMS_CONTROL_COLORS;
label_color = 6 + CMS_CONTROL_COLORS;
black = 7 + CMS_CONTROL_COLORS;
}



/*      setup_graphics() sets up the important graphics variables and
assigns values to the *_box[] arrays as well as several of the other
variables derived from the box boundary variables.
*/
```

```
void setup_graphics()
{
XGCValues  gcvalues;

dpy = (Display *)xv_get(main_frame, XV_DISPLAY);
gcvalues.graphics_exposures = False;
gcvalues.background = pixel_table[CMS_CONTROL_COLORS];
gc = XCreateGC(dpy, xv_get(main_frame, XV_XID),
            GCBackground | GCGraphicsExposures, &gcvalues);
gcvalues.line_width = 3;
gcvalues.foreground = pixel_table[fit_color];
gc_fits = XCreateGC(dpy, xv_get(main_frame, XV_XID),
    GCForeground | GCLineWidth | GCBackground | GCGraphicsExposures,
                                &gcvalues);

dat_box[0].x = dat_box[4].x = rect_xmind;
dat_box[0].y = dat_box[4].y = rect_ymind;
dat_box[1].x = rect_xmaxd;
dat_box[1].y = rect_ymind;
dat_box[2].x = rect_xmaxd;
dat_box[2].y = rect_ymaxd;
dat_box[3].x = rect_xmind;
dat_box[3].y = rect_ymaxd;

resid_box[0].x = resid_box[4].x = rect_xminr;
resid_box[0].y = resid_box[4].y = rect_yminr;
resid_box[1].x = rect_xmaxr;
resid_box[1].y = rect_yminr;
resid_box[2].x = rect_xmaxr;
resid_box[2].y = rect_ymaxr;
resid_box[3].x = rect_xminr;
resid_box[3].y = rect_ymaxr;

rect_y0 = (rect_ymaxr - rect_yminr) / 2 + rect_yminr;

rect_xdiff = (float)(rect_xmaxd - rect_xmind);
rect_ydiffd = (float)(rect_ymaxd - rect_ymind);
}


/*      initial_plot() handles the first plotting of a data set after
opening its window.  The function string is printed, the boxes are
drawn, and the data are drawn.
*/

void initial_plot()
{
Window  xwin;
```

```
xwin = (Window)xv_get(canvas_paint_window(dat_win), XV_XID);

draw_boxes(xwin);
print_function_text(xwin);

rboundd((float)1, data.datmin, (float)data.points_scan,
                                                data.datmax, xwin);
draw_data(xwin);
}


/*      clear_window() clears the window after closing a data set.
*/

void clear_window()
{
Window  xwin;

xwin = (Window)xv_get(canvas_paint_window(dat_win), XV_XID);
XClearWindow(dpy, xwin);
}


/*      plot() handles subsequent plottings of the data.  Note that
while the function string does not need to be reprinted, the data,
residuals, and even the axis labels themselves do (the latter being
necessary, for example, after changing the t0 value).  Also, unlike
the initial plot, the pseudo-data and residuals are plotted.  This
is also the location where the squared error is updated after a
calculation.
*/

void plot()
{
Window  xwin;

xwin = (Window)xv_get(canvas_paint_window(dat_win), XV_XID);

XClearWindow(dpy, xwin);

draw_boxes(xwin);
print_function_text(xwin);

rboundd((float)1, data.datmin, (float)data.points_scan,
                                                data.datmax, xwin);
draw_fits(xwin);
draw_data(xwin);
draw_resids(xwin);
```

```
sprintf(buffer, "error = %.3E", lstsqer);
xv_set(error_item, PANEL_LABEL_STRING, (caddr_t)buffer, NULL);
}


/*      print_function_text() prints the function string.  Note that
the actual strings are provided by the routine fit_fn_string().
*/

void print_function_text(Window xwin)
{
if (!(data.flags & 8)){
    XSetForeground(dpy, gc, pixel_table[label_color]);
    XDrawString(dpy, xwin, gc, func_x, func_y,
        fit_fn_string(data.func), strlen(fit_fn_string(data.func)));
    }
else{
    XSetForeground(dpy, gc, pixel_table[label_color]);
    XDrawString(dpy, xwin, gc, func_x, func_y,
      fit_fn_string(2+data.conv),strlen(fit_fn_string(2+data.conv)));
    }
}


char *fit_fn_string(int n)
{
static char  *string[] = {
        "error",
        " c1*exp(-t/a1) + c2*exp(-t/a2) - (c1+c2)exp(-t/b1) + c4 ",
        "       c1*exp((-t/a1)**b2) - c1*exp(-t/b1) + c4         ",
        "c1*exp((-1/2)*((t-a1)/sigma)^2) + c2; c3=noise; c4=chisq",
        "   c1*(sech((t-a1)/sigma))^2 + c2; c3=noise; c4=chisq   ",
         "    c1/(1+((t-a1)/sigma)^2) + c2; c3=noise; c4=chisq    ",
        };
return((n < 1 || n > 5) ? string[0] : string[n]);
}


/*      draw_boxes() handles the drawing of the boxes including the
tick marks but not the labeling of the axes.
*/

void draw_boxes(Window xwin)
{
short  dtempx, tempx, dtempy, tempy;
int  i;

dtempx = (rect_xmaxd - rect_xmind) / 5;
```

```
tempx = rect_xmind + dtempx;
dtempy = (rect_ymaxd - rect_ymind) / 5;
tempy = rect_ymind + dtempy;

XSetForeground(dpy, gc, pixel_table[box_color]);

XDrawLines(dpy, xwin, gc, dat_box, 5, CoordModeOrigin);

for(i=1; i<5; i++){
    XDrawLine(dpy, xwin, gc, tempx, rect_ymaxd, tempx,rect_ymaxd-10);
    XDrawLine(dpy, xwin, gc, rect_xmind, tempy, rect_xmind+10,tempy);
    tempx+=dtempx;
    tempy+=dtempy;
    }

XDrawLines(dpy, xwin, gc, resid_box, 5, CoordModeOrigin);
XDrawLine(dpy, xwin, gc, rect_xminr, rect_y0, rect_xmaxr, rect_y0);
}


/*      rboundd() determines the full width (xdiff) and height
(ydiffd) of the current data set.  The range of the data is extended
by 5% in both directions for graphing purposes to keep the data from
overlapping the top or bottom of the box.  Then rtickxyd() is called
to label the tick marks.
*/

void rboundd(xmin, ymind, xmax, ymaxd, xwin)
float  xmin, ymind, xmax, ymaxd;
Window  xwin;
{
float  fy;

xdiff = xmax - xmin;
ydiffd = ymaxd - ymind;

fy = ydiffd / 20;
ymind -= fy;
ymaxd += fy;
ydiffd += 2*fy;

rtickxyd(xmin, ymind, xwin);
}


/*      rtickxyd() labels the tick marks of the x and y axes for the
data box.  For a data range exceeding 10, as would be the case in the
fluorescence data, integer values are used; otherwise, floats are
used.
```

```
*/

void rtickxyd(xmin, ymind, xwin)
float  xmin, ymind;
Window  xwin;
{
Xv_Font  font;
Font_string_dims  fdims;
float  temp, temp_tick, tempt;
int  temp2;

font = (Xv_Font)xv_get(dat_win, XV_FONT);

if (data.flags%2){
    xmin_tick = floor(xmin);
    xmax_tick = ceil(xmin + xdiff);
    }
else{
    xmin_tick = xmin;
    xmax_tick = xmin + xdiff;
    }
temp_tick = xmin_tick;

xdiff = xmax_tick - xmin_tick;
temp = xdiff / 5.;
tempt = temp * data.time_per_point;
xs = rect_xdiff / xdiff;
xmin = xmin_tick;

while (temp_tick <= xmax_tick){
    if (tempt > (float)10.)
        sprintf(buffer, "%.0f",
            (temp_tick - data.tzero) * data.time_per_point);
    if (tempt > (float)1.)
        sprintf(buffer, "%.1f",
            (temp_tick - data.tzero) * data.time_per_point);
    else if (tempt > .1)
        sprintf(buffer, "%.2f",
            (temp_tick - data.tzero) * data.time_per_point);
    else if (tempt <= .1)
        sprintf(buffer, "%.3f",
            (temp_tick - data.tzero) * data.time_per_point);

    (void)xv_get(font, FONT_STRING_DIMS, buffer, &fdims);
    temp2 = (int)((temp_tick - xmin_tick) * xs + rect_xmind -
                            fdims.width/2);

    XDrawString(dpy, xwin, gc, temp2, rect_ymaxd+fdims.height+5,
                                        buffer, strlen(buffer));
```

```
    temp_tick = temp_tick + temp;
    }

if (ydiffd > 10){
    ymin_tickd = floor(ymind);
    ymax_tickd = ceil(ymind + ydiffd);
    }
else{
    ymin_tickd = ymind;
    ymax_tickd = ymind + ydiffd;
    }
temp_tick = ymin_tickd;

ydiffd = ymax_tickd - ymin_tickd;
temp = ydiffd / 5;
ysd = rect_ydiffd / ydiffd;
ymind = ymin_tickd;

while (temp_tick <= ymax_tickd){
    if (temp > (float)1.)
        sprintf(buffer, "%.0f", temp_tick);
    else if (temp > (float) .1)
        sprintf(buffer, "%.1f", temp_tick);
    else if (temp > (float) .01)
        sprintf(buffer, "%.2f", temp_tick);
    else if (temp < (float) .01)
        sprintf(buffer, "%.3f", temp_tick);

    (void)xv_get(font, FONT_STRING_DIMS, buffer, &fdims);
    temp2 = (int)(rect_ymaxd - (temp_tick - ymin_tickd) * ysd +
                                                    fdims.height/3);

    XDrawString(dpy, xwin, gc, rect_xmind-fdims.width-5, temp2,
                                            buffer, strlen(buffer));
    temp_tick = temp_tick + temp;
    }
}


/*      draw_data() handles the drawing of the data.  As it
calculates the location of each point in pixels, it plots that point
as a 3x3 cluster of pixels centered on the calculated point.  It
also calculates the graphics coordinate of the zero time point for
special treatment, namely overplotting in a different color from the
rest of the data.  Similarly it calculates the x-coordinates of the
left and right boundaries for calculation of the squared error and
plots vertical lines for those bounds.
*/
```

```
void draw_data(Window xwin)
{
int  ptx, pty, t0x, t0y, leftx, rightx;
int  i, j, k;

XSetForeground(dpy, gc, pixel_table[data_color]);
for (i=0; i<data.points_scan; i++){
    ptx = (int)((i + 1 - xmin_tick) * xs + rect_xmind);
    pty = (int)(rect_ymaxd - (*(data.data[0]+i) - ymin_tickd) * ysd);
    for(j=-1; j<2; j++){
        for(k=-1; k<2; k++){
            XDrawPoint(dpy, xwin, gc, ptx+j, pty+k);
            }
        }
    }

XSetForeground(dpy, gc, pixel_table[t0_color]);
t0x = (int)((data.tzero - xmin_tick) * xs + rect_xmind);
t0y = (int)(rect_ymaxd -
            (*(data.data[0] + data.tzero - 1) - ymin_tickd) * ysd);
for(i=-1; i<2; i++){
    for(j=-1; j<2; j++){
        XDrawPoint(dpy, xwin, gc, t0x+i, t0y+j);
        }
    }

XSetForeground(dpy, gc, pixel_table[resid_color]);
leftx = (int)((data.left_bnd - xmin_tick) * xs + rect_xmind);
rightx = (int)((data.right_bnd - xmin_tick) * xs + rect_xmind);
XDrawLine(dpy, xwin, gc, leftx, rect_ymind, leftx, rect_ymaxd);
XDrawLine(dpy, xwin, gc, rightx, rect_ymind, rightx, rect_ymaxd);
}


/*      draw_fits() plots the constructed data in the form of a
connected line plot.  Use of a point list and the filling thereof
are as above.
*/

void draw_fits(Window xwin)
{
XPoint  *ptlist;
int  i, j=0, temp;

ptlist = (XPoint *)malloc((data.points_scan+1) * sizeof(XPoint));
if(ptlist == NULL){
    allocation_error("the ptlist.");
    xv_destroy_safe(main_frame);
    }
```

```
for (i=0; i<data.points_scan; i++){
    (ptlist + i)->x = (int)((i + 1 - xmin_tick) * xs + rect_xmind);
    temp = (int)(rect_ymaxd -
                            (*(data.data[1]+i) - ymin_tickd) * ysd);
    if(temp < rect_ymind)
        temp = rect_ymind;
    else if (temp > rect_ymaxd)
        temp = rect_ymaxd;
    (ptlist + i)->y = temp;
    }

XDrawLines(dpy, xwin, gc_fits, ptlist,
                                data.points_scan, CoordModeOrigin);
free(ptlist);
}


/*      draw_resids() handles the plotting of the residual points in
the residual box.  As with the data, each point is plotted as a 3x3
cluster of pixels.  Again the zero time point is overplotted in a
different color for easy identification, and the left and right
bounding lines are added.
*/

void draw_resids(Window xwin)
{
int  ptx, pty, t0x, t0y, leftx, rightx;
int  i, j, k;

XSetForeground(dpy, gc, pixel_table[resid_color]);
for (i=0; i<data.points_scan; i++){
    ptx = (int)((i + 1 - xmin_tick) * xs + rect_xminr);
    pty = (int)(rect_y0 - (*(data.data[2] + i)) * ysd);
    if (pty < rect_yminr)
        pty = rect_yminr;
    else if (pty > rect_ymaxr)
        pty = rect_ymaxr;
    for(j=-1; j<2; j++){
        for(k=-1; k<2; k++){
            XDrawPoint(dpy, xwin, gc, ptx+j, pty+k);
            }
        }
    }

leftx = (int)((data.left_bnd - xmin_tick) * xs + rect_xmind);
rightx = (int)((data.right_bnd - xmin_tick) * xs + rect_xmind);
XDrawLine(dpy, xwin, gc, leftx, rect_yminr, leftx, rect_ymaxr);
XDrawLine(dpy, xwin, gc, rightx, rect_yminr, rightx, rect_ymaxr);
```

```
XSetForeground(dpy, gc, pixel_table[fit_color]);
t0x = (int)((data.tzero - xmin_tick) * xs + rect_xminr);
t0y = (int)(rect_y0 - (*(data.data[2] + data.tzero - 1)) * ysd);
if (t0y < rect_yminr)
    t0y = rect_yminr;
else if (t0y > rect_ymaxr)
    t0y = rect_ymaxr;
for(i=-1; i<2; i++){
    for(j=-1; j<2; j++){
        XDrawPoint(dpy, xwin, gc, t0x+i, t0y+j);
        }
    }
}
```

## B.4 calc.c

```
/*    This file contains the following routines:
calc(), convander(), calc_err(), biexp(), stretch(), normal(),
sech_squared(), lorentzian(), calct0(),  initialize(),
baseline_adjust(), set_pextrema_mats(), set_pdelta_mat(),
resp_prep(), monte(), and fitt().
*/

#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/notice.h>

#include "nr3.h"
#include "fitX.h"

extern struct dat_set  data;
extern double  lstsqer, *cnv;
extern int  ibase, cnvpts, resp_step, icnvmax;
extern int  mc_flags, flag_mask, saved;
extern float  pdelta[], pmax[], pmin[];
extern char  buffer[];
extern Frame  main_frame;
extern Panel_item  right_bnd_text;


/*      calc() is the primary calculational subroutine for fitting
decays.  (For fitting response functions, see calct0().)  It is
responsible for allocating (and at the end freeing) space for the
matrices used in constructing the pseudo-data after determining the
appropriate sizes.  All three of the necessary arrays are zeroed and
the response function array rspns[] is then filled, using either the
analytical function selected or the response data in conv.  Then the
decay array fitfn[] is filled by calling the appropriate subroutine.
Lastly the convolution is carried out and the squared error is
calculated in convander() before the dynamically allocated array
space is freed.
*/

void calc()
{
Xv_notice  notice;
float  *fitfn, *rspns, *conv, ftemp;
double  sigma;
```

```
int  ps, i, hwid, diff;

if (saved == -1)
    return;

/*      First the size of the arrays to be used in the convolution
is ensured to be of a length equal to an integer power of two of
sufficient size to hold at least points_scan points plus a 2*hwid
extenstion of the function and hwid padding.  For more information
on padding, see Numerical Recipes in C.  Then this space is
allocated.
*/

switch(data.conv){
case 1:
    sigma = (double)(data.param[0] / (data.time_per_point*2.35482));
    hwid = ((int)sigma + 1) * 3;
    break;
case 2:
    sigma = (double)(data.param[0] / (data.time_per_point*1.76275));
    hwid = ((int)sigma + 1) * 3;
    break;
case 3:
    sigma = (double)(data.param[0] / (data.time_per_point*2.));
    hwid = data.points_scan/2;
    break;
case 4:
    hwid = (icnvmax < cnvpts-icnvmax-1) ? cnvpts-icnvmax-1 : icnvmax;
    if (resp_step != data.stage_step){
        notice = (Xv_notice)xv_create(main_frame, NOTICE,
            NOTICE_MESSAGE_STRINGS, "Data-Response Mismatch!",
                                    "Steps per Point Differ!",
                                    "Results Will Be Distorted!",
                                    NULL,
            NOTICE_BUTTON_YES,      "Confirm",
            XV_SHOW,                TRUE,
            NULL);
        xv_destroy_safe(notice);
        }
    }
for(ps=64; ps<data.points_scan+3*hwid; ps*=2)
    ;
fitfn = vector(1, ps);
rspns = vector(1, ps);
conv = vector(1, 2*ps);
for (i=1; i<=ps; i++)
    rspns[i] = conv[i] = conv[i+ps] = 0.0;

/*      Next the response function is determined and calculated.
```

```
Note that for data.param[0] == 0 no convolution is performed, as
indicated by hwid==1.  This is equivalent to using a delta function.
Also, note that the function is stored in wrap-around order in the
first elements of the array rspns[] as required for the numerical
recipes routine convlv().
*/

if (data.param[0] < 0.00001)
    hwid = 1;
else switch (data.conv){
case 1:
    rspns[1] = 1.0 / (2.50663*sigma);
    for (i=1; i<=hwid; i++){
        rspns[i+1] = exp(-(float)(i*i)/(2.0*sigma*sigma)) * rspns[1];
        rspns[2*hwid + 2 - i] = rspns[i+1];
        }
    break;
case 2:
    rspns[1] = 1.0 / (2.*sigma);
    for (i=1; i<=hwid; i++){
        ftemp = 1 / cosh((float)i / sigma);
        rspns[i+1] = ftemp * ftemp * rspns[1];
        rspns[2*hwid + 2 - i] = rspns[i+1];
        }
    break;
case 3:
    rspns[1] = 1.0 / (3.14159*sigma);
    for (i=1; i<=hwid; i++){
        rspns[i+1] = rspns[1] / (1 + (float)(i*i) / (sigma*sigma));
        rspns[2*hwid + 2 - i] = rspns[i+1];
        }
    break;
case 4:
    for (i=1; i<=cnvpts-icnvmax; i++)
        rspns[i] = *(cnv + i - 1);
    diff = abs(cnvpts - 2*icnvmax - 1);
    for (i+=diff; i<=2*hwid+1; i++)
        rspns[i] = *(cnv + i - 1 - diff);
    break;
    }

/*      Now the fitting function is calculated by calling an
appropriate subroutine.  The results are arranged in fitfn in the
first data.points_scan+2*hwid elements, although the first tzero-1
are set to 0.  The array is then padded out to the end; thus,
there are at least hwid padded 0's.
*/

if (data.func == 1)
```

```
    biexp(fitfn, hwid);
else if (data.func == 2)
    stretch(fitfn, hwid);
for (i=data.points_scan+2*hwid+1; i<=ps; i++)
    fitfn[i] = 0.0;

/*      Next the subroutine convander() calculates the convolution
and unweighted squared error, which is stored in the external
variable lstsqer.  The fitted curve values are stored in
data.data[1] and the residuals are stored in data.data[2].
*/

convander(hwid, fitfn, rspns, conv, ps);

/*      Finally the arrays allocated above are freed.
*/

free_vector(fitfn, 1, ps);
free_vector(rspns, 1, ps);
free_vector(conv, 1, 2*ps);
return;
}


/*      convander() calls convlv(), a routine from numerical recipes,
to calculate the convolution as long as hwid!=1.  If hwid==1, no
convolution is performed; this is the equivalent to convolving with
a delta function.  Note that convlv() also requires twofft(),
realft(), four1() and some routines in nrutil.c.  Then the
constructed data are copied into data.data[1] and calc_err()
is called to calculate residuals and the squared error.
*/

void convander(hwid, fitfn, rspns, conv, ps)
int  hwid;
float  *fitfn, *rspns, *conv;
int  ps;
{
int  i;

if (hwid > 1)
    convlv(fitfn, ps, rspns, 2*hwid+1, 1, conv);
else{
    for (i=1; i<=data.points_scan; i++)
        conv[i] = fitfn[i];
    }

for (i=0; i<data.points_scan; i++)
    *(data.data[1]+i) = (double)conv[i+1];
```

```
calc_err();
}


/*      calc_err() calculates the residuals for all of the points and
the squared error between the left and right bounds as marked by
data.left_bnd and data.right_bnd.  The former are stored in
data.data[2]; the latter, in lstsqer.
*/

void calc_err()
{
int  i;
double  dtemp;

for (i=0; i<data.left_bnd-1; i++)
    *(data.data[2]+i) = *(data.data[0]+i) - *(data.data[1]+i);

if (!(data.flags & 16)){
    for (lstsqer=0.0; i<data.right_bnd; i++){
        *(data.data[2]+i) = *(data.data[0]+i) - *(data.data[1]+i);
        lstsqer += (*(data.data[2]+i)) * (*(data.data[2]+i));
        }
    }
else{
    for (lstsqer=0.0; i<data.right_bnd; i++){
        *(data.data[2]+i) = *(data.data[0]+i) - *(data.data[1]+i);
        if (*(data.data[0]+i) == 0.0)
            *(data.data[0]+i) += data.datmax / 100000.0;
        dtemp = (*(data.data[2]+i))*data.datmax/(*(data.data[0]+i));
        lstsqer +=  dtemp * dtemp;
        }
    }

for (; i<data.points_scan; i++)
    *(data.data[2]+i) = *(data.data[0]+i) - *(data.data[1]+i);
}


/*      biexp() calculates the values for a biexponential decay
function with possible exponential rise and constant offset.  In
order to avoid unnecessary calculations of exponentials, the program
takes advantage of the fact that the exponential of the sum of the
exponents is the product of the exponentials.  Note also the
protection versus underflow errors.
*/

void biexp(fitfn, hwid)
float  fitfn[];
```

```
int  hwid;
{
float  f1c, f1, f2c, f2, f3c, f3;
int  i;

if (data.time_per_point/(data.param[1] + .001) > 18 ||
                                                data.param[1] == 0){
    f1c = 0.0;
    f1 = 0.0;
    }
else{
    f1c = exp(-data.time_per_point/data.param[1]);
    f1 = 1.0;
    }

if (data.time_per_point/(data.param[2] + .001) > 18 ||
                                                data.param[2] == 0){
    f2c = 0.0;
    f2 = 0.0;
    }
else{
    f2c = exp(-data.time_per_point/data.param[2]);
    f2 = 1.0;
    }

if (data.time_per_point/(data.param[3] + .001) > 18 ||
                                                data.param[3] == 0){
    f3c = 0.0;
    f3 = 0.0;
    }
else{
    f3c = exp(-data.time_per_point/data.param[3]);
    f3 = 1.0;
    }

for (i=1; i<data.tzero; i++)
    fitfn[i] = 0.0;
for (; i<=data.points_scan+2*hwid; i++){
    fitfn[i] = data.param[5]*f1 + data.param[6]*f2
        - (data.param[5] + data.param[6])*f3 + data.param[8];
    f1 *= f1c;
    f2 *= f2c;
    f3 *= f3c;
    }
}


/*      stretch() calculates the values for a stretched exponential
```

```
decay function (aka Kohlrausch) with possible exponential rise and
constant offset.  Note that in general the trick used above to avoid
calculating excessive exponentials cannot be used here.
*/

void stretch(fitfn, hwid)
float  fitfn[];
int  hwid;
{
float  f3c, f3, f1, f1c;
int  i;

if (data.time_per_point/(data.param[3] + .001) > 18 ||
                                                data.param[3] == 0){
    f3c = 0.0;
    f3 = 0.0;
    }
else{
    f3c = exp(-data.time_per_point/data.param[3]);
    f3 = 1.0;
    }

for (i=1; i<data.tzero; i++)
    fitfn[i] = 0.0;
if (data.param[1] < .001){
    for (; i<=data.points_scan+2*hwid; i++)
        fitfn[i] = data.param[8];
    }
else{
    if (data.param[4] < .001 || data.param[4] > .999){
        if (data.time_per_point/(data.param[1] + .001) > 18){
            f1c = 0.0;
            f1 = 0.0;
            }
        else{
            f1c = exp(-data.time_per_point/data.param[1]);
            f1 = 1.0;
            }
        for (; i<=data.points_scan+2*hwid; i++){
            fitfn[i] = data.param[5]*(f1-f3) + data.param[8];
            f3 *= f3c;
            f1 *= f1c;
            }
        }
    else{
        for (; i<=data.points_scan+2*hwid; i++){
            f1 = pow((i - data.tzero)*data.time_per_point /
                        data.param[1], data.param[4]);
            if (f1 > 12.0)
```

```
                f1 = 12.0;
            fitfn[i] = data.param[5]*(exp(-f1)-f3) + data.param[8];
            f3 *= f3c;
            }
        }
    }
}


/*      The subroutine normal calculates the y-value, *yfit, of a
gaussian curve of the variable  x, using the parameters stored in
array, a, of length, na (in the present case this is unimportant
but retained for consistency).  Note that a was allocated in the
main routine to run from element 1 to element na, not from 0 to na-1.
This is for consistency with NR routines.  normal() also determines
the values of the first derivatives with respect to the appropriate
parameters at the point x and stores those values in the array dyda.
The subroutines sech_squared() and lorentzian() do exactly
the same thing but for the functions hyperbolic secant squared and
lorentzian respectively.
*/

void normal(x, a, yfit, dyda, na)
float  x, a[], *yfit, dyda[];
int  na;
{
float  f1, f2, f3, sd;

sd = a[1] / 2.35482;
f1 = (x - a[2]) / sd;
f2 = exp(-f1 * f1 / 2);
f3 = a[3] * f2 * f1;

*yfit = a[3] * f2 + a[4];
dyda[4] = 1;
dyda[3] = f2;
dyda[2] = f3 / sd;
dyda[1] = f3 * f1 / a[1];
}


void sech_squared(x, a, yfit, dyda, na)
float  x, a[], *yfit, dyda[];
int  na;
{
float  f1, f2, f3, f4, sech;
```

```
f4 = a[1] / 1.76275;
f1 = (x - a[2]) / f4;
sech =  (float)(1 / cosh(f1));
f2 = sech * sech;
f3 = (float)(2 * a[3] * f2 * tanh(f1));

*yfit = a[3] * f2 + a[4];
dyda[4] = 1;
dyda[3] = f2;
dyda[2] = f3 / f4;
dyda[1] = f3 * f1 / a[1];
}


void lorentzian(x, a, yfit, dyda, na)
float  x, a[], *yfit, dyda[];
int  na;
{
float  f1, f2, f3, f4;

f4 = a[1] / 2;
f1 = (x - a[2]) / f4;
f2 = (1 + f1 * f1);
f3 = 2 * a[3] * f1 / (f2 * f2);

*yfit = a[3] / f2 + a[4];
dyda[4] = 1;
dyda[3] = 1 / f2;
dyda[2] = f3 / f4;
dyda[1] = f3 * f1 / a[1];
}


/*      calct0() is the primary calculational sub-routine for fitting
response functions.  It allocates some space for parameters used in
the actual calculational routines and then calls the appropriate
one depending on the value of data.conv, the variable that determines
which response function the user wants fit to the data.  Afterwards
it calls calc_err() to calculate the squared error and frees up the
memory it allocated.
*/

void calct0()
{
int  i, ma=4;
float  *a, dummya[5], temp;
void (*funcs)();
```

```
a = vector(1, ma);
a[1] = data.param[0];
a[2] = data.param[1];
a[3] = data.param[5];
a[4] = data.param[6];

a[1] /= data.time_per_point;
a[2] /= data.time_per_point;

if (data.conv == 1)
    funcs = normal;
else if (data.conv == 2)
    funcs = sech_squared;
else if (data.conv == 3)
    funcs = lorentzian;

for (i=1; i<=data.points_scan; i++){
    funcs((float)i, a, &temp, dummya, ma);
    *(data.data[1] + i - 1) = (double)temp;
    }
calc_err();
free_vector(a, 1, ma);
}


/*    initialize() determines the maximum and minimum values of the
data set.  Also, if the data is of the non-bleach, fluorescence
variety (ps or fs), the baseline is adjusted based on the current
value of ibase by the routine baseline_adjust().  After some
additional initializations, including zeroing the parameters in
param[], the control panel is reset by reset_control_panel(), and
the values in the arrays pdelta[], pmax[], and pmin[] are set by
set_pdelta_mat() and set_extrema_mats().
*/

void initialize()
{
int  i;
char  *input;


/*      The maximum and minimum are first determined.  For
fluorescence data the baseline is determined and the data are
corrected therefor in baseline_adjust().
*/

data.datmax = (float)(*(data.data[0]) + .00001);
data.datmin = (float)(*(data.data[0]));
```

```
for (i=0; i<data.points_scan; i++){
    if (*(data.data[0] + i) > data.datmax)
        data.datmax = (float) *(data.data[0] + i);
    else if (*(data.data[0] + i) < data.datmin)
        data.datmin = (float) *(data.data[0] + i);
    }

if ((data.flags & 2) && !(data.flags & 8))
    baseline_adjust();

/*      Next the array data.param is initialized to zero.  Then pmax,
pmin, and pdelta are set in set_pmats().
*/

data.tzero = 1;
data.left_bnd = 1;
data.right_bnd = data.points_scan;
sprintf(buffer, "%d", data.right_bnd);

xv_set(right_bnd_text, PANEL_VALUE, (caddr_t)buffer, NULL);

for (i=0; i<9; i++)
    data.param[i] = 0.0;
set_pextrema_mats();
set_pdelta_mat();
reset_control_panel();
return;
}


/*      baseline_adjust() averages the values of the first ibase
points, subtracts this value from all of the points, and scales the
maximum to 100.
*/

void baseline_adjust()
{
int  i;
double  baseline;

/*      The baseline is calculated by averaging the points up through
the point ibase, in which case data.datmax is then corrected for the
baseline, after which correction the data is corrected for the
baseline as well and rescaled to the maximum times 100.  data.datmin
is then rescaled by subtracting off the baseline and multiplying by
100/data.datmax; data.datmax is then set to 100, the new maximum.
*/
```

```
if (ibase > data.points_scan || ibase <= 0)
    ibase = 1;

baseline = 0.0;
for (i=0; i<ibase; i++)
    baseline += *(data.data[0] + i);
baseline /= (double)ibase;
data.datmax -= (float)baseline;

for (i=0; i<data.points_scan; i++)
    *(data.data[0] + i) = (*(data.data[0] + i) - baseline) * 100.0 /
                              (double)data.datmax;
data.datmin = (data.datmin - (float)baseline) * 100.0 / data.datmax;
data.datmax = 100.0;
}


/*      set_pextrema_mats() sets the values in pmax[] and pmin[],
depending on the decay function chosen as well as the type of data.
*/

void set_pextrema_mats()
{
int  i;

set_flag_mask();

/*      pmax and pmin are first set to 0; elements corresponding to
parameters used by function data.func are reset later if necessary
(usually not the case for pmin except for the constant offset, which
is generally c4=param[8]).  Finally, corrections for bleaches to both
pmax and pmin are made as necessary.
*/

for (i=0; i<9; i++)
    pmax[i] = pmin[i] = 0.0;

pmax[0] = 1.0E6;
if (data.flags & 8){
    pmax[1] = (float)data.points_scan * data.time_per_point;
    pmax[5] = pmax[6] = pmax[7] = pmax[8] = 1.0E6;
    pmin[1] = 1.0E-6;
    return;
    }
if (data.func == 1 || data.func == 2){
    pmax[1] = pmax[3] = pmax[5] = pmax[8] = 1.0E6;
    pmin[8] = -1.0E6;
```

```
    }
if (data.func == 1)
    pmax[2] = pmax[6] = 1.0E6;
else if (data.func == 2)
    pmax[4] = 1.0;

if (data.flags & 4){
    for (i=5; i<8; i++){
        pmin[i] = -pmax[i];
        }
    }
}



/*      set_pdelta_mat() sets the values in pdelta[], depending on
the type of data.
*/

void set_pdelta_mat()
{
int  itemp;
int  i;

pdelta[9] = pdelta[10] = pdelta[11] = 1.0;
if (data.flags & 1){
    pdelta[0] = .01;
    pdelta[1] = pdelta[2] = 1.0;
    }
else{
    pdelta[0] = .001;
    pdelta[1] = pdelta[2] = .1;
    }
pdelta[3] = pdelta[4] = .01;

if (data.flags & 8){
    itemp = (int)log10(data.datmax);
    itemp = (int)exp10((double)itemp);
    pdelta[5] = (float)itemp / 1000.0;
    pdelta[6] = (float)itemp / 100.0;
    pdelta[7] = (float)itemp / 10000.0;
     pdelta[8] = 0;
    return;
    }
if (data.flags & 2)
    pdelta[5] = pdelta[6] = pdelta[7] = pdelta[8] = .1;
else
    pdelta[5] = pdelta[6] = pdelta[7] = pdelta[8] = .001;
}
```

```
/*      The following routine prepares a real response function for
use in convolution.  It has to find the maximum, which is defined to
be time zero, scale the values to unit area, and rearrange them into
wrap-around format.  It is called only from fcnv_open().
*/

void resp_prep()
{
int  i;
double  area, dmax, *dtemp;

dtemp = (double *)malloc(cnvpts * sizeof(double));
if (dtemp == NULL){
allocation_error("the dtemp in resp_prep; don't use a real response");
    return;
    }

/*    First find the maximum and move the data to dtemp.
*/

dmax = *dtemp = *cnv;
icnvmax = 0;
for (i=1; i<cnvpts; i++){
    *(dtemp + i) = *(cnv + i);
    if (dmax < *(cnv + i)){
        dmax = *(cnv + i);
        icnvmax = i;
        }
    }

/*      Next normalize the "pulse" area.
*/

area = (*dtemp) / 3.0;
area += *(dtemp + 1) * (4.0/3.0);
for (i=2; i<cnvpts-1; i+=2){
    area += (2.0/3.0)*(*(dtemp + i)) + (4.0/3.0)*(*(dtemp+i+1));
    }
if (cnvpts % 2)
    area += (*(dtemp + cnvpts - 1)) / 3.0;

if (area > 0.0){
    for (i=0; i<cnvpts; i++)
        *(dtemp + i) /= area;
    }
```

```
/*      Now arrange the response in wrap-around order.
*/

for (i=0; i<icnvmax; i++)
    *(cnv + cnvpts - icnvmax + i) = *(dtemp + i);
for (; i<cnvpts; i++)
    *(cnv + i - icnvmax) = *(dtemp + i);

free(dtemp);
return;
}


/*      monte() is the routine that randomly searches through a
subspace of the parameter space as determined by the bits in mc_flags
for a better "fit" of the constructed data to the real data as
determined by a smaller value for lstsqer.  This routine is replaced
by fitt() for fitting response functions.
        The routine first determines which if any of the parameters
are to be varied by determining if the corresponding bit is on and
a non-zero value has been entered via the control panel.  If no such
parameters exist, the routine returns.  Otherwise it seeds the random
number generator with the squared error and enters a loop 5000
iterations long in which it randomly and independently selects a
value for each of the varied parameters over a range from 50% to 150%
of its current value, calculates the new constructed data set,
determines the squared error, and compares it to the saved value in
old_error.  If the new value is less than the old, the new parameters
are copied over and the control panel and plot are updated.
*/

int monte()
{
Panel_item  pitem, nitem;
int  i, iterations=0, j=0, num_fit_params=0, fit_params[9];
float  mcmin[9], mcdelta[9], temp_params[9];
double  old_error;
unsigned  u2to32less1;
time_t  lt;

u2to32less1 = 0xFFFFFFFF;
time(&lt);

for (i=0; i<9; i++){
    if ((mc_flags & (1 << i)) && data.param[i] != 0.0){
        fit_params[num_fit_params] = i;
        num_fit_params++;
        }
```

```
    }

if (num_fit_params == 0)
    return(1);

for(i=0; i<num_fit_params; i++){
    mcmin[i] = data.param[fit_params[i]] / 2;
    if (3*mcmin[i] < pmax[fit_params[i]])
        mcdelta[i] = mcmin[i]*2;
    else
        mcdelta[i] = pmax[fit_params[i]] - mcmin[i];
    }

for(i=0; i<9; i++)
    temp_params[i] = data.param[i];
old_error = lstsqer;

srand((int)lt);
do{
    iterations++;
    j++;

    for(i=0; i<num_fit_params; i++)
        data.param[fit_params[i]] = (float)rand()/(float)u2to32less1
                                * mcdelta[i] + mcmin[i];
    calc();
    if (old_error > lstsqer){
        for(i=0; i<9; i++)
            temp_params[i] = data.param[i];
        old_error = lstsqer;
        reset_control_panel();
        plot();
        }
    } while(iterations < 5000);

for(i=0; i<9; i++)
    data.param[i] = temp_params[i];
calc();
reset_control_panel();
plot();
return(0);
}


/* fitt() is called as a result of hitting the "Fit" button
when the data being fit are for a response function as indicated
by the appropriate bit in data.flags.  This routine is only called
by mc_proc().
*/
```

```
void fitt()
{
int  i, j, k, count=0, num_fit_params;
float  *a, *x, *y, *sig, alamda, olamda, oldchisq, dchisq,
    **covar, **alpha;
float  dummy, dummya[5], chisq;
int  ma=4, *lista;
void (*funcs)();
unsigned  u2to32less1;

u2to32less1 = 0xFFFFFFFF;

/*      First set up the x and y vectors which contain the data.
Also set up the vector a, which will hold the parameters of the fit.
In the current case, this will have a fixed length of 4.  Note that
the user inputs a noise value to use as the estimated standard
deviation, which is kept constant for all the points.  Finally, set
up a vector lista, also of length 4.  This should be a permutation
of the numbers 1 to ma.  The first mfit values (in our case mfit
is usually just ma) are actually fit; the remainder are left fixed.
*/

x = vector(1, data.points_scan);
y = vector(1, data.points_scan);
sig = vector(1, data.points_scan);

if (data.conv == 1)
    funcs = normal;
else if (data.conv == 2)
    funcs = sech_squared;
else if (data.conv == 3)
    funcs = lorentzian;
a = vector(1, ma);

covar = matrix(1, ma, 1, ma);
alpha = matrix(1, ma, 1, ma);

lista = ivector(1, ma);
a[1] = (float)data.param[0];
a[2] = (float)data.param[1];
a[3] = (float)data.param[5];
a[4] = (float)data.param[6];

a[1] /= data.time_per_point;
a[2] /= data.time_per_point;

chisq = (float)data.param[8];
```

```
for (num_fit_params=0, k=4, i=0; i<2; i++){
    for (j=0; j<2; j++){
        if (mc_flags & (1 << (i*5+j))){
            num_fit_params++;
            lista[num_fit_params] = 2*i + j + 1;
            }
        else{
            lista[k] = 2*i + j + 1;
            k--;
            }
        }
    }
if (num_fit_params == 0)
    return;

for (i=1; i<=data.points_scan; i++){
    x[i] = (float)(i) ;
    y[i] = (float)*(data.data[0]+i-1);
    sig[i] = (float)data.param[7];
    }

/*      Next the Levenberg-Marquardt fitting method is used in a loop
to determine best fit values as outlined in "Numerical Recipes in C"
pp. 545-546.  The procedure must be carried out at least twice.
Convergence is considered as achieved when the value of chi-squared
has decreased by less than .1 or by a fractional amount less than
1E-3.    As a result, the numerical routine mrqmin() is called first
in order to initialze the routine (as indicated by *alamda=-1); then
a loop is entered which calls mrqmin() until convergence is achieved.
Then mrqmin() is called one last time with *alamda set to 0.  This
causes mrqmin to set covar and alpha to the covariance and curvature
matrices for the converged parameter values.
*/

alamda = -1;
mrqmin(x, y, sig, data.points_scan, a, ma, lista, num_fit_params,
                                covar, alpha, &chisq, funcs, &alamda);
dchisq = .2 + chisq;
do{
    count++;
    oldchisq = chisq;
    olamda = alamda;
    mrqmin(x, y, sig, data.points_scan, a, ma, lista, num_fit_params,
                                covar, alpha, &chisq, funcs, &alamda);
    if (olamda > alamda)
        dchisq = oldchisq - chisq;
    } while (dchisq > .1 && (dchisq/(chisq + 1)) > .001 &&
                                    count < 10000);
alamda = 0;
```

```
mrqmin(x, y, sig, data.points_scan, a, ma, lista, num_fit_params,
                            covar, alpha, &chisq, funcs, &alamda);

data.param[0] = (double)a[1];
data.param[1] = (double)a[2];
data.param[5] = (double)a[3];
data.param[6] = (double)a[4];
data.param[8] = (double)chisq;

data.param[0] *= data.time_per_point;
data.param[1] *= data.time_per_point;
for(i=1; i<=data.points_scan; i++){
    funcs(x[i], a, &dummy, dummya, ma);
    *(data.data[1]+i-1) = (double)dummy;
    }
calc_err();
reset_control_panel();
plot();

free_vector(x, 1, data.points_scan);
free_vector(y, 1, data.points_scan);
free_vector(sig, 1, data.points_scan);
free_ivector(lista, 1, ma);
free_matrix(covar, 1, ma, 1, ma);
free_matrix(alpha, 1, ma, 1, ma);
free_vector(a, 1, ma);
}
```

LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
TECHNICAL INFORMATION DEPARTMENT
BERKELEY, CALIFORNIA 94720