Modeling Cathodoluminescence of Rare Earth Ions with a Two Level System

Leon Maurer

March 10, 2008

Abstract

This is a summary of the math that I did while investigating this two level model, and how I took the data to test it with. This paper provides the theoretical background needed to understand the programs I wrote to fit the model to experimental data, as well as information on the automated data collection setup. The ultimate goal of this model is to find the time constant for the decay rate – τ, so that we can compare this to the known lifetime of the REI's excited state. If this is shorter than τ, then we have found evidence of a trap.

1 The Model

1.1 Differential Form

Our model is a simple two level system – N total ions, N_e excited ions, and N_g ions in the ground state. Then we have a pump rate p – which is proportional to beam current – that accounts for atom excitation, and a decay rate $k = \frac{1}{\tau}$ for the transition back to the ground state. With these parameters we can create two differential equations, one for when the beam is bombarding a region, and one for when it is not.

$$\tfrac{d}{dt}N_e = pN_g - kN_e = p(N - N_e) - kN_e = -(p+k)N_e + pN \tag{1}$$

$$\tfrac{d}{dt}N_e = -kN_e \tag{2}$$

I'll call the first equation (and other forms of it) the grown the equation and call the second the decline equation.

1.2 Exponential Form

The above equations are easy to solve. For the case where the first equation starts from zero and the second starts from saturation, these become:

$$N_e = \tfrac{pN}{p+k}\left(1 - e^{-(p+k)t}\right) \tag{3}$$

$$N_e = \tfrac{pN}{p+k}e^{-kt} \tag{4}$$

In general, the initial conditions will be somewhere between zero and saturation. We can work that in to the above equations in two ways. We can stick positive constants δ_1 and δ_2 in the exponent to shift the curve.

$$N_e = \tfrac{pN}{p+k}\left(1 - e^{-(p+k)(t+\delta_1)}\right)$$
$$N_e = \tfrac{pN}{p+k} e^{-k(t+\delta_2)}$$

If we known what the initial conditions are – call them s_1 and s_2, normalized to values between 0 and 1 (saturation), we can write the equation as follows:

$$N_e = \tfrac{pN}{p+k}\left(1 - (1 - s_1)e^{-(p+k)t}\right)$$
$$N_e = \tfrac{pN}{p+k} s_2 e^{-kt}$$

I prefer the former representation for working with the equations by hand, but I use the latter in my program because the initial conditions are already known – in this paper I switch between the two.

2 Using the Model

2.1 Spot Mode

In spot mode, the beam stays on one area, so only the growth equation is needed. The steady state (which is reached quickly) is:

$$N_e \approx \lim_{t\to\infty} \tfrac{pN}{p+k}\left(1 - e^{-(p+k)t}\right) = \tfrac{pN}{p+k}$$

However, this only tells us about the ratio $\frac{k}{p}$, but we want to measure k alone.

$$N_e \approx \tfrac{pN}{p+k} = \tfrac{N}{1+\frac{k}{p}}$$

2.2 Line Mode

In line mode the beam sweeps back and forth over a distance W in a time T. If the beam has width W_b then the beam bombards a point for $T_b = T\frac{W_b}{W}$ and does not for a time $T_o = T - T_b$.[1] Because the beam is alternately exciting and not exciting a region, N_e will be a combination of both the growth and decline

[1] This isn't quite right for regular line mode. It would be if the beam moved from point a to point b, and then restarted at point a again (this behavior could be generated by a sawtooth wave). Because the beam is moved by a triangle wave, it moves from a to b, and then from b to a before repeating. This means that the time between bombardments varies. Consider a point close to b. It will be bombarded at the end of the a to b sweep, and then soon again at the beginning of the b to a sweep, but then it will not be bombarded again for a longer time. My model does not take this in to account – it might be complicated to work in, and it might not change the result much. The automated data collection method that is discussed later solves this problem by using a sawtooth wave.

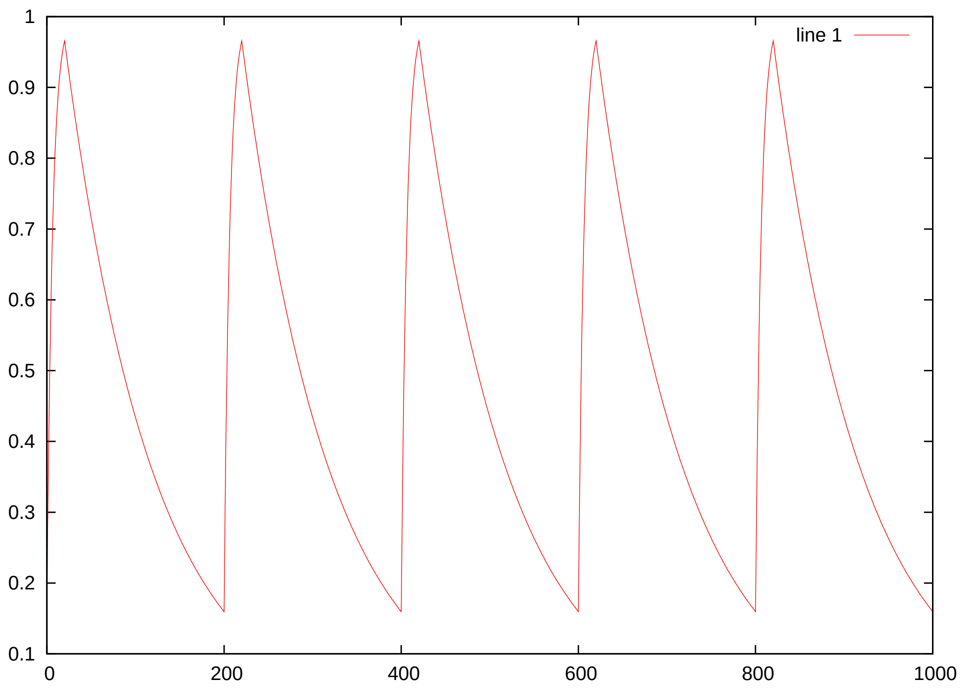

Figure 1: An example graph of intensity versus time in the steady state.

formulas with different initial conditions. We are interested in the steady state – where the values at the start and end of a sweep are the same. The condition for this is:

$$1 - e^{-(p+k)\delta_1} = e^{-k(T_o+\delta_2)} \tag{5}$$

$$1 - e^{-(p+k)(T_b+\delta_1)} = e^{-k\delta_2} \tag{6}$$

The first equation says that the value at the start of the growth is the same as at the end of the decline, while the second says that the value at the end of the growth should be the same as at the start of the decline. Unfortunately, I haven't found a way to solve for δ_1 and δ_2 algebraically – there will be more on how I solve it numerically later.

For fitting purposes, what we are actually interested in is the time average of N_e over one cycle, which should be proportional to the observed intensity:

$$I \propto \tfrac{1}{T}\tfrac{pN}{p+k}\left(\int_{\delta_1}^{\delta_1+T_b}\left(1 - e^{-(p+k)t}\right)dt + \int_{\delta_2}^{\delta_2+T_o} e^{-kt}dt\right) \tag{7}$$

3 Approximations

While it is possible to just solve everything numerically, it is useful to make approximations to get some intuition about the model. I have worked on two different approximations. The first is one I came up with, and the second is the one that Penn did. The first turns out to be a more general (and thus slightly more complicated) version of the second.

3.1 To Zero

The main assumption in this approximation is that in the steady state, during the decline part of the curve, the value gets very close to zero. This means, that the growth curve starts out close to zero – in other words $s_1 \approx 0$ or $\delta_1 \approx 0$. By the second steady state equation, this implies $s_2 = e^{-k\delta_2} = 1 - e^{-(p+k)T_b}$. Now we can take the time average.

$$\begin{aligned} I &\propto \tfrac{1}{T}\tfrac{pN}{p+k}\left(\int_0^{T_b}\left(1-e^{-(p+k)t}\right)dt + \int_0^{T_o}\left(1-e^{-(p+k)T_b}\right)e^{-kt}dt\right) \\ &= \tfrac{pN}{p+k}\left(\tfrac{T_b}{T} + \tfrac{1}{T}\left(1-e^{-(p+k)T_b}\right)\left(\tfrac{1}{k}\left(1-e^{-kT_o}\right) - \tfrac{1}{p+k}\right)\right) \end{aligned}$$

The $-\frac{1}{p+k}$ comes from the growth term and can be discarded since $p \gg k \Rightarrow \frac{1}{k} \gg \frac{1}{p+k}$ in our experiments. Now, it helps to think of N_e as the number of excited atoms per unit length – to get the total number of excited atoms we can multiply by W.

$$I \propto \tfrac{pN}{p+k}\left(W\tfrac{T_b}{T} + \tfrac{W}{Tk}\left(1-e^{-(p+k)T_b}\right)\left(1-e^{-kT_o}\right)\right)$$

But $\frac{T_b}{T} = \frac{W_b}{W}$ so:

$$I \propto \tfrac{pN}{p+k}\left(W_b + \tfrac{W}{Tk}\left(1-e^{-(p+k)T_b}\right)\left(1-e^{-kT_o}\right)\right)$$

Finally, it makes sense to define $A_o = W_b\frac{pN}{p+k}$ as the number of excited ions in spot mode, and $I_o = \frac{W}{T}\frac{pN}{p+k}$ as the number of excited ions if the entire line were bombarded at once (as if it were all in spot mode). Then our approximation becomes:

$$I \propto A_o + I_o\tfrac{1}{Tk}\left(1-e^{-(p+k)T_b}\right)\left(1-e^{-kT_o}\right) \tag{8}$$

3.2 To One

This approximation is that the the pump rate is very high and, during the time when the beam is bombarding the region, N_e grows to $\frac{pN}{p+k}$ almost instantly. Then our time average becomes:

$$I \propto \int_0^{T_b}\tfrac{pN}{p+k}dt + \int_0^{T_o}\tfrac{pN}{p+k}e^{-kt}dt = \tfrac{pN}{p+k}\left(T_b + \tfrac{1}{k}\left(1-e^{-kT_o}\right)\right) \tag{9}$$

After evaluating this, multiplying through by $\frac{W}{T}$, and preforming the same substitutions as above, the time average becomes:

$$I \propto A_o + I_o \tfrac{1}{Tk}\left(1 - e^{-kT_o}\right) \tag{10}$$

Although the assumption is quite different from the previous one, the results are very similar. This should be no surprise since our assumption implies that $1 - e^{-(p+k)T_b} \approx 1$, which causes this approximation to fall out from the previous one.

If we make the substitution $f = \frac{1}{T}$ and the approximation that $T \approx T_o$ (which is reasonable if $W \gg W_b$). then we can rewrite it as:

$$I \propto A_o + I_o \tfrac{f}{k}\left(1 - e^{\frac{-k}{f}}\right) \tag{11}$$

This is the approximation that Penn made in his thesis. It didn't fit my data as well, possibly because he used a higher current than I did.

4 Doing Things Numerically

The tricky part is finding the steady state – once s_1 and s_2 are known, the integral can be evaluated exactly.

I wrote my program in Octave – an open source MATLAB clone, and it included a multivariate solving function. I could just tell it to solve the steady state conditions and it would spit out δ_1 and δ_2. However it didn't always find an answer in extreme situations, and it could be a little slow. I figured that, since I knew how the growth and decline functions behaved, I could find a better way to solve for the steady state. My first attempt was very simple. Start the growth function from zero and find the value after T_b seconds. Then start the decline function from there and find the value after T_o seconds. Then repeat the process many times – starting from the previous value rather than zero – until the value at the start of the growth function stabilizes.

This method worked – convergence was guaranteed (which was a problem with the built in solver), but it was very slow. It occurred to me that 0 was the zeroth order Taylor expansion of the growth function:[2]

$$1 - e^{-(p+k)t} = 0 + (p+k)t - \tfrac{(p+k)^2}{2}t^2 + \tfrac{(p+k)^3}{2}t^3 \ldots$$

So I tried higher order Taylor expansions. Taking the first order Taylor expansion and plugging it in to the steady state equations yields:

$$(p+k)\delta_1 = e^{-k(T_o+\delta_2)} = e^{-kT_o}e^{-k\delta_2}$$
$$(p+k)(T_b+\delta_1) = e^{-k\delta_2}$$

The second equation can be substituted into the first, yielding:

$$(p+k)\delta_1 = e^{-kT_o}(p+k)(T_b+\delta_1)$$

$$\delta_1 = \tfrac{e^{-kT_o}T_b}{1-e^{-kT_o}} = \tfrac{T_b}{e^{kT_o}-1} \tag{12}$$

[2]It made more sense to Taylor expand the growth function than the decline function because $t_b < t_o$, so the growth function would stay closer to zero – the point the expansion was about.

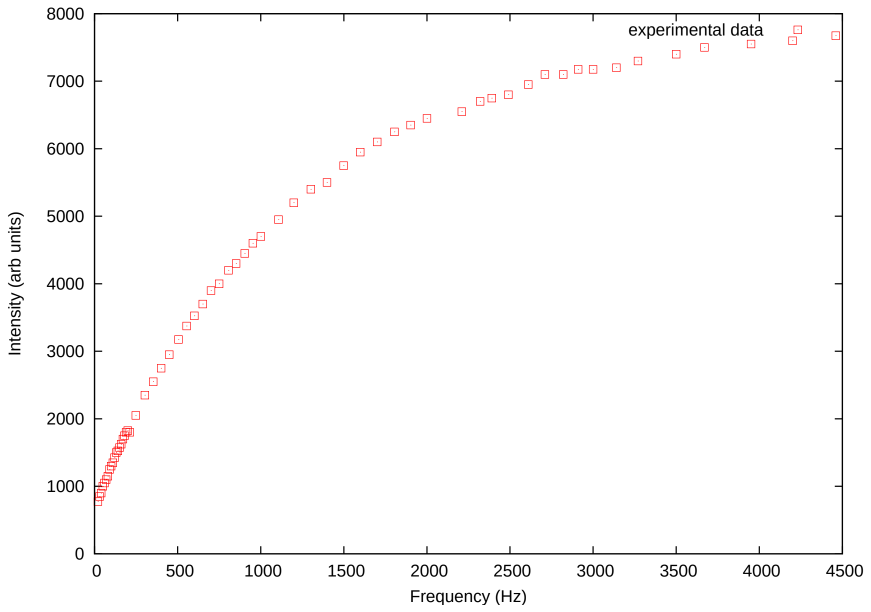

Figure 2: Data taken from a line mode experiment done by hand.

I used this as a starting guess instead of 0 in the previous solution algorithm. It worked faster, but I figured that I could do one better – using the third order Taylor approximation.[3] The idea is the same but it is much messier – I used Maple (a computer algebra system) to solve the resulting cubic equation for the steady state. That approximation proved very accurate – usually it was close enough that it did not have to use the cycling method.

5 Fitting Experimental Data

5.1 Taking Data

5.1.1 By hand

This process is long, tedious, and full of ways to introduce errors. However, it gives you an appreciation of what's going on. Information on the automated process is in the next sub-subsection.

Taking line mode measurements is accomplished through a function generator driving the x or y beam scan input of the SEM. The function generator should be put in triangle wave mode with a high peak to peak voltage (I used 20 volts – the highest the generator would go). Then the frequency should be set very low, so that one can observe the intensity throughout the sweep, and the sample should be moved until a region is found where the intensity is relatively constant throughout the sweep. Finally, the frequency should be raised until any remaining positional intensity inconsistencies are no longer visible – start taking measurements from there (this was around 20 Hz for me).

[3]The second order wouldn't do because of it's shape – it couldn't cover all possible starting values. If you graph it along with the growth function, you'll see why.

5.1.2 Automatically

This is much easier and faster than the method outlined above. First, get the monochromator positioned at the correct frequency. Start up linemode.vi and enter in the relevant parameters (start frequency, end frequency, frequency step size, number of samples to take at each frequency, and the time between the samples). When you run the program, it will ask you to blank the beam – this allows the program to measure the background noise. It will subtract this number from all the following measurements so that you don't have to. Once it has found the background noise, it will prompt you to unblank the beam so that it can begin. After that, you can just sit back and relax until it's done. It displays the current reading and the average reading for the previous frequency. Once it is done, it will display the data (although the scales may be off) and it will prompt you to save two files. The first one includes all the readings, while the second only includes the average of all the samples at each frequency. You can name these files whatever you'd like. They are just plane tab deliminated text files. The first column is the frequency, the second is the readout from the temperature sensor, and the following column or columns are the intensity data. Note that the data taken this way is not directly comparable to the data taken in the previous method because the units of intensity are different.

The setup is fairly straightforward. A signal generator and a multimeter are attached to the computer by GPIB. The labview program sends some cryptic commands to the devices (both manuals can be found online, and they list the commands), and they do the computer's bidding. There are a few things to note about the equipment.

The signal generator is set to produce a sawtooth wave, to get around the problems with triangle waves discussed in an earlier footnote. However, the signal generator's maximum sawtooth wave frequency is lower than it's maximum frequency for other waves (like sine and triangle). However, this maximum frequency is high enough that it shouldn't prove a problem – see the manual for details.

Also, the multimeter can collected data in three different modes – fast, medium, and slow. In each mode, the multimeter can only do so many readings per second. The program sets it to medium, which I believe allows up to 16 readings per second – again, consult the manual to make sure.

Finally, make sure the gain on the current amplifier is ok. If the overload light is on, then turn it down. If that doesn't help, turn it to zero mode, and tweak the small adjustment screw (in a hole on the front panel) until you get it so that the light turns off. This procedure is outlined in its manual.

5.2 Fitting Data

5.2.1 With Octave

Fitting was done by least squares – the difference between the predicted and actual data was squared and summed. This was fed in to the built in minimization function in Octave, which used one of several algorithms. Here is an example Octave session

```
octave:1> source("./unified_beam_model.m")
octave:2> load hzdata7_10.txt
octave:3> fmins("numerical_model",[100,2000,2000])
```

The first line loads the program. The second loads the data. The third runs the minimization function on the model we described – the function numerical_model returns the sum of the squares. In the brackets

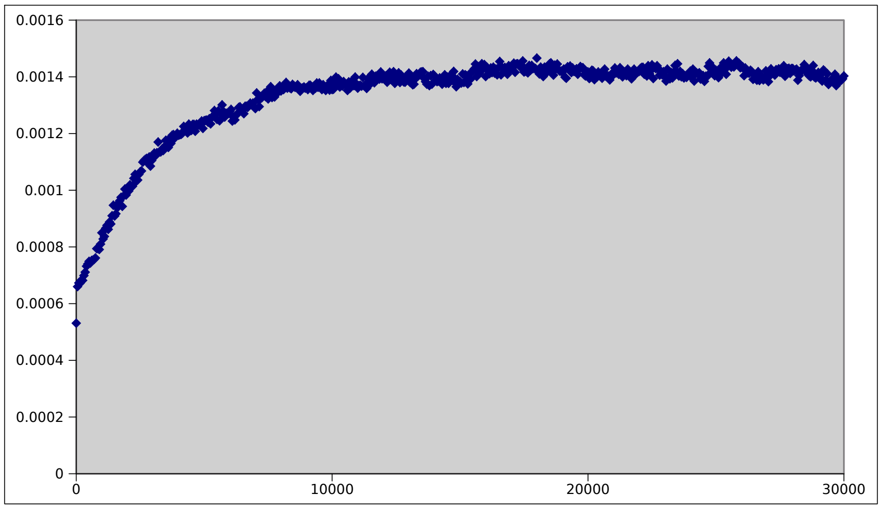

Figure 3: Data taken from an automated line mode experiment. The data doesn't look as clean as the data taken by hand, but that's because the computer is capturing the essential grittiness of the situation better than a human can.

is a (bad) starting guess for N,[4] p, and k. The program then runs for a while before spitting out the fit parameters (in the same order as before).

```
ans =

   7.6627e+02   1.6266e+05   1.4763e+04
```

Sometimes the default fitting algorithm doesn't do a good job (you can see if the fit is good from the graph it produces). In that case, you can use a different algorithm – which is slower but more accurate – using the previous answer as your starting point.

```
octave:4> fmins("numerical_model",ans,[0,0,0,0,0,2])
```

At last count, I have made 11 functions like numerical_model – they are mostly variations on a theme. For instance numerical_model_fixed_k allows me to keep k constant and try to fit p and N around it. There is also rational_model (and several variations) for fitting spot mode data.[5]

A useful fit is numerical_model_with_width. W is a parameter in our model, but it's hard to measure directly, so it can make sense to put it in the model (or, more specifically, the relative size $\frac{W}{W_b}$). Using this model, on the data:

[4]This isn't really N, but N together with other proportionality constants.
[5]For more information on these, see the code.

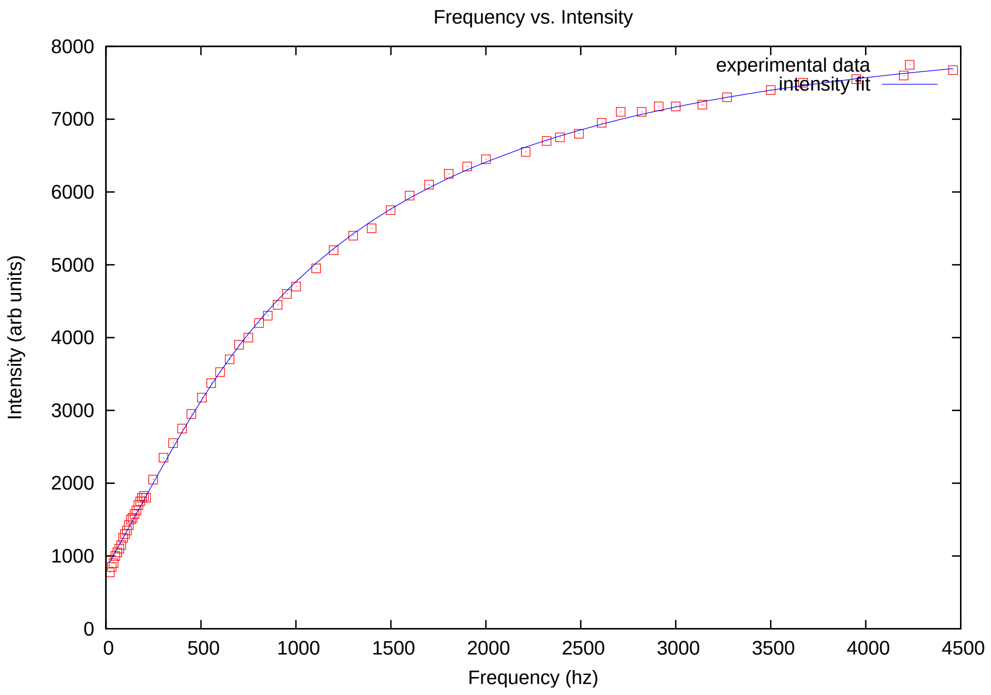

Figure 4: The experimental data with the fit provided by the below parameters.

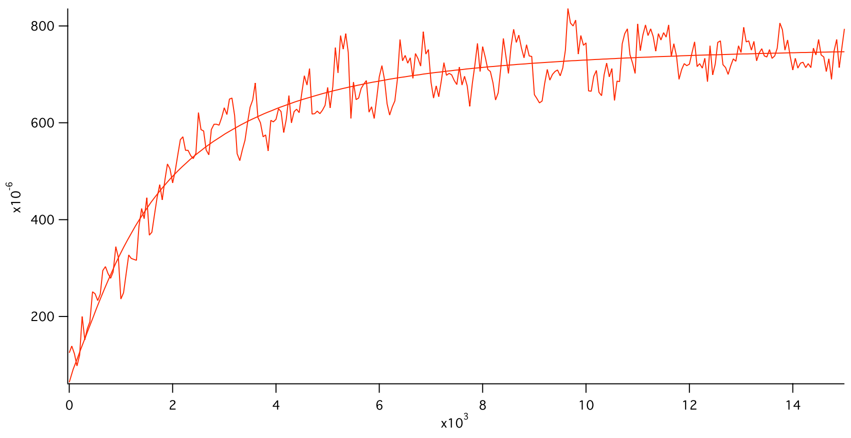

Figure 5: A fit done by Igor Pro.

```
octave:6> fmins("numerical_model_with_width",[500,1.e5,1e4,1000],[0,0,0,0,0,2])
...
ans =
   7.7521e+02   1.4197e+05   1.2962e+04   8.6129e+01
```

Here $\frac{W}{W_b}$ is 86.129 – in the previous case it was 100 (a default value that I chose). It is interesting to note that $\frac{k}{p}$ is nearly the same in both cases, and that both p and k decreased by approximately $100 - 86.129 = 13.871$ percent. My testing shows that $\frac{p}{k}$ is almost constant for all $\frac{W}{W_b}$ except when W_b gets close to W. The fitted W is smaller than my estimates by around 2 orders of magnitude – part of the problem is that a good fit can be found for about any value of W.

If $\frac{W}{W_b} \approx 100$ then $k \approx 1.4e + 04$ so $\tau \approx 71e - 6$ seconds. This is smaller than the known lifetime of the REI's excited state, which is not evidence for a trap (it is evidence that I should try taking some more measurements to get better data).

5.2.2 With Igor Pro

First, open up or create an Igor experiment that has the fitting code in it. Then, load the data from one of the files with the averaged values by going to the Data Menu → Load Waves → Load Delimited Text and selecting the file you want. Choose the names for you waves (you can choose to skip importing the temperature column). Then go to Analysis → Curve Fitting. In the "Function and Data" tab, select "linemode" as the function (and the appropriate X and Y waves). In the "Coefficients" tab, you must enter an initial guess (w_0 is p, w_1 is k, and w_2 is N). Note that Igor is much less forgiving than the lenient octave fit function, and Igor doesn't do as many passes to get it right, so a reasonable guess is useful (you'll get the hang of it). Then just sit back and relax – fitting will take some time.

The code itself is a simplified, but straightforward, port of the octave code. The two languages have some annoying differences that (like, in how they handle array indicies and global variables) and the Igor manual wasn't always helpful, so porting was surprisingly tricky, and it had some strange bugs for a while, but I think that they're all ironed out. However, I didn't have enough time to port the faster fitting code, so the Igor code could be sped up significantly. Also, I only ported the linemode fit – the spot mode fit can be made through the Curve Fitting window without resorting to real programming.

The Igor code

```
#pragma rtglobals=1 // Use modern global access method.

Function growth(start_value,t)
Variable start_value
Variable t
NVAR pump
NVAR k
return 1-(1-start_value)*exp(-(pump+k)*t)
End

Function decline(start_value,t)
Variable start_value
Variable t
NVAR k
return start_value*exp(-k*t)
End

Function growth_int(start_value,t)
Variable start_value
Variable t
NVAR pump
NVAR k
return t+(1-start_value)/(pump+k)*(exp(-(pump+k)*t)-1)
End

Function decline_int(start_value,t)
Variable start_value
Variable t
NVAR k
return start_value/k*(1-exp(-k*t))
End

Function whole_integral(start_state)
Variable start_state
NVAR to
NVAR tb
return  growth_int(start_state,tb) + decline_int(growth(start_state,tb),to)
End

Function itr_solve(start_value)
Variable start_value
Wave new_value
Wave old_value
NVAR tb
   NVAR to
   NVAR accuracy
   new_value[0] = start_value
   new_value[1] = growth(new_value[0],tb)
   do
   old_value = new_value
   new_value[0] = decline(new_value[1],to)
   new_value[1] = growth(new_value[0],tb)
   while(abs((old_value[1]-new_value[1])/new_value[1])>accuracy)
return new_value[0]
End

Function linemode(w,x) : FitFunc
Wave w
Variable x
Variable sol
Variable beam_width = 1
Variable sweep_width = 100
Variable speed = sweep_width * x
Variable n = w[2]
Variable val
NVAR tb
NVAR to
NVAR t
NVAR pump
NVAR k
tb = beam_width/speed
to = (sweep_width-beam_width)/speed
t = sweep_width/speed
pump = w[0]
k = w[1]
if (x==0) //beam isn’t moving
val = beam_width*n*pump/(pump+k)
else
sol = itr_solve(0)
val = n*(sweep_width/beam_width)*whole_integral(sol)/t
endif
if (val == NaN)
print "OMG"
endif
print x, val, sol
return val
End
```

The Octave code.

```
#constants for the exponential growth and decline
global p=10;
global k=.1;
global tb=0;
global to=0;
global accuracy = .00001;
global data;
global old;
global oldp=0;
global oldk=0;
global beam_width = 1 ;
global sweep_width = 100;

#so we can see output
more off

#exponential growth
function value = growth(start_value,time)
global p;
global k;
value = 1-(1-start_value)*exp(-(p+k)*time);
endfunction;

#exponential decline
function value = decline(start_value,time);
global k;
value = start_value*exp(-k*time);
endfunction;

#stretched exponential growth
function value = sgrowth(start_value,time,power)
global p;
global k;
value = 1-(1-start_value)*exp(-((p+k)*time)^power);
endfunction;

#stretched exponential decline
function value = sdecline(start_value,time,power);
global k;
value = start_value*exp(-(k*time)^power);
endfunction;

#integral of growth
function area = growth_int(start_value,time)
global p;
global k;
area = time+(1-start_value)/(p+k)*(exp(-(p+k)*time)-1);
endfunction;

#integral of decline
function area = decline_int(start_value,time)
global k;
area = start_value/k*(1-exp(-k*time));
endfunction;

#integral of sgrowth
function area = sgrowth_int(start_value,time,power)
area = quad(inline('sgrowth(start_value,t,power)',"t"),0,time);
endfunction;

#integral of sdecline
function area = sdecline_int(start_value,time,power)
   area = quad(inline('sdecline(start_value,t,power)',"t"),0,time);
endfunction;

function area = whole_integral(start_state)
  global to;
  global tb;
  area = (growth_int(start_state(1),tb)+decline_int(start_state(2),to));
endfunction;

function area = swhole_integral(start_state,powers)
  global to;
  global tb;
  area = (sgrowth_int(start_state(1),tb,powers(1))+sdecline_int(start_state(2),to,powers(2)));
endfunction;

#finds the steady state for the growth/decline cycle by repeting until
#the start value converges to within the specified relative accuracy
function new_value = itr_solve(start_value)
  global tb;
  global to;
  global accuracy;
  if !(abs(start_value) <= 1)
    printf("Inappropriate start value: %f\n",start_value);
    start_value = 0;
  endif
  new_value(1) = start_value;
  new_value(2) = growth(new_value(1),tb);
  do
    old_value = new_value;
    new_value(1) = decline(new_value(2),to);
    new_value(2) = growth(new_value(1),tb);
  until((new_value(1)==0)||(abs((old_value(1)-new_value(1))/new_value(1))<accuracy))
endfunction;

#uses a linear approximation to find a decent starting guess
function guess = linear_approx()
  global tb;
  global to;
  global k;
  guess = growth(0,tb/(exp(k*to)-1));
endfunction;

#by approximating growth by a taylor expansion of order 3, and then
#solving the steady state equations we get an algebraic solution that is
#very accuare for start values <.6 or so, this cut the number of loops
#we need to do dramaticially for high beam speeds

#TODO: the expression can be factored more, which will make the equation
#simpler and the computation faster

#TODO: why does it guess a start value of 1 for low speed? Why does it
#sometimes give NaN for low speed?
function guess = cubic_approx()
  global tb;
  global to;
  global k;
  global p;
  guess = growth(0,(sqrt(5*exp(2*k*to)+(2*exp((-p-k)*tb)-12)*exp(k*to)+2*exp(2*(-p-k)*tb)-6*exp((-p-k)*tb)+9)/((p+k)^3*(exp(k*to)-exp((-p-k)*tb
endfunction;

#TODO: find a way to have the computer make an intellegent choice
#between these 3 solution methods
function sol = solve()
  guess = linear_approx();
  if guess > .1
    guess = cubic_approx();
  endif
  if !(abs(guess) <= 1)
    printf("Inappropriate start value: %f\n",guess);
    guess = 0;
  endif
#printf("%f",guess);
  sol=itr_solve(guess);
```

```
#printf(" %f\n",sol(1));
endfunction;

function new_value = sitr_solve(start_value,power1,power2)
  global tb;
  global to;
  global accuracy;
  if !(abs(start_value) <= 1)
    printf("Inappropriate start value: %f\n",start_value);
    start_value = 0;
  endif
  new_value(1) = start_value;
  new_value(2) = sgrowth(new_value(1),tb,power1);
  do
    old_value = new_value;
    new_value(1) = sdecline(new_value(2),to,power2);
    new_value(2) = sgrowth(new_value(1),tb,power1);
  until((new_value(1)==0)||(abs((old_value(1)-new_value(1))/new_value(1))<accuracy))
endfunction;


function showcycle(svals)
  global tb;
  global to;
  for t=0:1:1000
    time = t*(tb+to)/1000;
    if time<tb
      values(t+1)=growth(svals(1),time);
    else
      values(t+1)=decline(svals(2),time-tb);
    endif
  endfor
  plot((0:1000),values);
  printf("avg: %g\n",whole_integral(svals)/(tb+to));
endfunction;

function showsecond(svals)
  global tb;
  global to;
  time = -1/1000;
  for t=0:1:1000
    time = time + 1/1000;
    if time>(to+tb)
      time = time - to - tb;
    endif
    if time<tb
      values(t+1)=growth(svals(1),time);
    else
      values(t+1)=decline(svals(2),time-tb);
    endif
  endfor
  plot((0:1000),values);
endfunction;

function main
global tb;
global to;
global beam_width;
global sweep_width;
speed = sweep_width * input("Enter frequency:\n");
tb = beam_width/speed; #time spent on on sector
to = (sweep_width-beam_width)/speed; #time spent on other sectors
t = sweep_width/speed; #total time for one sweep
start_state = solve();
printf("Spd: %d S_vals: [%f, %f] Area: [%f, %f] Avg: %f\n",speed,start_state(1),start_state(2),growth_int(start_state(1),tb),decline_int(start_
endfunction;

function sweep
```

```
global tb;
global to;
global p;
global k;
global beam_width;
global sweep_width;
printf("%f, %f\n",p,k);
for hz=1:30:301
  speed = hz * sweep_width;
tb = beam_width/speed; #time spent on on sector
to = (sweep_width-beam_width)/speed; #time spent on other sectors
t = sweep_width/speed; #total time for one sweep
start_state = solve();
printf("Spd: %d S_vals: [%f, %f] Area: [%f, %f] Avg: %f\n",speed,start_state(1),start_state(2),growth_int(start_state(1),tb),decline_int(start_
endfor
endfunction;

function plot_sweep
  global tb;
  global to;
  global p;
  global k;
  global beam_width;
  global sweep_width;
  printf("p, k: %f, %f\n",p,k);
#this is for speed=0 -- one point is saturated (has value 1) the rest untouched (value 0)
  results(1)=1;
  for i=1:5000
    speed = i * sweep_width;
    tb = beam_width/speed; #time spent on on sector
    to = (sweep_width-beam_width)/speed; #time spent on other sectors
    t = sweep_width/speed; #total time for one sweep
    start_state = solve();
    results(i+1)=(sweep_width/beam_width)*whole_integral(start_state)/t;
    printf("%i %i %g\n",i,speed,results(i+1));
  endfor
  plot((0:5000)',results');
endfunction;

function sum = rational_model(x)
  global data;
  sum = 0;
  x = abs(x);
  n = x(3);
  k = x(2);
  for i = 1:(size(data))(1)
    p = x(1)*data(i,1);
    results(i) = p*n/(p+k);
    sum = sum + (data(i,2)-50-results(i))^2;
  endfor;
  printf("%g %g %g %g\n", x(1), x(2), x(3), sqrt(sum/(size(data))(1)));
  hold off;
  plot(data(:,1),data(:,2),"x;experimental data;");
  hold on;
  plot(data(:,1),results+50,"3;intensity fit;");
  hold off;
endfunction;

function sum = rational_model_with_k(x)
  global data;
  sum = 0;
  x = abs(x);
  n = x(1);
  k = 1.4761e4;#5000;
  for i = 1:(size(data))(1)
    p = x(2)*data(i,1);
    results(i) = p*n/(p+k);
    sum = sum + (data(i,2)-50-results(i))^2;
  endfor;
  printf("%g %g %g\n", x(1), x(2), sqrt(sum/(size(data))(1)));
  hold off;
  plot(data(:,1),data(:,2),"x");
  hold on;
  plot(data(:,1),results+50);
  hold off;
endfunction;

function sum = rational_model_with_k_p(x)
  global data;
  sum = 0;
  x = abs(x);
  n = x(1);
  k = 1.4761e4;
  for i = 1:(size(data))(1)
    p = 1.6276e5/.32*data(i,1);
    results(i) = p*n/(p+k);
    sum = sum + (data(i,2)-50-results(i))^2;
  endfor;
  printf("%g %g\n", x(1), sqrt(sum/(size(data))(1)));
  hold off;
  plot(data(:,1),data(:,2),"x");
  hold on;
  plot(data(:,1),results+50);
  hold off;
endfunction;

function sum = two_site_rational_model_with_k_p(x)
  global data;
  sum = 0;
  x = abs(x);
  n1 = x(1);
  n2 = x(2);
  k = 5000;
  for i = 1:(size(data))(1)
    p1 = 3.2635e5/.32*data(i,1);#x(3)*data(i,1);
    p2 = 2.4105e4/.32*data(i,1);#x(4)*data(i,1);
    results(i) = p1*n1/(p1+k) + p2*n2/(p2+k);
    sum = sum + abs(data(i,2)-50-results(i));
  endfor;
  printf("%g %g %g %g %g\n", x(1), x(2), sqrt(sum/(size(data))(1)));
  hold off;
  plot(data(:,1),data(:,2),"x");
  hold on;
  plot(data(:,1),results+50);
  hold off;
endfunction;

function sum = simple_exp_model(x)
  global data;
  global p;
  global k;
  global beam_width;
  global sweep_width;
  sum = 0;
  x = abs(x); # no funny business
  n = x(1);
  k = x(2);
  for i = 1:(size(data))(1)
    f = data(i,1);
    result(i) = 50 + n*(beam_width + sweep_width*(f/k)*(1-exp(-k/f)));
    sum = sum + (data(i,2) - result(i))^2;
  endfor
  printf("%g %g %g\n", x(1), x(2), sqrt(sum/(size(data))(1)));
  hold off;
  plot(data(:,1),data(:,2),"x;experimental data;");
  hold on;
  plot(data(:,1),result,"3;intensity fit;");
  hold off;
endfunction;

function sum = exp_model(x)
  global data;
  global p;
  global k;
  global tb;
  global to;
  global beam_width;
  global sweep_width;
  sum = 0;
  x = abs(x); # no funny business
  n = x(1);
  k = x(2);
  for i = 1:(size(data))(1)
    f = data(i,1);
    speed = sweep_width*f;
    tb = beam_width/speed; #time spent on on sector
    to = (sweep_width-beam_width)/speed; #time spent on other sectors
    t = sweep_width/speed; #total time for one sweep
    result(i) = 50 + n*(beam_width + sweep_width*(f/k)*(1-exp(-k*to)));
    sum = sum + (data(i,2) - result(i))^2;
  endfor
  printf("%g %g %g\n", x(1), x(2), sqrt(sum/(size(data))(1)));
  hold off;
  plot(data(:,1),data(:,2),"x;experimental data;");
  hold on;
  plot(data(:,1),result,"3;intensity fit;");
  hold off;
endfunction;

function sum = complex_exp_model(x)
  global data;
  global p;
  global k;
  global tb;
  global to;
  global beam_width;
  global sweep_width;
  sum = 0;
  x = abs(x); # no funny business
  n = x(1);
  p = x(2);
  k = x(3);
  for i = 1:(size(data))(1)
    f = data(i,1);
    speed = sweep_width*f;
    tb = beam_width/speed; #time spent on on sector
    to = (sweep_width-beam_width)/speed; #time spent on other sectors
#    t = sweep_width/speed; #total time for one sweep
    result(i) = n*(sweep_width/beam_width)*whole_integral([0,growth(0,tb)])/(tb+to);
#    result(i) = n*(sweep_width/beam_width)*(tb + (1-exp(-(p+k)*tb))*((1-exp(-k*to))/k-1/(p+k)))/(to+tb);
    sum = sum + (data(i,2) - result(i))^2;
  endfor
  printf("%g %g %g %g\n", x(1), x(2), x(3), sqrt(sum/(size(data))(1)));
  hold off;
  plot(data(:,1),data(:,2),"x;experimental data;");
  hold on;
  plot(data(:,1),result,"3;intensity fit;");
  hold off;
endfunction;

function sum = numerical_model(x)
  global data;
  global p;
  global k;
  global tb;
  global to;
  global old;
  global oldp;
  global oldk;
  global beam_width;
  global sweep_width;
  sum = 0;
  x = abs(x); # no funny business
  n1 = x(1);
  p = x(2);
  k = x(3);
  for i = 1:(size(data))(1)
    if ((p != oldp) | (k != oldk))
      speed = sweep_width*data(i,1);
      tb = beam_width/speed; #time spent on on sector
      to = (sweep_width-beam_width)/speed; #time spent on other sectors
      t = sweep_width/speed; #total time for one sweep
      site1 = solve();
      old(i)=(sweep_width/beam_width)*whole_integral(site1)/t;
      sum = sum + (data(i,2)-50-n1*old(i))^2;
    else
      sum = sum + (data(i,2)-50-n1*old(i))^2;
    endif
  endfor
  printf("%g %g %g %g\n", x(1), x(2), x(3), sqrt(sum/(size(data))(1)));
  hold off;
  plot(data(:,1),data(:,2),"x;experimental data;");
  hold on;
  plot(data(:,1),n1*old+50,"3;intensity fit;");
  hold off;
  oldp=p;
  oldk=k;
endfunction;

function sum = numerical_model_with_width(x)
  global data;
  global p;
  global k;
  global tb;
  global to;
  global old;
#  global oldp;
#  global oldk;
  global beam_width;
  global sweep_width;
  sum = 0;
  x = abs(x); # no funny business
  n1 = x(1);
  p = x(2);
  k = x(3);
  sweep_width = x(4);
  for i = 1:(size(data))(1)
#    if ((p != oldp) | (k != oldk))
      speed = sweep_width*data(i,1);
      tb = beam_width/speed; #time spent on on sector
      to = (sweep_width-beam_width)/speed; #time spent on other sectors
      t = sweep_width/speed; #total time for one sweep
      site1 = solve();
      old(i)=(sweep_width/beam_width)*whole_integral(site1)/t;
      sum = sum + (data(i,2)-50-n1*old(i))^2;
#    else
#      sum = sum + (data(i,2)-50-n1*old(i))^2;
#    endif
  endfor
  printf("%g %g %g %g %g\n", x(1), x(2), x(3), x(4), sqrt(sum/(size(data))(1)));
  hold off;
  plot(data(:,1),data(:,2),"x;experimental data;");
  hold on;
  plot(data(:,1),n1*old+50,"3;intensity fit;");
  hold off;
#  oldp=p;
#  oldk=k;
endfunction;

function sum = numerical_model_fixed_k(x)
  global data;
  global p;
  global k;
  global tb;
  global to;
  global beam_width;
  global sweep_width;
  sum = 0;
  x = abs(x); # no funny business
  n = x(1);
  p = x(2);
  k = 1/200e-6;
  for i = 1:(size(data))(1)
      speed = sweep_width*data(i,1);
      tb = beam_width/speed; #time spent on on sector
      to = (sweep_width-beam_width)/speed; #time spent on other sectors
      t = sweep_width/speed; #total time for one sweep
      site1 = solve();
      old(i)=(sweep_width/beam_width)*whole_integral(site1)/t;
      sum = sum + (data(i,2)-50-n*old(i))^2;
  endfor
  printf("%g %g %g\n", x(1), x(2), sqrt(sum/(size(data))(1)));
  hold off;
  plot(data(:,1),data(:,2),"x");
  hold on;
  plot(data(:,1),n*old+50);
  hold off;
endfunction;

function sum = two_site_numerical_model_fixed_k(x)
  global data;
  global p;
  global k;
  global tb;
  global to;
  global beam_width;
  global sweep_width;
  sum = 0;
  x = abs(x); # no funny business
  n1 = x(1);
  n2 = x(2);
  p1 = x(3);
  p2 = x(4);
  k = 1/200e-6;
  for i = 1:(size(data))(1)
      speed = sweep_width*data(i,1);
      tb = beam_width/speed; #time spent on on sector
      to = (sweep_width-beam_width)/speed; #time spent on other sectors
      t = sweep_width/speed; #total time for one sweep
      p = p1;
      site1 = solve();
      p = p2;
      site2 = solve();
      old(i,2)=(sweep_width/beam_width)*whole_integral(site2)/t;
      old(i,1)=(sweep_width/beam_width)*whole_integral(site1)/t;
      sum = sum + (data(i,2)-50-n1*old(i,1)-n2*old(i,2))^2;
  endfor
  printf("%g %g %g %g %g\n", x(1), x(2), x(3), x(4), sqrt(sum/(size(data))(1)));
  hold off;
  plot(data(:,1),data(:,2),"x");
  hold on;
  plot(data(:,1),n1*old(:,1)+n2*old(:,2)+50);
  plot(data(:,1),n1*old(:,1));
  plot(data(:,1),n2*old(:,2));
  hold off;
endfunction;

function sum = snumerical_model(x)
  global data;
  global p;
  global k;
  global tb;
  global to;
  global beam_width;
  global sweep_width;
  sum = 0;
  x = abs(x); # no funny business
  n = x(1);
  p = x(2);
  k = x(3);
  power1 = x(4);
  power2 = x(5);
  for i = 1:(size(data))(1)
    speed = sweep_width*data(i,1);
    tb = beam_width/speed; #time spent on on sector
    to = (sweep_width-beam_width)/speed; #time spent on other sectors
    t = sweep_width/speed; #total time for one sweep
    site1 = sitr_solve(0,power1,power2);
    results(i)=n*(sweep_width/beam_width)*swhole_integral(site1,[power1,power2])/t;
    sum = sum + (data(i,2)-50-results(i))^2;
  endfor
  printf("%g %g %g %g %g %g\n", x(1), x(2), x(3), x(4), x(5), sqrt(sum/(size(data))(1)));
  hold off;
  plot(data(:,1),data(:,2),"x;experimental data;");
  hold on;
  plot(data(:,1),results+50,"3;intensity fit;");
  hold off;
endfunction;
```

An example Octabe datafile.

```
# Created by Leon 7/12/07
# name: data
# type: matrix
# rows: 62
# columns: 2
# Hz I
20.6 775
30.3 850
40 900
50 1000
60.1 1050
70.1 1100
80.1 1150
90.3 1250
100 1300
110.1 1350
120.5 1425
130.8 1500
140.3 1525
150.4 1575
160.9 1625
170.8 1700
180.2 1750
190 1800
200 1825
212 1800
248 2050
303 2350
353 2550
401 2750
450 2950
505 3175
554 3375
601 3525
651 3700
701 3900
750 4000
806 4200
852 4300
904 4450
953 4600
1001 4700
1107 4950
1199 5200
1302 5400
1399 5500
1499 5750
1599 5950
1701 6100
1805 6250
1902 6350
2000 6450
2210 6550
2320 6700
2390 6750
2490 6800
2610 6950
2710 7100
2820 7100
2910 7175
3000 7175
3140 7200
3270 7300
3500 7400
3670 7500
3950 7550
4200 7600
```

```
4460 7675
```