

Experiment 5: Temperature-Dependence of the HCl Rovib Spectrum: A Simulation

One of the virtues of spectroscopy is that an enormous amount of information is contained in a single picture (the spectrum). In this experiment we will assemble the components of a spectrum one piece at a time: first the molecular energy levels, then the positions of the absorbance peaks, and finally the intensities of the peaks. As a final part of the process, you will make a QuickTime movie of how your spectrum changes as the temperature is changed.

Numerical simulation is an invaluable tool for designing experiments and interpreting experimental data. In this example, we can easily change the temperature of the experiment over a range that far surpasses what is practical experimentally. Thankfully, structured programming in high-level data analysis programs such as Igor Pro has become very user-friendly. You have already seen how to use a few of the programming tools available in Igor Pro in the Igor tutorial. For this experiment, we will pull together some of these ideas, and introduce a few more tools.

Much of this experiment is taken from one developed and graciously offered by Dr. Melissa Hines, a professor of chemistry at Cornell University.

The Rovib Energies of HCl

A diatomic molecule can be modeled as two balls (atoms) connected by a spring (bond). If we consider the principle motions of the diatomic molecule to be bond vibration and molecular rotation¹, then the total energy of the system (the diatomic) is given by:

$$E_{rovib} = E_{rot} + E_{vib} \quad (1)$$

where E_{rot} is the energy due to rotations, and E_{vib} is the energy due to vibrations.

Recall that an absorption spectrum is essentially a plot of the amount of light absorbed versus the energy of the light absorbed. Therefore, in order to simulate the absorption spectrum of a diatomic molecule such as HCl, we must be able to calculate the energies of light that are absorbed. Since the energies of matter and light are quantized (can only have certain values), only certain wavelengths (or energies) of light can be absorbed. These were predicted by Neils Bohr based on the simple condition that:

$$E_{photon} = \Delta E_{rovib} \quad (2)$$

where E_{photon} is the energy of the absorbed light. For most molecules, ΔE_{rovib} is on the order of $\sim 10^3 \text{ cm}^{-1}$, which puts E_{photon} in the infrared region of the EM spectrum. Typically, E_{vib} is larger than E_{rot} by at least a factor of 10, so that rotational energies are actually small corrections to the vibrational energy spacings. Therefore, we'll start by examining the expressions for vibrational energy.

If the bond were a perfect spring (i.e.: vibrations were elastic and followed Hooke's law, $F = -kx$), then the Schrodinger wave equation could be solved exactly, and the vibrational energy of the molecule would be given by the expression for the energies of a quantum mechanical harmonic oscillator:

$$E_{HO}(v) = h\nu(v + 1/2) \quad (3a)$$

¹ The molecule can also have translational motion, but translational energies are so small that we can neglect them in this analysis.

where E_{HO} is in units of Joules, ν is the frequency of the vibration in Hertz, and $v = 0, 1, 2, \dots$ is the vibrational quantum number that characterizes the vibrational state of the diatomic (i.e.: how much vibrational energy it has). Since vibrational spectra are typically obtained in units of wavenumbers (cm^{-1}), we will convert all energies (in Joules) to wavenumbers by dividing by hc :

$$E_{HO}(v) = \bar{\nu}(v + 1/2) \quad (3b)$$

Real bonds, however, do not act like ideal harmonic oscillators because there are attractive forces between electrons on one atom and nuclei on the other, and repulsive forces between nuclei and between electrons. Real bonds are in fact "anharmonic" oscillators. The Schrodinger wave equation cannot be solved exactly using such real potentials. However, the true vibrational energy of a diatomic molecule can be fairly accurately modeled as a power series expansion of equation (3), in which each successive term is a smaller correction to the harmonic energy:

$$E_{vib}(v) = \bar{\nu}_v(v + 1/2) \quad (4)$$

where

$$\bar{\nu}_v = \bar{\nu}_e - \bar{\nu}_e x_e(v + 1/2) + \bar{\nu}_e y_e(v + 1/2)^2 - \dots \quad (5)$$

The constants in equation (5) characterize the way that the bond vibrates: $\bar{\nu}_e$ is the harmonic vibrational frequency (in wavenumbers), which depends on both the stiffness of the bond (described by the force constant, k), and the reduced mass of the molecule (μ):

$$\bar{\nu}_e = \frac{1}{2\pi c} \sqrt{\frac{k}{\mu}} \quad (6)$$

where μ is given by:

$$\mu = \frac{m_1 m_2}{m_1 + m_2} \quad (7)$$

and m_i is the mass of an individual atom. The constants x_e and y_e , characterize the degree of anharmonicity in the bond (i.e.: the inelasticity of the vibrations due to repulsions and attractions between nuclei and electrons).

Now let's consider rotational motions. The simplest picture of a rotating object is the rigid rotor picture, where two masses are connected by a rigid rod whose length is fixed. As for the harmonic oscillator, the Schrodinger wave equation (SWE) can be solved exactly for the rigid rotor, resulting in the rotational energies:

$$E_{RR}(J) = hBJ(J + 1) \quad (8)$$

where E_{RR} is in units of Joules, B is a sort of angular frequency of rotation in Hertz, and $J = 0, 1, 2, \dots$ is the rotational quantum number that characterizes the rotational state of the diatomic (i.e.: how much rotational energy it has). We know, however, that equation (8) will not be an accurate description of the rotational energies of our diatomic molecule since the "rod" (bond) connecting the two masses (atoms) actually vibrates (changes length). When a diatomic molecule rotates, it acts like a "non-rigid rotor". In fact, as it rotates, the bond can begin stretching *even more* due in part to centrifugal forces (think of spinning a rock on a slinky). This means that the rotational energies must

depend not only on the rotational quantum number, J , but also on the vibrational quantum number, v . As for the anharmonic oscillator, the SWE cannot be solved exactly for such a system. However, the true rotational energies can be approximated very well using the expression in equation (9) (now in wavenumbers, having divided by hc):

$$E_{rot}(v, J) = B_v J(J+1) - D_v [J(J+1)]^2 \quad (9)$$

where

$$B_v = B_e - \alpha_e(v + 1/2) + \dots \quad (10)$$

and

$$D_v = D_e - \beta_e(v + 1/2) + \dots \quad (11)$$

In much the same way that $\bar{\nu}_e$, x_e and y_e are the molecular constants that characterize the vibrations of a particular bond, B_e , α_e , D_e , and β_e characterize the rotations of a particular molecule. The equilibrium rotational constant, B_e , completely characterizes the rotations of a rigid rotor, and is related to the equilibrium bond length, r_e , according to equation (12):

$$B_e(\text{cm}^{-1}) = \frac{h}{8\pi^2 c \mu r_e^2} \quad (12)$$

The centrifugal stretching constant, D_e , is related to the equilibrium bond length as well as bond stiffness, and the coefficients α_e and β_e (termed coupling constants) characterize the extent of rotational-vibrational coupling (i.e.: how much the bond stretches in response to rotating).

If we combine equations (4) and (9), we obtain an expression for the total energy of a diatomic molecule due to rotations and vibrations (the “rovib” energy):

$$E_{rovib}(v, J) = \bar{\nu}_v(v + 1/2) + B_v J(J+1) - D_v [J(J+1)]^2 \quad (13)$$

The subscripts v on $\bar{\nu}_v$, B_v and D_v are reminders that the values of these parameters depend on the vibrational quantum number, and are not true constants like B_e .

TASK 1: Calculation of the Energy levels

- Create macro `NewConstants` to hold molecular constants
- Create functions to calculate $E_{rovib}(v, J)$
- Make plots of $E_{rovib}(0, J)$ vs. J and $E_{rovib}(1, J)$ vs. J

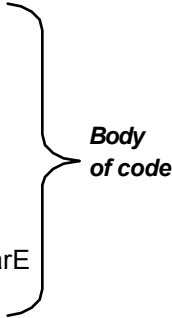
The construction of any molecular spectrum begins with the calculation of the energy levels available to the molecule. You will begin by creating a macro called `NewConstants` that will allow you to enter all of the values for the molecular constants that characterize HCl into Igor’s memory. (Although you could use the numerical values of these parameters in every function, it is much nicer to be able to change them in one central location.) Accepted values for all of the necessary molecular parameters can be found in the latest compilation by Huber and Herzberg², a portion of

² K. P. Huber and G. Herzberg, *Molecular Spectra and Molecular Structure: IV. Constants of Diatomic Molecules* (Van Nostrand, New York, 1979).

which is included as Appendix B (note that the values of some of the parameters have been hidden in the footnotes, and that you want the data for lowest energy electronic state, which is the state labeled X).

Start up Igor and open the Procedure window (the Procedure window is where all the function and macro code is stored). The first step in any function or macro is the declaration of variables. You will need to declare a variable to hold the value of each of the constants that are hidden in equation 13 (ie: in equations 5, 10, and 11): B_e , α_e , D_e , β_e , $\bar{\nu}_e$, $\bar{\nu}_e x_e$ and $\bar{\nu}_e y_e$. In Igor, the declaration of variables that can be used by the entire program requires two steps: declaration of local variables (available to the macro alone), and declaration of global variables (available to the entire Igor experiment). The local declaration of variables is accomplished using the keyword **variable** (see line 2 below). We will then ask Igor to prompt the user to enter the values of the molecular constants by using the Igor function **Prompt** (lines 3-7). The actual global declaration occurs after the normal variable declaration, and is accomplished using the syntax: **variable /G** (lines 8-9). Finally, the macro ends with the command **EndMacro** (line 10). With these structural ideas in mind, examine the following code. *Once you understand it*, you may type it into your Procedure window:

```
Macro NewConstants (iBe, iAlphaE, iDe, iBetaE, iNuBarE, iNuBarExE, iNuBarEyE)
  variable iBe, iAlphaE, iDe, iBetaE, iNuBarE, iNuBarExE, iNuBarEyE
  Prompt iBe, "Enter Be:"
  Prompt iAlphaE, "Enter AlphaE:"
  Prompt iDe, "Enter De:"
  Prompt iBetaE "Enter BetaE:"
  Prompt iNuBarE "Enter NuBarE:"
  Prompt iNuBarExE "Enter NuBarExE:"
  Prompt iNuBarEyE "Enter NuBarEyE:"
  variable/G Be=iBe, AlphaE=iAlphaE, De=iDe, BetaE=iBetaE, NuBarE=iNuBarE
  variable/G NuBarExE=iNuBarExE, NuBarEyE=iNuBarEyE
EndMacro
```



This section of code illustrates several points of style that you should follow in writing code. First, every line except the first line, variable declaration line(s), and last line are indented by one tab stop. This is so that the body of the code (the "work engine" part) stands out. When we start to work with loop structures, the body of the loop should be indented one further tab stop. Second, the naming of variables in programming code has become standardized over the years: when words are strung together to make a variable name, the initial letters of each word are capitalized and all other letters are lower case. This is to avoid variable names such as "inubarexe" or "INUBAREXE", which are almost impossible to read.

It is a good idea to periodically check your code for errors as you program. In Igor Pro, you can do this by clicking on the Command window; Igor will then automatically compile everything in the Procedure window. If your code has errors, Igor will give you error messages with some (hopefully) helpful information on how to fix them. If you see a "compile" button at the bottom of your Procedure window, Igor has not yet compiled the code in the window so it will not yet run (you can also click on the compile button to compile the code).

Click on the Command window now to check your code. If you do not receive an error message (yay!), look under the 'Macro' menu for your new macro and try to run it. Enter the values

for H^{35}Cl obtained from Huber & Herzberg. All constants should be in units of cm^{-1} , and numbers like 5.3194×10^{-4} can be entered as 5.3194e-4. These values should also be entered in your notebook, with the reference.

Your global variables are now loaded and are ready to be used in functions! To test this, go back to the Command window and type: **Print "Be=", Be**. Igor should respond in the History area with **Be = 10.5934**.

➡ Are you writing in your notebook? Have you saved your experiment yet ??????

Now that the global variables are loaded, it's time to start working on the calculation of the energy levels. We will create here a suite of functions that will allow us to calculate the energy of any quantum state (v,J) . Our suite will consist of the following functions:

```
Function Bv(v)           // Calculates  $B_v$  for vibrational level  $v$  using  $B_e$  and  $\alpha_e$ 
Function Dv(v)           // Calculates  $D_v$  for vibrational level  $v$  using  $D_e$  and  $\beta_e$ 
Function Nubar(v)        // Calculates  $\bar{\nu}_v$  for vibrational level  $v$  using  $\bar{\nu}_e$ ,  $\bar{\nu}_e \times \chi_e$  and  $\bar{\nu}_e y_e$ 
Function RotEnergy(v,J)  // Calculates the rotational energy for state  $(v,J)$ 
Function VibEnergy(v)    // Calculates the vibrational energy for state  $(v)$ 
Function RovibEnergy(v,J) // Calculates the total rovib energy for state  $(v,J)$ 
```

Why are we breaking the calculation up like this? One of the cardinal rules in programming is to compartmentalize your code. That is, break big jobs into small sections. This makes debugging easier, interpreting someone else's code easier, and additions or modifications easier. In this case, we are breaking the calculation of equation (13) into five smaller pieces. We'll tackle one piece at a time.

Functions in Igor are defined with the keyword **Function** (instead of **Macro**). Functions and macros are similar, but generally functions run faster, while macros are more versatile. We will use functions when we need a value *returned* to some other function or macro so that we can use it in another calculation. You will be lead through the writing of code for the first function, but you must write the others on your own.

The first function is **Bv(v)**, which will calculate a value for B_v using equation (10). The name of the function is **Bv**, while the **v** in parentheses indicates that the function will need to be given a value of v (the vibrational quantum number) in order to do the calculation (it is an "input variable"). All of the molecular constants in equation (10) have already been declared as global variables and can be used by all functions all of the time. The typical syntax for a Function is shown in the sample code below. The first line defines the function and the variables that the function needs in order to do the calculation. The second line is the variable declaration, which lets the program know that you will be using **v** as a variable in the calculation. The meat of the function is in the return statement, which simultaneously defines the calculation to be performed and tells the function to return the answer to you.

```
Function Bv(v)           // Function calculates  $B_v$  in  $\text{cm}^{-1}$  from global molecular constants
variable v
    return( Be-AlphaE*(v+0.5) )
End
```

The double slash (//) is a symbol used by Igor to denote comments (text ignored by Igor). As you program, you *must* insert comments so that it is easier for other people to understand your code.

When you feel that you understand how this code works, enter it in the Procedure window below the Macro code. Test your function by typing in the Command window: **Print "Bv(0)=", Bv(0)**. Igor should respond with: **Bv(0) = 10.4398**. Perform this calculation in your notebook using equation (10) to be sure it's right before you go on.

Now you need to write the code for the other four functions in our suite. Use carriage returns between the end of one function (or macro) and the beginning of the next so that they can be read more easily. In creating **RotEnergy**, **VibEnergy** and **RovibEnergy**, take advantage of the other functions that you have written by using them in your code. For example, the body of the code for **RovibEnergy** should look like:

```
return( VibEnergy(v) + RotEnergy(v,J) )
```

Be sure to pay attention to what variables need to be declared (for this set of functions, it will be some combination of *v* and *J*, as defined in the function declaration – in general it is whatever variables are used by the function).

⇒ If you haven't saved your file yet, do so. This is your last reminder!

When you are finished with all 5 functions, you are ready to test them by graphing the energy levels of the ground ($\nu=0$) and excited ($\nu=1$) vibrational states of HCl as a function of rotational quantum number, *J*, for $0 \leq J \leq 50$. To do this, you will need to (a) make waves that will hold the values of all 51 of the ground and excited state vibrational energies, and (b) assign the appropriate energy values to the waves. You have already learned how to make waves in the Igor tutorial. For the *x*-axis (*J* values), we will take advantage of the wave **x**, which already exists in Igor's memory and contains the integer values 0,1,2,3... such that **x[0]=0**, **x[1]=1**, etc.. To see how this wave is useful in this case, click on the command window and type the following text:

```
make /n=51 Vib0, Jwave
Jwave=x
Vib0 = RovibEnergy(0,Jwave)
AppendToTable Vib0, Jwave
```

Do you understand what you just did?! The first line creates a wave named **Vib0** to hold the values of the ground state vibrational energies and a wave named **Jwave** to hold the values of *J*. The second line tells Igor to set the waves **x** and **Jwave** to be equal (so now **Jwave[0]=0**, **Jwave[1]=1**, etc.). The third line tells Igor to calculate many values of the function **RovibEnergy(v,J)** using $\nu=0$ and all of the values in the wave **Jwave** for the values of *J*, then place the result of the calculation in **Vib0** (so now **Vib0[0]= RovibEnergy(0,0)**, **Vib0[1]= RovibEnergy(0,1)**, etc.). The last line tells Igor to display the contents of the waves **Vib0** and **Jwave** in a table. Click on Table0 -. you should see your new wave, **Vib0**, and the corresponding values for each value of *J* at $\nu=0$.

⇒ Calculate two values of Vib0 in your notebook to check that the function RoVibEnergy(v,J) is working properly, and that values have been assigned to Vib0 correctly.

To plot **Vib0** versus *J*, enter the code:

Display Vib0 vs Jwave

Since the energies are quantized, these values should not be displayed as a continuous line. Modify the symbols to show discrete values and add axes labels.

Now you must repeat the procedure for the rovib energies when $v=1$, however, you will not need to recreate **Jwave** (it already exists) – you can just use it as needed. Append the Vib1 values to your plot by typing:

AppendToGraph Vib1 vs Jwave

Your graph should look similar to the one in Figure 1. Be sure that both axes start at 0.

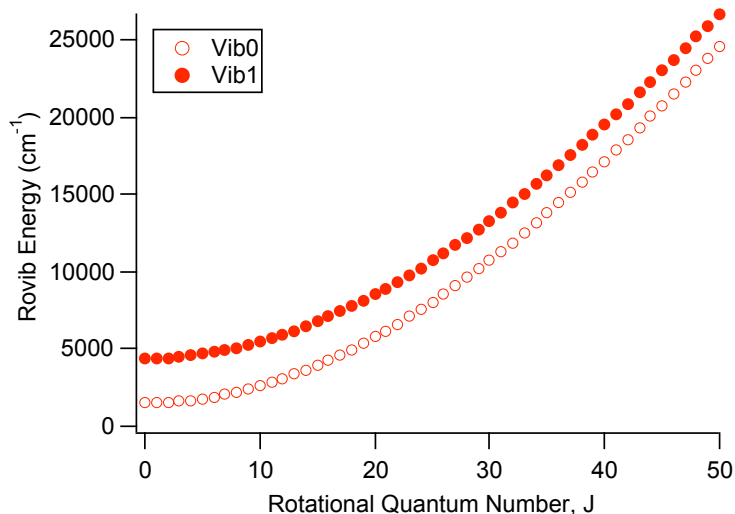


Figure 1: The rovibrational energies of H^{35}Cl as a function of rotational quantum number.

⇒ Print a copy of your graph to hand in with your report (be sure that it has a Figure caption or a title!).

Since the wave x contains integer values, we could actually use x whenever we need quantum numbers, rather than create a new wave to hold quantum numbers. You may use either approach, but be sure to document your choice in your notebook.

Positions of Absorption Peaks in the HCl Rovib Spectrum

When a molecule absorbs light, the energy from the light causes the molecule to undergo a transition from an initial state to a final state. According to the Bohr condition, the molecule can only absorb the light if the energy of the light matches the difference in energies between the two states of the molecule ($E_{\text{photon}} = \Delta E_{\text{matter}}$). However, not every one of these photons will be absorbed equally well – that is, we may not be able to observe a measurable absorbance intensity for each type of photon that matches the Bohr condition. This intensity condition is embodied in part in "selection rules", which are a set of restrictions on the allowed changes in the values of the quantum numbers. They are a result of integrals that are required to calculate the intensities of the transitions,

and ultimately arise from fundamental rules such as conservation of angular momentum. The selection rules for rovib absorption transitions of a diatomic³ are:

$$\Delta v = +1, \Delta J = \pm 1 \quad (14)$$

This means that the only rovib transitions that can be observed for a diatomic are those that correspond to changes in energy between states with the quantum numbers described in equation (14). Therefore, according to the selection rules, the allowed rovib *absorption* ($\Delta v = +1$) transition energies for a diatomic are given by:

$$E_{\text{photon}} = E_{\text{rovib}}(v+1, J \pm 1) - E_{\text{rovib}}(v, J) \quad (15)$$

Transitions for which $\Delta J = -1$ are called P-Branch transitions (“energy-**P**oor”), and transitions for which $\Delta J = +1$ are called R-Branch transitions (“energy-**R**ich”). In a spectrum, the x-axis represents transition energy (frequency, wavelength, etc.). Therefore, the terms transition energy and peak position (i.e.: position along the x-axis) are used interchangeably. P-Branch transitions occur at lower transition energies than R-Branch transitions, which is presumably the source of their names. Spectroscopists use the shorthand $P(J)$ and $R(J)$ to signify these transition energies.

$$P(J) = E_{\text{photon,P}} = E_{\text{rovib}}(v+1, J-1) - E_{\text{rovib}}(v, J) \quad (16a)$$

$$R(J) = E_{\text{photon,R}} = E_{\text{rovib}}(v+1, J+1) - E_{\text{rovib}}(v, J) \quad (16b)$$

It is important to remember, however, that $P(J)$ and $R(J)$ actually correspond to values of E_{photon} and thus to *differences* in energies of matter, E_{rovib} . We could write out expression for $P(J)$ and $R(J)$ in terms of molecular constants using equations (4) and (9), but since we already have a function that calculates values of E_{rovib} for any value of v and J , that would defeat the purpose of compartmentalizing code! Instead, we will have Igor calculate values of $P(J)$ and $R(J)$ using equations (16a,b) directly.

TASK 2: Calculation of Peak Positions (Transition Energies)

- Create function `PeakPosition` to calculate transition energies, and check code

To calculate the positions of the absorbance peaks along the x-axis, we will create a function called **PeakPosition**. There are two approaches that we could take to do this. The easiest (but more inelegant) way is to define two separate functions, one to calculate the energies of the P branch transitions and one to calculate the energies of the R branch transitions. Each of these functions would require the input variables v and J (see equations 16a,b).

A more elegant (and not much harder) approach is to define a single function that has one additional input variable, the branch name. This approach requires the use of a structure called **conditional branching**, which is used when you want the function to do one thing in one case, but something different in another case. It is a structure that is common to almost all programming languages. The conditional branch in Igor appears as an **if-else-endif** structure. Our function will

³ These selection rules are actually only strictly true for a HO/RR system. In practice, they work quite well for predicting the rovib spectrum of most diatomics at a reasonable resolution.

therefore actually require three input variables: **Branch**, **v**, and **J**. The variable **Branch** is a **string** variable (i.e.: a variable that holds a string of text) that allows us pass string values of either “P” or “R” to our function as the input (string values are indicated in Igor by quotation marks). The frame of our function looks like this:

```
Function PeakPosition(Branch, v, J)           // Calculates P or R transition energies
string Branch                               // Pass values of “P” or “R” only!
variable v, J                               // v is the initial state vib quantum number

    if ({some expression})                 // if this is a P-branch transition ...
        {do this if the expression is true} // then calculate the P-Branch peak position
    else                                    // if this is an R branch transition ....
        {do this if the expression is false} // then calculate the R branch peak position
    endif

End
```

but we need to fill in the squiggly brackets with Igor code.

We will start with the *{some expression}*, which needs to be something like “Is the string variable Branch equal to P?”. Although Igor has a number of comparison operators on hand for comparing *numbers* (=, <, etc.) they do not work for *strings* (string comparisons are complicated by questions such as: does “P” equal “p” or “{space}P”, etc.). Igor has a different set of built-in functions for comparing strings. There is an on-line catalog of available functions that is kept in the ‘**Function Help...**’ menu under the ‘**Misc**’ menu. Choose ‘**String**’ functions from the pull-down menu. By clicking on a choice, you can view what the function does and how to use it. If you surf around the String functions for a few minutes, you should find the function **cmpstr**, which will work for our purposes. *Describe how it works in your notebook.*

We can now replace *if({some expression})* with:

```
if ( cmpstr(Branch, “P”) == 0 )           // if this is a P branch transition ....
```

Be sure that you understand this code before you use it.

Now you need to complete **PeakPosition** by filling in the other squiggly brackets with code to actually calculate the P and R branch transition energies. Be sure to use a **Return** statement so that you will have the result of the calculation returned when you call the function. Keep your code well commented!

When you are finished, test your code by entering the following **Print** statement in the command line (the response is also shown below the statement).

```
Print PeakPosition(“P” , 0 , 10), PeakPosition(“R” , 0 , 10)
2651.68      3072.18
```

Note that you should get exactly these numbers. Even small differences mean that you have an error somewhere, either in your molecular constants or in the programming.

↪ Calculate the values of $E_{\text{photon}} = \Delta E_{\text{tot}}$ for the P-Branch ($v=0, J=10 \rightarrow v=1, J=9$) and R-Branch ($v=0, J=10 \rightarrow v=1, J=11$) transitions in your notebook using your values for the molecular constants. Does your code work correctly?

Be sure that you understand how all of these pieces of code work before you go on! Also, be sure that you are writing in your notebook as you go. You may print out your procedure window and paste it into your notebook in the appropriate places if you wish.

TASK 3: Generating a Stick Spectrum

- Make wave `Spectrum0` to hold values of transition intensities
- Create and modify function `FillSpectrum` to assign values of transition intensities to `Spectrum0` at correct transition energies
- Plot `FillSpectrum` using intensities of 1 for all transition energies
- Investigate the effects of different vibrational and rotational levels on the look of the spectrum

Now that you have created a function to calculate peak positions, you can start to generate a stick spectrum. A stick spectrum is a plot of the transition energies (peak positions) using sticks (lines) of unit height (intensity).

We will take a rather inelegant approach to creating a spectrum by creating a wave called `Spectrum0` that actually holds the values of the *intensities* of the absorbance peaks (i.e.: the y-axis in a spectrum) rather than the peak positions (the x-axis). The **index** of `Spectrum0` will then represent peak positions in integral numbers of cm^{-1} . For example, if `Spectrum0` is indexed from 0 to 4000, our x-axis will be allowed to have integer values ranging from 0 to 4000. Consider for example P(1) which occurs for HCl at approximately 2762 cm^{-1} . With our approach we can simply write `Spectrum0[2762]=1` to make a peak of unit intensity occur at 2762 cm^{-1} . This turns out to be inelegant because we are limited to 1 cm^{-1} resolution for absorption peaks. However, the ease in programming will offset this inconvenience for our purposes⁴.

Make a wave named `Spectrum0` that contains 4001 points (we will choose a spectral range of 0 cm^{-1} to 4000 cm^{-1} since this covers the spectral range for the IR spectrum of HCl - it would be different for a different molecule). You now have a wave that will hold the intensities of all the lines in the spectrum, that spans the correct spectral range, and that is indexed by the resident wave **x**. The initial plan is to set all of the spectral line intensities to be equal to one (we will add correct intensities later). Realize that only those points in `Spectrum0` that correspond to *observable* transitions should have a non-zero intensity; all other points in `Spectrum0` will have zero intensity.

What we need to do now is (1) use the function `PeakPosition` to calculate the transition energies of the observable absorbance peaks, and (2) assign an intensity of 1 to each observed transition in `Spectrum0`. This calls for another Function⁵, which we will name `FillSpectrum`.

To do this, we need to learn another programming structure. A **loop** is useful whenever you want to have the same task repeated. In Igor, the supported loop structure is a **do-while** loop (other programming languages may have slightly different varieties of loop structures). The general form of the do-while loop is:

⁴ A more elegant, and seemingly logical, approach would be to create two waves, one for peak position, and one for intensity. In practice, it turns out to be rather complicated to program a versatile way correlate these two waves.

⁵ We could have used a Macro, but we need the speed of a Function.

```
do
  {things to be repeated}
  while( {expression} )           // if expression is true, go through the loop again
```

In this case, you want the value of **PeakPosition** to be calculated for two branches (P and R) and for many values of J (say, $J=0$ to $J=50$). You also need to be able to assign a value of 1 to the intensity of each observed transition. Therefore, the skeleton of the code for our Function looks like this:

```
Function FillSpectrum(v)
variable v           // value of v for the initial rovib state
variable J           // value of J for the initial rovib state
  J=0                // start at J = 0
  do
    { calculate the P and R absorbance peak positions corresponding to this J }
    { set Spectrum0 equal to one at those peak positions }
    J+=1             // increase J by 1
    while (J<=50)    // check to see if J is still less than or equal to 50, if so loop again
  End
```

This loop starts by calculating the P and R transition energies for an initial state with the specified value of ν that was passed to the function and $J=0$, then sets the intensities of those transitions to 1. It then increments the value of J to $J=0+1=1$. Since J is still less than 50, the loop starts again with $J=1$. The calculation and intensity assignment will be done again and again for each incremented J until $J+1=51$, at which point the loop breaks. The variables **v** and **J** are declared on two separate lines in this case because **v** is a value that is passed to the function by the outside (user), while **J** is used only internally by the function. (Note that the maximum allowed value of J here is not physically meaningful - it's just some large number.)

Now we need to fill in the body of the loop with Igor code. You already have written a function (**PeakPosition**) that does the first task in the body of the loop. Therefore, we can devote our efforts to the second task of assigning an intensity of one to each of these lines. It would be nice to just say **Spectrum0[PeakPosition] = 1**. However, this will not actually work because the energies calculated by **PeakPosition** are non-integral (for example, $P(1)=2762.16 \text{ cm}^{-1}$), but the indices of **Spectrum0** are integral (they are the values in **x**: 0, 1, 2, 3, ..., 2762, 2763, ... etc.). Fortunately, Igor has a function that takes any number (i.e.: 2762.16) and finds the point number (i.e. index number) on a wave that is closest to that value. This function is called **x2pnt**. **Look up x2pnt in the Igor function list, and describe how it works in your notebook.** To see **x2pnt** in action, type in the following code in the **Command** window:

```
Print x2pnt(Spectrum0, 2762.16)
```

Igor should return the value **2762**, since point 2762 is the closest point to the number 2762.16. Rather than having actual numbers flying around in the code (very clumsy), we will use a new variable called **PeakPoint** to hold the value of **x2pnt(Spectrum0, 2762.16)**, i.e.: to hold the integer

value of the peak position. Now we can accomplish the task of assigning an intensity of one to the spectral line near point 2762 with the statements:

```
PeakPoint = x2pnt(Spectrum0, 2762.16)
Spectrum0[ PeakPoint ] = 1
```

Recall that square brackets are used for indexing a wave, so the second statement reduces to `Spectrum0[2762] = 1` in this example. Of course, you don't want the number 2762.16 flying around either. Modify this code to replace the value 2762.16 with the code that you use to calculate that peak position.

Once you have gotten this far, it is tempting to just replace *{ set Spectrum0 equal to one at that line position }* with `Spectrum0[PeakPoint] = 1` and be done with it. However, it is prudent to put in a line for error checking. Suppose, for example, that we ask Igor to find a point in `Spectrum0` that is out of `Spectrum0`'s range (0 to 4000 cm^{-1}). This might happen if you switch molecules, or make a mistake with the input of molecular parameters. Clearly there would be a problem. In some cases you might just get an error message, but in others you might actually crash the program, or even the machine. It is good programming technique to try to avoid these sorts of annoyances. In this case, you will need to make sure that the value of `x2pnt(Spectrum0, {transition energy})` is greater than the lower bound of `Spectrum0` (which is 0), and less than the total number of points in `Spectrum0` (the upper bound). The function `numpnts` can be used to determine the total number of points in `Spectrum0`. (Look up `numpnts` in the Igor 'Waves' function list.) The logical operator `%&` acts like an AND (it is called a bitwise AND) can be used to compare values of `PeakPoint`.

You can now bring all of this information together to construct the body of your do-while loop. Below is an example of the code you should end up with in place of the old body of the loop:

```
do
  PeakPoint = x2pnt(Spectrum0, PeakPosition("P",v,J)) // find point closest to transition
  if ((PeakPoint >= 0) %& (PeakPoint < numpnts(Spectrum0))) // peak is within range ?
    Spectrum0[PeakPoint] = 1
  endif
  J+=1
while (J<=50)
```

This code will fill in the P-branch part of the spectrum. **Be sure that you understand how it works! Ask questions if you don't understand!!**

To complete `FillSpectrum`, you need to do the following things:

- Declare the new variable `PeakPoint` in the second variable declaration at the beginning of the function. (Note again that variables that appear in the "call statement" (in the definition of the function: i.e.: *v*) must be declared on a separate line from variables that are used only inside of the function (i.e.: *J* and `PeakPoint`)).

- At the beginning of the body of the function, set the value of all points in **Spectrum0** to 0 (in case this is not the first time through the calculation). You can do this quickly by simply assigning the entire **Spectrum0** wave a value of zero.
- Add some code within the loop to calculate the R-branch peak positions and set their intensities to equal one.

Now you can test your new function! From the Command line, call **FillSpectrum** with $v=0$, then plot **Spectrum0** (type **Display Spectrum0** in the command window – Igor will choose the x-axis automatically). Adjust the x-axis display to give you a nice picture of the spectrum, change the markers to “sticks to zero”, and label the axes. Print out a copy of this spectrum *now* because it will change later. Mine looks like this:

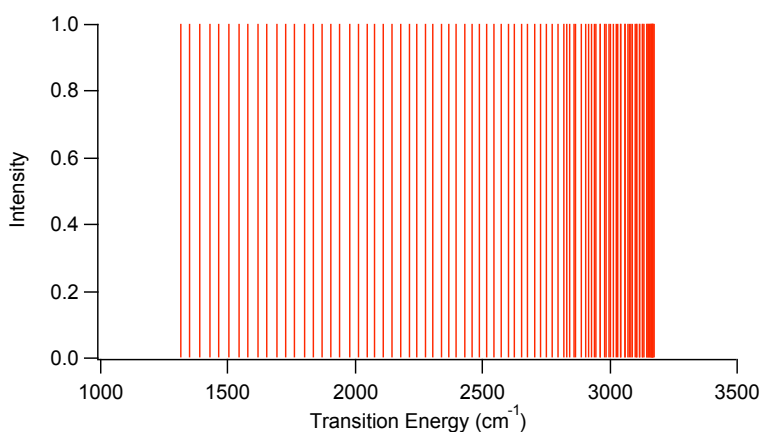


Figure 3: A crude H^{35}Cl stick spectrum

Once your function works, tweak it as follows:

- Add the ability to change the maximum value of J used in the calculation in the do loop. To do this, add **Jmax** to the list of parameters passed to the function (in the parentheses, next to v), include **Jmax** as a variable in the first variable declaration, and change the **while ($J \leq 50$)** statement accordingly.
- Add the ability to change the intensity wave **Spectrum0**. This is so we can simulate multiple spectra, say for transitions starting at $v=0$, or at $v=1$, etc. We will use a variable named **Spectrum** to hold the name of the spectrum wave (for example, **Spectrum0**). Change all references within the code from **Spectrum0** to **Spectrum**. Add **Spectrum** to the list of parameters passed to the function, and declare it as a wave on a separate line between the two other sets of variable declarations.
- Make sure that the line position for $P(0)$ is *not* marked (a $J=0 \rightarrow J=-1$ transition is *not possible* since J is never negative!). While this really should be accomplished within **PeakPosition**, it is acceptable to assign $P(0)$ an intensity of zero in **FillSpectrum**. Be sure to do this *outside* the body of the loop.

When you have gotten everything to work, you should experiment with your function. Set your x-axis so that it spans the full range of transition energies (0-4000 cm^{-1}). Try running **FillSpectrum** with various different values of **Jmax**, **v**, and **Spectrum**. To change the value of **Spectrum**, you will need to make new **Spectrumv** waves first (i.e.: match the value of **v** to the name of the spectrum wave so that when **v=0**, **Spectrum=Spectrum0**, etc.). You should get to the point where you are able to answer/discuss the following questions:

- Notice that there is a maximum energy beyond which there are no transitions. Why would this be?
- What happens to the spectrum as **Jmax** is increased? Is it *physically* meaningful to change Jmax? Is there some value of Jmax that would be physically meaningful? How would you calculate it?
- What does it mean when there is a transition line at 0 cm^{-1} ?
- What happens to the spectrum as you change v? Try displaying several spectra (say, **Spectrum0** and **Spectrum1**) on the same plot (use different colors). What happens when v is very large? Why?

You should have a good sense of all of the information contained in these plots **before you go on**.

↪ Print out several representative versions of your stick spectra from your investigations (you may have to adjust your x-axis). You need to do this **now** because the waves will get modified later and your spectra will look different.

Take a deep breath – this section was the longest one!

Intensities of the Rovib Transition Lines

The reason why your stick spectrum looks nothing like an experimental spectrum is that the intensities are wrong. The intensity of a transition is directly proportional to the number of molecules in the *initial* state (except in the case of very intense light sources). Therefore, to get the correct the line intensities, we must be able to determine the population of each (*v*,*J*) state at a given temperature T. According to statistical mechanics, this is given by the Boltzmann equation:

$$N_{v,J} = g e^{-\Delta E(v,J)/kT} \left(\frac{N}{\sum_{v,J} e^{-\Delta E(v,J)/kT}} \right) \quad (17)$$

where N is the total number of molecules, $\Delta E(v, J)$ is the **excess** rovib energy of the state *v*,*J* **above** the zero point energy (i.e.: the energy needed to access the (*v*, *J*) state from the (0,0) state), g is the degeneracy of the state, and *k* is the Boltzmann constant. Simply stated, the Boltzmann equation says that unless the energy required to populate a state is less than or equal to the thermal energy ($=kT$), the state has little chance of being populated. If the state is not populated, the intensity of the transition originating from that state will be zero. The zero point energy is not included because a molecule can never have an energy less than the zero point energy.

For the purposes of this simulation, only the *relative* intensities of spectral lines are important, not absolute intensities. Therefore, only the *relative* populations of different states are really

important. The term in parentheses in equation (17) is just a constant at a given temperature and for a given number of molecules. Therefore, if we consider the populations of all states relative to the (0,0) state, then we can write:

$$I = \frac{N_{v,J}}{N_{0,0}} = \frac{g_{v,J} e^{-E(v,J)/kT}}{g_{0,0} e^{-E(0,0)/kT}} \quad (18)$$

The degeneracy (g) of any rovib state is equal to $(2J+1)$.⁶ Furthermore, the *excess* energy above the zero point energy is $\Delta E(v,J) = E_{\text{rovib}}(v,J) - E_{\text{rovib}}(0,0)$. Therefore, the relative intensity of a transition arising from any state (v,J) is:

$$I(v,J) = (2J+1) e^{-(E(v,J)-E(0,0))/kT} \quad (19)$$

↪ Derive equation (19) from equation (18) in your notebook.

TASK 4: Creating a Thermal Envelope

- Create function `RelativeIntensity` to calculate values of relative transition intensities
- Plot `RelativeIntensity` vs. J for $v=0, 1$
- Modify function `FillSpectrum` to assign relative values of transition intensities to `Spectrum` at correct transition energies
- Investigate the effects of different vibrational and rotational levels and temperature on the look of the spectrum

Using equation (19), write a function called **RelativeIntensity** which has three input variables: v , J , and T . Since we are working in units of cm^{-1} for energy, you should use $k=0.69502 \text{ cm}^{-1}/\text{K}$ as the value of the Boltzmann constant. Note that we really should have included k in our original **NewConstants** macro. You can do this now if you wish. You will need to re-run the macro once you have changed it.

↪ Show the conversion of k from units of J/K to cm^{-1}/K in your notebook.

Test your function by typing: `Print "RelativeIntensity(0,1,298)=", RelativeIntensity(0,1,298)`. You should get a value of **2.71234**. Do this calculation in your notebook by hand to be sure that it's right.

Plot the relative intensities of the transitions originating from the ground ($v=0$) and first excited ($v=1$) vibrational states of H^{35}Cl at room temperature as a function of J , for $J=0$ to 50. You will first need to make the waves **Intensity0** and **Intensity1** to hold the values of these intensities. You may want to go back and see how you did this for the Rovib energies earlier. Your plot should resemble that in Figure 4.

⁶ This is because J is not the only quantum number pertaining to rotation - there is also the quantum number m , which describes the plane of orientation. Since m spans the range $-J$ to $+J$, there are $2J+1$ sublevels for each J level. In the absence of external fields, all of the m sublevels are degenerate (they have the same energy), and so many m states can be populated with equal probability.

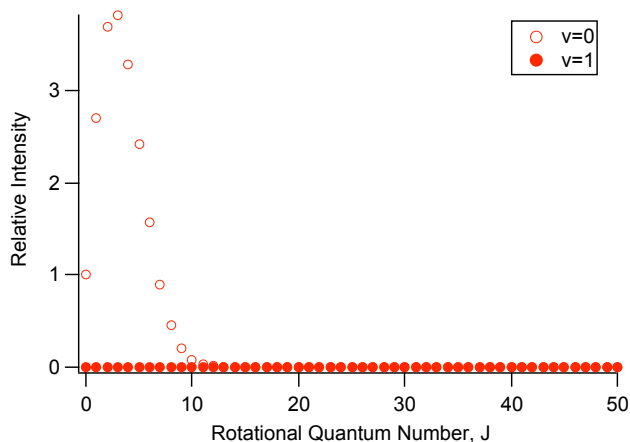


Figure 4: Relative populations of H^{35}Cl at room temperature

Appearances are deceiving: the shape of the $\nu=1$ level distribution (“thermal envelope”) is actually the same shape as the $\nu=0$ level distribution, but the relative intensities are much smaller. Verify the similarity of the $\nu=0$ and $\nu=1$ thermal envelopes with a graph generated as follows:

```
display Intensity0           // The wave for v=0
append/R Intensity1         // Append the wave for v=1 using the right hand axis
```

⇒ Print out both versions of the thermal distribution plot.

Now return to the `FillSpectrum` function and modify it to display the true intensities of each line. You will need to add the input variable `T` (ambient temperature) to parameter list, and declare `T` as a variable in the first set of declaration statements. Modify the lines that assign an intensity of 1 to each transition line to now assign an intensity calculated by your function `RelativeIntensity`.

Run `FillSpectrum` at room temperature (298 K) for `v=0` and `Jmax=50`. Expand the center region. You will probably see something that looks like Figure 5, below.

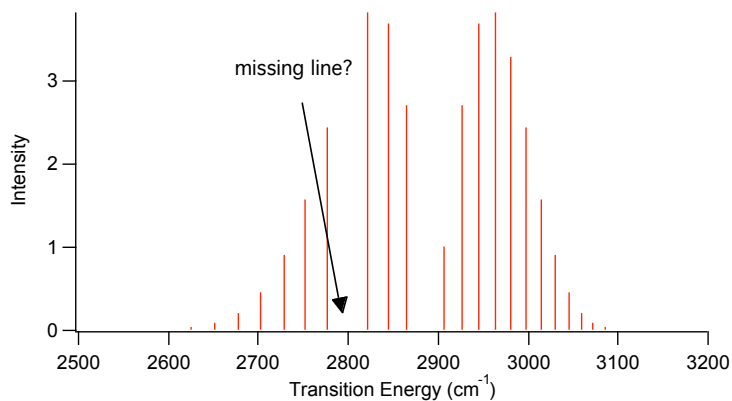


Figure 5: A slightly flawed spectrum of H^{35}Cl at room temperature.

If you compare this plot to the experimental H^{35}Cl spectrum, you will see that you are missing P(4)! The reason for this is fairly subtle. It happens that R(50) ($=2799.2\text{ cm}^{-1}$) falls within 1 cm^{-1} of P(4) ($=2798.97\text{ cm}^{-1}$), so that both transitions are assigned to the same location in **Spectrum0**. Since R(50) is calculated *after* P(4) in the do-while loop, the value of P(4) is *overwritten* by the value of R(50) in **Spectrum0**. The intensity of R(50) at 298 K is so tiny that we end up with a missing line!

A more realistic way to handle this is to **sum** overlapping lines rather than overwrite them. Go back and modify **FillSpectrum** so that overlapping lines are summed instead of overwritten (use the += assignment rather than the = assignment). Now re-run your function. The P(4) line should be there.

Experiment again with **FillSpectrum** using different values of ν , **Jmax**, and **T**. Choose a **Jmax** that allows you to see significant effects on the entire spectrum up to a temperature of 8000 K. Go back and look at the questions on page 84. Can you answer these questions better now?

➡ Print out several representative versions of your spectra at different temperatures, ν values and **Jmax** values, using illustrative x-axis ranges. You do not need to print out 50 plots, just a few representative versions!

TASK 6: Making the Movie

- Create macro RunSimulation to create a movie of the evolution of the spectrum with increasing **T**
- Create movies for $\nu = 0$, $\nu > 0$, expanded region of spectrum

Now you can create your movie showing the evolution of the HCl spectrum as the temperature is raised. We will use a temperature range of 1 K to 8000 K. To prepare,

- close all graphs except for the one showing **Spectrum0** (Igor can only make a movie of the top-most window).
- Resize this graph until it is approximately 7" wide by 4" tall.
- Autoscale and label the axes.
- In the upper left-hand corner of the graph, add annotation which says "T=1 K".

Now you need to write a macro called **RunSimulation**. In its first incarnation, you want it to repeatedly call **FillSpectrum** with temperatures increasing from 1 K to 8000 K, multiplying by a factor of 1.1 on each pass. Use your best value of **Jmax** (I suggest a value larger than **Jmax**=50 since you will be simulating high temperatures) and $\nu=0$ (and thus also **Spectrum0**) for a first try. You should notice by now that this calls for a loop structure.

Within your loop structure, you will also need to update the annotation on your graph after calculating the spectrum at the new temperature. The following code will do this for you:

```
NoteString = "T =" + num2str(T) + "K"           // T is the temp. variable
Textbox/C/N=text0 NoteString
```

As usual, you should not use this code until you understand how it works (the second line is Igor's way of making a textbox. You need not worry about that part). Be sure to declare **NoteString** as a string variable at the beginning of the macro.

Once you have gotten your simulation running, it is time to make the movie! In the macro **RunSimulation**, add the command **NewMovie** (on its own line) right after your variable declarations, then add **CloseMovie** right before **End**. Within the temperature loop, add the command **AddMovieFrame** *after* the annotation update code. These commands will slow the macro considerably, but it still runs nicely as a QuickTime movie. When you run your macro, Igor will create the movie. It can be viewed by exiting Igor and double-clicking on the Movie icon that you just created (the QuickTime viewer called 'MoviePlayer' is already installed on the Macs).

Make at least two more movies: one with $\nu > 0$, and one with the x-axis expanded to cover the region 2800-3500 cm^{-1} . What do you learn from these two new movies?

Band Head Formation

When you watch your movie, you should see a band head form as the temperature is increased. A band head forms when one branch of the spectrum "folds back" upon itself, so that there is a maximum (or minimum) transition energy (a band head). The value of J at which the band folds can be calculated by treating the expression for the transition energy as a continuous function of J . Ignoring D terms, we have:

$$P(J) = \bar{\nu}_{BO} - (B_1 + B_0)J + (B_1 - B_0)J^2 \quad (20)$$

$$R(J) = \bar{\nu}_{BO} + 2B_1 + (3B_1 - B_0)J + (B_1 - B_0)J^2 \quad (21)$$

where $\bar{\nu}_{BO}$ is the band origin energy (due to vibrational energy changes only). To find the value of J at which $P(J)$ reaches a maximum, we can differentiate equation (20) with respect to J and set the resulting expression equal to 0. Solving for J , we find that

$$J_P^* = \frac{B_1 + B_0}{2(B_1 - B_0)} \quad (22)$$

where J_P^* is the rotational quantum number at which the P-branch turns around. Similarly, we find that

$$J_R^* = \frac{-(3B_1 - B_0)}{2(B_1 - B_0)} \quad (23)$$

⇒ Derive equations (22) and (23) in your notebook, then calculate values of J_P^* and J_R^* for the $\nu=0 \rightarrow 1$ and $\nu=1 \rightarrow 2$ spectra.

If the P-branch does not fold back, then the value of J_P^* is negative (it does not fold). This is the case for the ro-vib spectrum of HCl.

Go back to the questions on page 84 and rethink your answers. You should feel very comfortable with all of these types of questions by the time you have finished your simulation. If

not, go back and play some more with different values of v , J_{\max} , and T for your different functions until you think you have a good physical understanding of rovibrational spectroscopy.

Wrapping it Up

In addition to your notebook pages, your report should include the following:

- an abstract
- the code that *you have written* in the Procedure window (that is, you do not need to print out Graph or Table macros that *Igor* created)
- plots of the various figures that you made during the simulation
- a **labeled** diskette containing your Igor experiment, *free of viruses!!!!*
- a short discussion of some of the relevant points raised on page 84.

Be sure that your notebook record contains sufficient detail so that someone could reconstruct your simulation **without the aid of this manual**. You may assume that the “someone” has a working knowledge of Igor that is similar to your understanding before you started this experiment. You must include explanations of how the Igor functions that we used (**x2pnt**, etc.) work. Don’t forget the derivations also.

General References

Spectroscopy:

J. I. Steinfeld, *Molecules and Radiation: An Introduction to Modern Molecular Spectroscopy*, 2nd Ed., MIT Press, Cambridge, MA (1993) (On reserve in Murphy Library)

K. P. Huber and G. Herzberg, *Molecular Spectra and Molecular Structure: IV. Constants of Diatomic Molecules*, Van Nostrand, New York (1979) (An updated version of the tables in Herzberg, pages for H³⁵Cl are included in this manual as Appendix B.)

P. W. Atkins, *Quanta*, 2nd Ed., Oxford University Press, Oxford (1991) (see *Angular momentum, Branch, Vibrational spectroscopy*.)

Programming:

Igor Pro User's Manual