Exercise 5

Molecular Statistics, Week 5

2014

1 Introduction

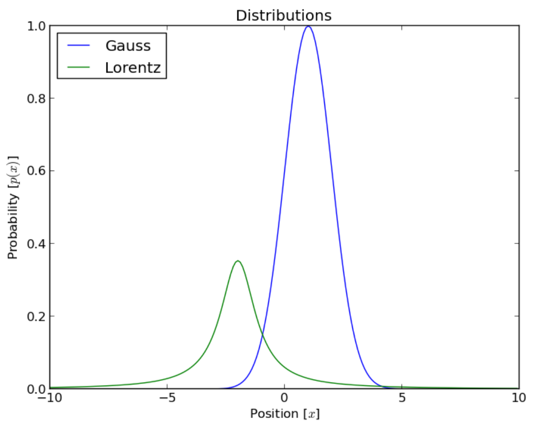
Often it will be necessary to data-mine, manipulate and visualize data obtained manually from experiments or from other software. Python is great for this and the goals of this exercise is:

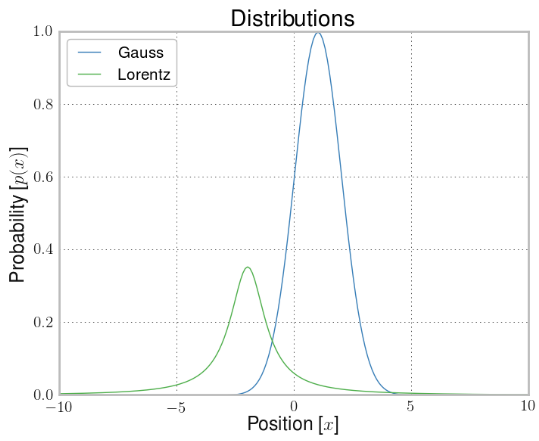
1. Use Python to load/read data
2. Use Numpy to manipulate data
3. Use matplotlib to illustrate data
4. Use string manipulation for tables

Note to remember. If you sit with some data and you do not know how to plot the specific plot, then take a look at matplotlib.org/1.3.1/gallery.html for inspiration.

1.1 Changing the look of matplotlib

You can change pretty much anything in a matplotlib plot using the `rc` function. A small header file to override most of the default stuff in matplotlib to make it look more professional can be found here github.com/charnley/matplotlib-header. Here is a before / after example.





2 Exercises

Todays exercises will each be based on different sets of data which requires different plot representation. You will be given different data files to load/read in Python. Try to use Google to answer questions you might have, as not all details you need are included in the exercises.

2.1 Energy of a covalent bond during stretching

This data represents what will happen to the overall energy if you move a hydrogen, in a water-dimer, away from it's global minimum. This is calculated with 4 different methods and stored in corresponding data files. The data is structured like this:

```
    -0.30    0.000000000    1.094813771    -0.789196930    -152.5782121321
```

The first column is the displacement of the hydrogen in Ångstrom. The next three are the dipole-moment (which we are not using) and the last is the total energy in Hartree.

Your job is to see if the four energy models, B3LYP, CCSD(t), CCSD(t)-F12 and MP2 describes a similar energy surface or not.

Exercise Hint: If you have no idea where to start use the following code as inspiration:

```python
f = open('method.dat', 'r')
for line in f:
    print line
```

Also look up what the command `continue` does in a for-loop and what the method `.split()` does with a string.

1. Load all the energies into a Numpy array/list and corresponding array/list for displacement, for each method. Plot the data in the same plot.
2. Remember to add labels for axis and methods.
3. Convert the energy to kcal/mol. Plot the result.
4. Usually when computing quantum chemical energies, we are not interested in absolute values. Plot the relative change in energy away from the displacement = 0.0.

2.2 Binding free energies

This data represents a theoretical thermodynamic study of host-ligand binding energies. The data is from Gilson *et al.*[1] The system in question is the Cucurbit7uril (CB7) host molecule with different organic ligands inside it. Calculations of the binding energy (ΔG°) have been done based on a semi-empirical method called PM6-DH+. The result of these calculations are stored in `calculated_new.csv` and has the following structure:

```
1 -139.8 130.3 -0.0318791946309
```

The columns are respectively ID, change in enthalpy (ΔU) [kcal/mol], change in solvent energy (ΔW) [kcal/mol] and lastly change in entropy (ΔS°) [(kcal/mol)/Kelvin]. The binding energy is then calculated with the following formula:

$$\Delta G^\circ = \Delta U - T\Delta S^\circ + \Delta W + \delta \tag{2.1}$$

where δ is an empirical parameter set to -5.83 kcal/mol and T is the temperature of 298.0 Kelvin.

Similarly the `experimental.csv` contains the experimental binding energies with the following data structure:

```
1, -5.3
```

where the columns are ID and binding energy in kcal/mol respectively.

1. Load the data into corresponding arrays and calculate the theoretical binding energies using the above equation.
2. Plot the experimental vs calculated energies.
3. Add a fixed $x = y$ line to the plot. Fix the limits of the x and y axis to match each other, to better visualize how correct the calculated ΔG° are.
4. Add two fixed dotted lines representing +/- 2 kcal/mol experimental uncertainty.
5. Calculate the Pearson Correlation factor r and corresponding p value. Does the calculated energies correlate with the experimental values? **Hint:** Search for the `scipy` module for a nice Pearson function.
6. Calculate the Root-mean-square deviation (RMSD) between the experimental and calculated binding energies.

$$\text{RMSD} = \sqrt{\frac{\sum_i^N (x_{\text{exp}} - x_{\text{cal}})^2}{n}} \tag{2.2}$$

7. As you might have noticed, there are two outliers with over 4.5 kcal deviation from the experimental value. Use `plt` to highlight which of the ligands these are, in the plot. **Hint:** Use the method `plt.tekst()` as described in the matplotlib guide / internet.

[1] Hari S. Muddana and Michael K. Gilson, dx.doi.org/10.1021/ct3002738 — J. Chem. Theory Comput. 2012, 8, 2023-2033

If you are using Latex, you will want to print out data directly as a Latex table. This can be done with the method `.format()`. Some examples can be found at http://ebeab.com/2012/10/10/python-string-format/

8. Print the ligand id, experimental and calculated energy out as a Latex table

2.3 Random precision errors in assigned chemical shifts

When assigning measured chemical shifts of a protein to their respective amino-acids you will have to match the chemical shift measured by one experiment with another. Due to experimental error these values are not exactly the same, even though they should be in theory. Because of this it can be difficult to be sure that the two matched chemical shifts actually origin from the same amino-acid. To avoid making assignment errors it is thus informative to know how well the measured chemical shifts 'should' match.

The file `chemical_shift_errors.txt` obtained from the course website contains all differences (or errors) in assigned chemical shifts of all amide protons for a single protein in a single column format.

1. Load the file containing the data and store it a variable.
2. Plot a histogram of the data using Matplotlib.

When plotting a histogram you can select the number of bins to present the data with by giving the argument `bins=10`. (10 is the default value in Matplotlib). If you try changing the number of bins you can severely affect how the data 'looks', especially if your number of datapoints are relatively low. The Freedman-Diaconis Rule can be used to select the number of bins automatically. The following code takes as argument the data and returns the optimal number of bins according to the Freedman-Diaconis Rule.

```
def bins(data):
    data.sort()
    n=len(data)
    width = 2*(data[3*n/4]-data[n/4])*n**(-1./3)
    return int((data[-1]-data[0])/width)
```

3. Plot the histogram again using the Freedman-Diaconis Rule to select number of bins.

If these errors are completely random, they should approximately follow a normal distribution (also known as a Gaussian distribution). We can use the module `scipy.stats` to fit a distribution to a dataset. Import this in your program as follows:

```
import scipy.stats as ss
```

We will begin by fitting a normal distribution to our data. The command `ss.norm.fit(data)` returns the mean and standard deviation that best describes the data. To draw this curve we will need a set of x and y-values that cover our data range.

4. Use `np.arange()` together with the `max()` and `min()` functions to generate x-values that range from the lowest data point to the highest in steps of `1e3`. Store these in a variable called `x`.

The function `ss.norm.pdf(x, param_1, param_2)` returns the probability densities for a normal distribution for all values of `x`. `param_1` and `param_2` are the mean and standard deviation you obtained from the fit. Fitting more complicated distributions will return more than two parameters. To avoid having to adjust the number of arguments for each distribution you look at, the following works for every distribution in the `scipy.stats` package:

```
parameters = norm.fit(data)
y = norm.pdf(x, *parameters)
```

5. Fit a normal distribution to your data.
6. Try to plot the fitted distribution together with the histogram. *Hint!* Use `normed=1` as argument to your histogram.
7. Try fitting other popular distributions such as the Cauchy/Lorenz distribution `ss.cauchy` or the Student's t-distribution `ss.t`.

2.4 Protein structure determination

Not all protein structures can easily be determined experimentally. These kinds of proteins will often have their structure determined by simulation where a force-field is used to describe how the atoms interact with each other. If a good force-field is used, the correct (also called native) structure should correspond to the lowest energy of the force-field. When developing new force-fields you want to see how well the energy correlate with the deviation from the native structure.

The file `rmsd_energy_unfolded.txt` obtained from the course website, contains the energies, in units of $kcal/(mol \cdot RT)$ at $300K$, of several proposed structures as well as the atomic root mean square deviation (rmsd) in Å from the native structure. You will never get an rmsd of zero for structures proposed by simulation, since the force-field will never fully describe all interactions correctly, however rmsds under 3-5 Å is usually considered accurate.

Note: The first column contain the rmsds and the second one the energies.

1. Load the datafile in Python and put the rmsds in one list, and the energies in another.
2. Plot the rmsds vs. energies using small black dots. Does the force-field used seem to be good in this case?

Since the local energy minimum is not exactly at 0 Å it can be hard to know if you have achieved the correct structure and the non-zero rmsd stems from the force-field, or if an incorrect structure is found. To test this a second simulation is usually run starting from the native structure to what the structure relaxes to with the given force-field.

3. Download and load the file `rmsd_energy_native.txt`.
4. Plot this together with the data from before, using a red colour. *Note!* Change the border colour of the dots to red as well.
5. Based on this second simulation, would you say that the force-field performs well?

2.5 3D Gaussian

Instead of plotting data in 3 dimensions (which is also possible with matplotlib), we can use the `plt.imshow()` function that shows the extra plane with colors.
To illustrate this method we will be using a two-dimensional independent Gaussian function, which is defined as;

$$f(x, y) = A \exp\left(-\left(\frac{(x - x_o)^2}{2\sigma_x^2} + \frac{(y - y_o)^2}{2\sigma_y^2}\right)\right) \tag{2.3}$$

where we set $x_o = y_o = 0.0$, $A = 1.0$, $\sigma_x = 1.0$ and $\sigma_y = 2.0$.

1. Define the Gaussian function, and the arrays for the x and y values.
2. Create an array `Z` with the Gaussian function values.

Now we want to illustrate the function using the `plt.imshow()` function, which is used like this:

```
extent = [X[0], X[-1], Y[0], Y[-1]]       # extent defines the edges of the plot

im = plt.imshow(Z,
                interpolation='nearest', # Disable smoothing of the data.
                extent=extent,           # Use the defined axis/edges
                origin='lower',          # Corrects the origin
                cmap='gray')             # cmap defines what colormap to be used
```

The `origin='lower'` parameter is needed to make sure the data starts at the correct position. Set `Z[-1][-1] = 1.0` and remove the parameter to see what it does.

3. Plot the Gaussian array using the above code.
4. Find a suitable colormap

You can add a colorbar by using the colorbar function.

```
cbar = plt.colorbar(im)
cbar.set_label('Text here', rotation=270)
```

Notice the rotation parameter. It is there to make it easier to read the label.

5. Add a colorbar to the plot, with and without the rotation parameter

Sometimes it is necessary to only use a part of the data only.

6. Plot the data at $x = 1.0$ for the full range of y.