Programming Project 3: Computing the Hessian Matrix Numerically

Description

The goal of the project is to compute the Hessian (second derivative matrix) of the molecular energy. That is, if $E(\mathbf{X})$ represents the energy of an N-atom molecule for a set of fixed nuclear coordinates: [1]

$$\mathbf{X} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ \vdots \\ x_N \\ y_N \\ z_N \end{pmatrix} \qquad X_1 = x_1,\ X_2 = y_1,\ \ldots,\ X_{3N} = z_N \tag{1}$$

then we wish to determine the matrix $\mathbf{H}$ with elements

$$H_{AB} = \frac{\partial^2 E}{\partial X_A \partial X_B} \tag{2}$$

which can be explicitly written as

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 E}{\partial X_1^2} & \frac{\partial^2 E}{\partial X_1 \partial X_2} & \cdots & \frac{\partial^2 E}{\partial X_1 \partial X_{3N}} \\ \frac{\partial^2 E}{\partial X_2 \partial X_1} & \frac{\partial^2 E}{\partial X_2^2} & \cdots & \frac{\partial^2 E}{\partial X_2 \partial X_{3N}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E}{\partial X_{3N} \partial X_1} & \frac{\partial^2 E}{\partial X_{3N} \partial X_2} & \cdots & \frac{\partial^2 E}{\partial X_{3N}^2} \end{pmatrix} \tag{3}$$

It is possible, but extremely complicated, to derive analytic expressions for the Hessian matrix of the energy for a specific approximation (Hartree-Fock, MP2, coupled-cluster, etc.). Another approach, which is applicable to any approximation, is to determine the Hessian at configuration $\mathbf{X}$ numerically by evaluating the energy for several nuclear configurations $\mathbf{X}', \mathbf{X}'', \ldots$ in the vicinity of $\mathbf{X}$. Numerical Hessians are necessarily approximate, but can in principle be evaluated to arbitrary precision.

For this project, we will use one of the simplest set of numerical formulas for the Hessian. Diagonal elements will be given by [2]

$$\frac{\partial^2 E}{\partial X_A^2} \approx \frac{E(X_A + h) + E(X_A - h) - 2E(X_A)}{h^2} \tag{4}$$

[1] In quantum mechanics, the "fixed nuclei" approximation is known as the Born-Oppenheimer approximation.

[2] `http://en.wikipedia.org/wiki/Finite_difference#Higher-order_differences`

where h represents some small fixed distance, $E(X_A+h)$ represents the energy obtained when the coordinate X_A is moved a distance h, and $E(X_A)$ represents the energy $E(\mathbf{X})$ at the reference configuration, $\mathbf{X}$. Off-diagonal elements will be given by [3]

$$\frac{\partial^2 E}{\partial X_A \partial X_B} \approx \frac{E(X_A+h,X_B+h)+E(X_A-h,X_B-h)-E(X_A+h,X_B)-E(X_A-h,X_B)-E(X_A,X_B+h)-E(X_A,X_B-h)+2E(X_A,X_B)}{2h^2} \tag{5}$$

where $E(X_A + h, X_B + h)$ represents the energy when coordinates X_A and X_B are both displaced by h, $E(X_A + h, X_B)$ represents the energy when only coordinate X_A is displaced, and $E(X_A, X_B)$ represents the reference energy, $E(\mathbf{X})$. This is known as the "central differences" approach, because we obtain the derivatives by displacing each coordinate forward and backward relative to its reference value. Expressions (4) and (5) may look somewhat complicated, but the derivation is actually quite straightforward and will be included at the end of this hand-out.

This is the procedure we will follow in order to evaluate these expressions:

1. read the reference molecular geometry from a file
2. generate input files to evaluate the energy of the reference configuration and each displaced configuration
3. run the energy computations described by each input file
4. read the energy for each configuration from the output file of each computation
5. plug the energy values into formulas (4) and (5) in order to construct the Hessian matrix (3)

This would be a *very* tedious and time-consuming procedure by hand. Programming allows us to put in some extra work early on in exchange for saving boat-loads of time for ourselves in the future.

[3] http://en.wikipedia.org/wiki/Finite_difference#Finite_difference_in_several_variables

1 Procedure

1.1 Script 1: Read the molecular geometry from `molecule.xyz`

1 Figure out how to read the number of atoms, the atom labels, and the Cartesian coordinates from `molecule.xyz`

There are many ways to do this, but the functions `readline()`, [4] `readlines()`, [5] and `split()` [6] provide one simple way. The first two functions act on file objects, while the `split()` function acts on a string. Here is a short example of the sort of thing one can do with the `split()` function:

```
>>> string = '1 2 3 4 5  a  b  c  d'
>>> string.split()
['1', '2', '3', '4', '5', 'a', 'b', 'c', 'd']
>>> [ int(x) for x in string.split()[0:5] ]
[1, 2, 3, 4, 5]
```

The number of atoms N should be returned as an `int`, the labels should be returned as an N-dimensional list of strings, and the Cartesian coodinates should be returned as a $3N$-dimensional list of `float`s:

$$[x_1, y_1, z_1, \ldots, x_N, y_N, z_N] \tag{6}$$

corresponding to the configuration vector $\mathbf{X}$ described above.

2 Write functions that read the number of atoms, the atom labels, and the Cartesian coordinates from `molecule.xyz`

Once you figure out how to get the number of atoms, the atom labels, and the Cartesian coordinates from `molecule.xyz`, you should write functions to return each of these in the form described in the last step. Here is an example of a function which returns the number of atoms reported in `molecule.xyz`:

```
>>> def get_N():
...   return int(open("molecule.xyz").readline())
...
>>> get_N()
3
```

Once these functions are written, you will be able to call them in your other scripts without rewriting the code. For example, if the script is called `molecule.py` and contains the `get_N()` function above, then you can import that code as follows:

```
>>> from molecule import *
>>> get_N()
3
```

The line `from molecule import *` is equivalent to pasting the contents of `molecule.py` into your file or terminal. Any code contained in that file will be run again, so imported files usually contain only functions and variable definitions.

1.2 Script 2: Generate input files for displaced geometries.

Our ultimate goal is to determine second derivatives of the molecular energy with respect to nuclear motion using equations (4) and (5). In order to do so, we need to compute the energy at several different points on the potential energy surface:

[4] `https://docs.python.org/2/library/stdtypes.html#file.readline`

[5] `https://docs.python.org/2/library/stdtypes.html#file.readlines`

[6] `https://docs.python.org/2/library/stdtypes.html#str.split`

1. singly displaced configurations: for each coordinate X_A, we need the energy of the system with that coordinate shifted to $X_A + h$ (forward displacement) as well as the energy of the system with that coordinate shifted to $X_A - h$ (backward displacement), where h is some previously fixed displacement size
2. doubly displaced configurations: for every unique pair of coordinates $(X_A, X_B), i \neq j$, we need the energy of the system with both coordinates shifted forward, $(X_A + h, X_B + h)$, and the energy of the system with both coordinates shifted backward, $(X_A - h, X_B - h)$

We also need the energy of the reference configuration – that is, the energy corresponding to the geometry in `molecule.xyz`. For this project, you should use a fixed displacement size of $h = 0.005\ a_0$ (Bohr).

The steps that follow will guide you through the procedure for generating input files for each of these configurations.

1 Write a `make_input()` function.

This function should take three arguments:

```
def make_input(dirname,labels,coords):
```

The first argument should take a single string containing the name of the directory in which the input file should be placed, the second should take a list of strings containing the atom labels, and the third argument should take a list of `floats` with coordinates in the form (6). Using this information, this function should create (or overwrite) a file named "`input.dat`" in the directory "`dirname`". The input file for the reference configuration should look something like this:

```
molecule h2o {
units bohr
  O   0.0000000  0.0000000 -0.1345037
  H   0.0000000 -1.6849167  1.0673357
  H   0.0000000  1.6849167  1.0673357
}

set basis cc-pVDZ
energy('scf')
```

The `os` module[7] may prove useful for this task – in particular, functions such as `makedirs()`[8] and `path.exists()` are worth looking at.[9] To use these functions, you must import the module by adding the line `import os` near the top of your file. You will also need the `write()` function[10], which allows you to write to a file object `f` as `f.write(string)`, provided it has been opened in write mode (`f=open(filename,"w")`). You may also want to use string formatting, [11] [12] which allows you to have a string with placeholders. Here is an example of string formatting with placeholders:

```
>>> for i in range(-3,4):
...    print "%8.3f" % pow(10,i)
...
   0.001
   0.010
   0.100
   1.000
  10.000
 100.000
1000.000
```

[7] `https://docs.python.org/2/library/os.html`
[8] `https://docs.python.org/2/library/os.html#os.makedirs`
[9] `https://docs.python.org/2/library/os.path.html#os.path.exists`
[10] `https://docs.python.org/2/library/stdtypes.html#file.write`
[11] `https://docs.python.org/2/library/stdtypes.html#string-formatting-operations`
[12] `http://www.diveintopython.net/native_data_types/formatting_strings.html`

(where `pow(a,b)` is a Python function that returns `a` to the power of `b`, a^b). The general syntax is `string % (item1,item2,...)`. This represents a string with placeholders, followed by a "modulo" operator `%` to indicate that the values for the placeholders are next, followed by a "tuple" of values to be filled in. These can be somewhat tricky to decipher at first, so I recommend getting some help to figure them out.

2 **Devise a naming scheme for the directories.**
Each directory will contain one input file in which either one or two coordinates have been displaced by a fixed amount ($h = 0.005\ a_0$) forward or backward, as described above. You should come up with a consistent scheme for naming each directory in a way that identifies which displacement it contains, and use this scheme for the next steps.

3 **Generate the input for the reference configuration.**
You should be able to do this using your `make_inputs()` function as well as the functions written in the first script. In my code, this step looks as follows:

```
from molecule import *
make_input("x0x0_00",get_labels(),get_xyz())
```

where the string `"x0x0_00"` identifies the directory for the reference configuration in my naming scheme, `get_labels()` is my function to return labels (which lives in a script I have called `molecule.py`), and `get_xyz()` is my function to return a vector of coordinates (also in `molecule.py`).

4 **Generate inputs for single displacements.**
As described above, given N atoms with 3 Cartesian coordinates per atom, we need $3N$ forward displacements and $3N$ backward displacements. You should be able to use a `for` loop, combined with your `make_inputs()` function and your functions from the first script in order to generate inputs for forward and backward single displacements.

5 **Generate inputs for double displacements.**
Given $3N$ coordinates, there are $\frac{3N(3N-1)}{2}$ unique coordinate pairs. For each pair of coordinates, we need one input file with forward displacements along both coordinates, and one input file with backward displacements along both coordinates. This step will be very similar to the previous one, except that it requires two nested `for` loops.

After running your script, you should test a few of the input files by entering the directory and executing Psi4. The command to execute Psi4 is simply `psi4`. The program expects the input file to be named `input.dat` by default, and prints to a file named `output.dat` by default.

1.3 Script 3: Run the input files.

This script will automate the process of `cd`-ing into directories and running `psi4`.

1 **Write a `run_input()` function.**
You will once again need the `os` module. The `os.chdir()` function [13] can be used to change directories, and the the `os.system()` function [14] can be used to execute a command within a given directory. While you are troubleshooting your code, the `os.getcwd()` function may be useful. This function returns a string representing the path to your current working directory.

2 **Run jobs.**
Using your `run_input()` function, you can run the job in directory `dirname` as `run_input(dirname)`. Your script should run jobs for all of the created input files: the reference configuration, the single displacements (one `for` loop), and the double displacements (two `for` loops).

[13] `https://docs.python.org/2/library/os.html#os.chdir`
[14] `https://docs.python.org/2/library/os.html#os.system`

1.4 Script 4: Compute the Hessian matrix using single-point energies

Equations (4) and (5) can be expressed more compactly as

$$E_0^{(2)} \approx \frac{E_+ + E_- - 2E_0}{h^2} \tag{7}$$

$$E_{0,0}^{(1,1)} \approx \frac{E_{+1,+1} + E_{-1,-1} - E_{+1,0} - E_{-1,0} - E_{0,+1} - E_{0,-1} + 2E_{0,0}}{2h^2} \tag{8}$$

where the superscripts represent the order of the derivative with respect to one or two variables, and the subscripts represent the type of displacement (forward, backward, or none) for the same one or two variables. If you don't like this notation, you can ignore these expressions and look at (4) and (5). This script will evaluate these expressions in order to determine

$$H_{ij} = \frac{\partial^2 E}{\partial X_A \partial X_B}$$

at the reference configuration $\mathbf{X}$ in `molecule.xyz` by reading energies from the output files of the jobs run with the previous script.

1 Write a function to return energies.
You should write one or more functions which return the energy corresponding to a particular displacement. In my code, I have a single function with definition

```
def E(i,j,hi,hj):
```

where `i` is the index of one coordinate, `j` is the index of another coordinate, and `hi,hj` represent the displacements along `i` and `j`, respectively. That is, `E(0,0,0,0)` returns the reference energy, `E(1,0,-1,0)` returns the energy for a single backward displacement along coordinate `1`, and `E(7,4,1,1)` returns the energy for a double forward displacement along coordinates `7` and `4`. Another reasonable approach is to have separate functions for single and double displacements – either one is okay.
The task of reading energies from the output files can be achieved using the same set of functions that we used to interpret `molecule.xyz` in our first script. My script uses only `readlines()`. It is also useful to note that one can obtain sections of strings similar to the way one obtains sections of lists:

```
>>> mystring = "abcdef222GFFT"
>>> mystring[0]
'a'
>>> mystring[1:6]
'bcdef'
>>> mystring[7:]
'22GFFT'
```

Here is one example of how one might extract a number from a text file:

```
>>> mystring = "line1\nline2\nline3\na = 4.0\nline5"
>>> mystring.splitlines()
['line1', 'line2', 'line3', 'a = 4.0', 'line5']
>>> for line in mystring.splitlines():
...    if line[:3] == "a =":
...      print float(line[3:])
...
4.0
```

2 Initialize the Hessian matrix as a $3N \times 3N$ `list` of `lists`.
All of the elements should be `0.0`. We will fill them in momentarily. Using two `for` loops with the `range()` function, this can be achieved in one line.

3 Fill in diagonal elements of the Hessian using equation (4).
Using your function(s) to return energies, this should look very similar to the formula itself.

4 Fill in off-diagonal elements of the Hessian using equation (5).
This will be similar to the previous step, but will require two `for` loops instead of one.

5 Print your Hessian matrix neatly to a file called `hessian.txt`.
To create/overwrite the file, you can simply `open()` it in write mode (`"w"`). You can write to a file object using the `write()` function.[15] To make things look neat, you should print each element as a formatted `float` using placeholders as described above in the instructions for the `make_input()` function. Here is an example of how one might write to a file in Python:

```
>>> f = open("newfile","w")
>>> for i in range(18):
...    f.write(" %d" % i)
...
>>> f.close()
>>> open("newfile").read()
' 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17'
```

Once this is done, you should check your answer. If it doesn't match, you have some debugging to do – welcome to programming. If it does match, congratulate yourself! [16]

[15] `https://docs.python.org/2/library/stdtypes.html#file.write`

[16] Do not expect all of your programs to work without debugging. Learning to find your mistake is almost as important as learning to program in the first place.

2 Extra Files

2.1 `molecule.xyz`

The `.xyz` format is a standard chemical file format used by visualization programs such as Jmol and cheMVP. Typically the units for `.xyz` files are Ångströms (Å), but the one we are using is in Bohr (a_0). The general structure is as follows:

```
N
COMMENT LINE
A1 x1 y1 z1
A2 x2 y2 z2
...
AN xN yN zN
```

where the first line contains an integer indicating the number (N) of atoms in the molecule, the second line is generally either blank or contains text describing the system, and the final N lines each contain an atomic symbol followed by three numbers indicating the Cartesian coordinates of the atom (hence the extension `.xyz`). For this project, we will start with water:

```
3
WATER MOLECULE (units Bohr)
O            0.000000000000      0.000000000000     -0.134503695264
H            0.000000000000     -1.684916670000      1.067335684736
H            0.000000000000      1.684916670000      1.067335684736
```

3 Derivation of Central Difference 2nd Derivative Formulas

To begin, it should be pointed out that you are likely already familiar with one finite difference formula:

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$$

This expression is used to define derivatives in Calculus I. In the language used by applied mathematicians, this would be called a first-order, forward-difference derivative. That is, a first derivative determined by displacing the function argument by $+h$. An analogous central-difference derivative expression would be

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x-h)}{2h}$$

which involves both forward and backward displacements.

The most straightforward approach to deriving such formulas starts from a Taylor expansion.

$$f(x) = \sum_{n=0}^{\infty} \frac{1}{n!} \left.\frac{\partial^n f}{\partial x^n}\right|_{x=0} x^n \tag{9}$$

Which, for our purposes, is more conveniently expressed as

$$f(x) = f_0 + f_0^{(1)} x + \frac{1}{2!} f_0^{(2)} x^2 + \frac{1}{3!} f_0^{(3)} x^3 + \mathcal{O}(x^4) \tag{10}$$

$f_0^{(n)}$ here simply denotes the value of the n^{th} derivative of f at $x = 0$. The last term, $\mathcal{O}(x^4)$, describes the magnitude of the error incurred by truncating the Taylor expansion at a particular point – we are interested in a very small region about $x = 0$, so higher powers of x will generally mean better accuracy.

The basic strategy used to derive expressions for the derivatives is to plug in various x-values $\{x_k\}$, yielding a series of equations, and then solve for the derivatives in terms of the function values at these points $\{f(X_A)\}$. Here we will take the "uniform grid" approach, in which we use x values with a fixed interval h between them $\{\ldots, -2h, -h, 0, +h, +2h, \ldots\}$.

3.1 Derivatives of one variable

To get a first approximation of the first and second derivatives in one variable, we only need three points from our grid $\{-h, 0, +h\}$. Let $f_{\pm k}$ represent $f(\pm kh)$ to keep things a little bit neater.

$$f_{+1} = f_0 + f_0^{(1)} h + \frac{1}{2!} f_0^{(2)} h^2 + \mathcal{O}(h^3)$$
$$f_{-1} = f_0 - f_0^{(1)} h + \frac{1}{2!} f_0^{(2)} h^2 - \mathcal{O}(h^3)$$

Hopefully you can see that when we subtract the two equations the even powers of h cancel, which allows us to solve for the first derivative to third order in h.

$$f_0^{(1)} = \frac{f_{+1} - f_{-1}}{2h} + \mathcal{O}(h^3)$$

Similarly, adding the two equations cancels even powers which allows us to solve for the second derivative to fourth order in h.

$$f_0^{(2)} = \frac{f_{+1} + f_{-1} - 2f_0}{h^2} + \mathcal{O}(h^4)$$

You can easily check this on paper.

This formula can be used at $x_0 \neq 0$ as well by displacing relative to x_0 rather than 0.

$$\left(\frac{\partial^2 f}{\partial x^2}\right)_{x=x_0} = \frac{f(x_0+h) + f(x_0-h) - 2f(x_0)}{h^2} + \mathcal{O}(h^4)$$

Since total derivatives of a one-variable function work just like partial derivatives of one variable in a multivariate function, this generalizes directly to the latter case.

3.2 Mixed partials

Since we are ultimately interested in computing second derivatives for a multivariate function, we will also need to find an expression for mixed partial derivatives of the form $\frac{\partial^2 f}{\partial x \partial y}$. The Taylor expansion for a multivariate function is

$$f(x_1, \ldots, x_n) = f_0 + \sum_{i=1}^{n} \left(\frac{\partial f}{\partial X_A}\right)_0 X_A + \frac{1}{2!} \sum_{i,j=1}^{n} \left(\frac{\partial^2 f}{\partial X_A \partial X_B}\right)_0 X_A X_B + \frac{1}{3!} \sum_{i,j,k=1}^{n} \left(\frac{\partial^3 f}{\partial X_A \partial X_B \partial x_k}\right)_0 X_A X_B x_k + \ldots$$

For a bivariate function, we can expand this up to second derivatives (extending the notation used above) as

$$f(x,y) = f_{0,0} + f_{0,0}^{(1,0)} x + f_{0,0}^{(0,1)} y + \frac{1}{2!} f_{0,0}^{(2,0)} x^2 + f_{0,0}^{(1,1)} xy + \frac{1}{2!} f_{0,0}^{(0,2)} y^2 + \ldots$$

where we are interested in solving for $f_{0,0}^{(1,1)} \equiv \left.\frac{\partial^2 f}{\partial x \partial y}\right|_{x=y=0}$. The only new points needed for this are

$$\begin{aligned} f_{+1,+1} &= f_{0,0} + f_{0,0}^{(1,0)} h + f_{0,0}^{(0,1)} h + \frac{1}{2!} f_{0,0}^{(2,0)} h^2 + f_{0,0}^{(1,1)} h^2 + \frac{1}{2!} f_{0,0}^{(0,2)} h^2 + \mathcal{O}(h^3) \\ f_{-1,-1} &= f_{0,0} - f_{0,0}^{(1,0)} h - f_{0,0}^{(0,1)} h + \frac{1}{2!} f_{0,0}^{(2,0)} h^2 + f_{0,0}^{(1,1)} h^2 + \frac{1}{2!} f_{0,0}^{(0,2)} h^2 - \mathcal{O}(h^3) \end{aligned}$$

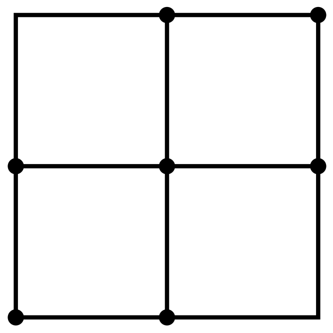
By adding together these equations, we obtain

$$f_{0,0}^{(1,1)} = \frac{f_{+1,+1} + f_{-1,-1} - 2f_{0,0}}{2h^2} - \frac{f_{0,0}^{(2,0)} + f_{0,0}^{(0,2)}}{2} + \mathcal{O}(h^4)$$

Since $f_{0,0}^{(2,0)}$ and $f_{0,0}^{(0,2)}$ are partial derivatives with respect to a single variable, we can plug in the expression for $f_0^{(2)}$ derived above, giving

$$f_{0,0}^{(1,1)} = \frac{f_{+1,+1} + f_{-1,-1} - f_{+1,0} - f_{-1,0} - f_{0,+1} - f_{0,-1} + 2f_{0,0}}{2h^2} + \mathcal{O}(h^4)$$

Such expressions are sometimes described according to the grid of points used in them. In this case, we have:



where the middle point is the origin and all line segments have length h.

As for derivatives of a single variable, the generalization to arbitrary reference values is straightforward:

$$\left(\frac{\partial^2 f}{\partial x \partial y}\right)_{\substack{x=x_0 \\ y=y_0}} = \frac{f(x_0+h, y_0+h) + f(x_0-h, y_0-h) - f(x_0+h, y_0) - f(x_0-h, y_0) - f(x_0, y_0+h) - f(x_0, y_0-h) + 2f(x_0, y_0)}{2h^2} + \mathcal{O}(h^4)$$

and it makes no difference wether or not the function has more than two variables, since these are held constant for partial derivatives.