SciComp Project 3
Week 4+5 (10 points)

September 20, 2020
To be handed in October 4 before 12:00 noon

1 Background

1.1 Importance of Rapid Convergence in Root Finding and Optimization

Finding a minimum energy configuration for a system of mutiple atoms is a widely applicaple for designing molecular systems in solid state physics, chemistry, medicine and other fields. Many applications will require tens to hundreds of atoms in the model. Calculating the energy involves solving the many-body Schrødinger equation, in which all the electrons interact with each other.

With hundreds of atoms and thousands of electrons, solving the many-body Schrödinger equation (even approximately) can take weeks of computetime to calculate a single energy point, whereby a full optimization can easily take many years of CPU-time to run. Even on a parallel computer with, say, 1000 CPU cores, optimizing the geometry of a single molecule can easily take months in "human time".

When every evaluation of the energy function takes, for example, a CPU-week to compute (burning large amounts of CO_2), the number of function evaluations matter: a problem that may be extra bad in quantum chemistry, but is common to many fields.

In this assignment, you will try your hand at optimizing a simple system of interacting atoms. However, we do not have months to sit and wait for computations to complete, so we will use an extremely simple approximation to the potential energy, the *Lennard-Jones*-potential. This approximation is appropriate for noble gas atoms, yielding a crude picture of the formation; yet we will pretend that it is a full *ab initio* quantum chemical energy calculation, and assume that it takes a week to complete. Thus, you will be asked to *count energy function evaluations* and report the time spent.

1.2 Minimizing the potential in a cloud of Argon atoms using the Lennard-Jones potential

The Lennard-Jones potential works for systems of neutral atoms or molecules. It assumes that no new electronic bonds nor new molecules are formed, and that the potential can be modeled only with a short-range repulsion force (arising from the Pauli exclusion principle) and a long-range

van der Waals attraction. This yields a classical description of the system, in which the neutral atoms or molecules move as classical particles, interacting only through this simple potential.

The Lennard-Jones potential can be written in several ways, but the most common is:

$$v_{LJ}(r_{ij}) = 4\epsilon \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right) \tag{1}$$

$v_{LJ}(r_{ij})$ is the potential at distance $r_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|_2$ between two atoms i and j. The repulsive force is stronger than the van der Waals-force, but decreases more rapidly with distance. Thus, for a pair of atoms, there is a distance where the total potential is minimal, the potential well. ϵ is this minimal potential between two atoms, and σ is the inter-particle distance where the potential is zero: $v_{LJ}(\sigma) = 0$. The constants ϵ and σ are generally found by fitting to proper ab initio calculations, or using experimental data.

The full potential LJ-energy of N neutral particles is the sum of pair-potentials:

$$V_{LJ}(\mathbf{X}) = \sum_{i=1}^{N} \sum_{j=i+1}^{N} v_{LJ}(r_{ij}) \tag{2}$$

where $\mathbf{X} \in \mathbb{R}^{N \times 3}$ is the matrix of coordinates for the N particles.

In this assignment, we will be modeling argon, for which

$$\sigma = 3.401\text{Å} \text{ and } \epsilon = 0.997\text{kJ/mol} \tag{3}$$

1.3 Array Programming

Try to use array programming when programming with Python/Numpy or Matlab, as the speed difference can be up to a factor of hundreds. As an example of how to use array programming, here is how a distance matrix (to be used in the LJ-potential) can be calculated with array operations using `NA = newaxis` (as I explain further here):

$$D_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|_2 = \|\mathbf{x}_{i\mathbf{j}} - \mathbf{x}_{\mathbf{i}j}\|_2 \tag{4}$$

In Eq. (4), $\mathbf{x}_i$ is extended with a `newaxis j` and $\mathbf{x}_j$ with a `newaxis i` to put them on the same footing, so that the pair differences can be calculated as elementwise subtraction $\mathbf{x}_{i\mathbf{j}} - \mathbf{x}_{\mathbf{i}j}$. In Numpy code, it looks as follows:

```
# points:              (N,3)-array of (x,y,z) coordinates for N points
# distance(points): returns (N,N)-array of inter-point distances:
def distance( points ):
        displacement = points[:,NA] - points[NA,:]
        return sqrt( sum(displacement*displacement, axis=-1) )
```

2 Questions for Week 4: Solving Nonlinear Equations

2.1 Solving Nonlinear Equations in 1D

a. Write a function that computes the Lennard-Jones energy for a collection of particles described by an $(N, 3)$-array `points`. The resulting function should not take more arguments than this, as it will be called with by your root-finding functions below. In Python, you can do this using a programming construct called *closures* as follows:

```
def LJ(sigma,epsilon):
  def V( points ):
    ...implementation that depends on sigma and epsilon...
return V
```

Then $V = LJ(\sigma, \epsilon)$ will return a function specialized to σ and ϵ such that $V($`points`$)$ returns the total Lennard-Jones energy for the system.

Demonstrate your solution by making 1) a plot of the potential between two Ar atoms, one placed at $\mathbf{x}_0 = (x, 0, 0)$ the other at $\mathbf{x}_1 = (0, 0, 0)$ with x ranging from 3 to 11; and 2) a plot of the LJ-potential in the same range with two extra points added: $\mathbf{x}_2 = (14, 0, 0)$ and $\mathbf{x}_3 = (7, 3.2, 0)$.

b. Write a bisection root finding function `x, n_calls = bisection_root(f,a,b,tolerance=1e-13)` that finds `x` such that $f($`x`$) = 0$ given a bracket $x \in [a; b]$, and counts the number of calls to the function f. (A reasonable length is 8-12 lines of code).

In this assignment, let the convergence test be on how close we get $f(x)$ to zero. Test it to find the zero of the the LJ-potential between two argon atoms as a function of interatomic distance, and verify that you get $x = \sigma$. How many calls to the energy function were needed to get from the start bracket $[a, b] = [2, 6]$ to $|f(x)| < 10^{-13}$?

c. The derivative of the pair-potential $v_{LJ}(r) = 4\epsilon((\sigma/r)^{12} - (\sigma/r)^6)$ is

$$\frac{d}{dr} v_{LJ}(r) = 4\epsilon \left(\frac{6\sigma^6}{r^7} - \frac{12\sigma^{12}}{r^{13}} \right)$$

Write a Newton-Rhapson solver `x, n_calls = newton_root(f,df,x0,tolerance,max_iterations)`, (a reasonable length is 4-8 lines of code), and test it in the same way as above. For simplicity, assume that a call to the derivative `df` has the same cost as a call to `f`. How many calls were needed to get from $x_0 = 2$ to $|f(x^*)| < 10^{-12}$, i.e., 12 decimal digits for x^* after the comma?

d. Make a combination of Newton-Rhapson and bisection that is *guaranteed to converge*, but takes advantage of the quadratic convergence of Newton-Rhapson iteration, and test it on the same example. How many calls to the LJ-energy function was needed to get from $x_0 = 2$, $[a, b] = [2, 6]$ to obtain $|f(x^*)| < 10^{-13}$?

Note: *If you have trouble completing this step, simply skip it and move on to the remaining questions, which only requires your bisection root solver to work. You can always return to solve it once you have completed tasks (e) and (f).*

2.2 Solving N-dimensional Nonlinear Equations

Using the chain rule for derivatives $f(g(x))' = f'(g(x))g'(x)$, we can write down the derivative of a pair-potential with respect to the position of one of the particles:[1]

$$\begin{aligned}\frac{\partial}{\partial \mathbf{x}_i} v_{LJ}(r_{ij}) &= 4\epsilon \left(\frac{6\sigma^6}{r_{ij}^7} - \frac{12\sigma^{12}}{r_{ij}^{13}} \right) \frac{\partial r_{ij}}{\partial \mathbf{x}_i} \\ &= 4\epsilon \left(\frac{6\sigma^6}{r_{ij}^7} - \frac{12\sigma^{12}}{r_{ij}^{13}} \right) \frac{\mathbf{x}_i - \mathbf{x}_j}{r_{ij}}\end{aligned} \tag{5}$$

This we can then use to find an expression for the gradient of the total LJ-energy:

$$\begin{aligned}\frac{\partial}{\partial \mathbf{x}_k} V_{LJ}(\mathbf{X}) &= \frac{\partial}{\partial \mathbf{x}_k} \frac{1}{2} \sum_{i=1}^{N} \sum_{j \neq i} v_{LJ}(r_{ij}) \\ &= \frac{\partial}{\partial \mathbf{x}_k} \sum_{i=1}^{N} \frac{1}{2} \sum_{j \neq i}^{N} (\delta_{i,k} + \delta_{j,k}) v_{LJ}(r_{ij}) \\ &= \frac{\partial}{\partial \mathbf{x}_k} \sum_{j \neq k}^{N} v_{LJ}(r_{kj}) \\ &= 4\epsilon \sum_{j \neq k}^{N} \left(\frac{6\sigma^6}{r_{kj}^7} - \frac{12\sigma^{12}}{r_{kj}^{13}} \right) \frac{\mathbf{x}_k - \mathbf{x}_j}{r_{kj}}\end{aligned} \tag{6}$$

The position of all the particles is a point $\mathbf{X} \in \mathbb{R}^{3N}$, which we organize as a $(N, 3)$-array, so that $\mathbf{x}_i = (x_i, y_i, z_i)$. The total potential energy is a function $V_{LJ} : \mathbb{R}^{3N} \to \mathbb{R}$, and the gradient taken at any particular configuration of particle positions is hence a $3N$-dimensional vector: $V_{LJ}(\mathbf{X}) \in \mathbb{R}^{3N}$. Thus, the gradient to V_{LJ} is a function $\nabla V_{LJ} \colon \mathbb{R}^{3N} \to \mathbb{R}^{3N}$. The negative of the gradient is the *force* acting on the system, and its direction is that in which the potential decreases most rapidly. But notice that it is a $3N$-dimensional direction: it acts on *all* the particles at once.

We can write it down in Python as follows (in Matlab, it would look very similar):

```
def LJgradient(sigma, epsilon):

    def gradV(X):
        d = X[:,NA] - X[NA,:]         # (N,N,3) displacement vectors
        r = sqrt( sum(d*d,axis=-1) ) # (N,N)   distances

        fill_diagonal(r,1)           # Don't divide by zero
        T = 6*(sigma**6) * (r**-7) - 12*(sigma**12) * (r**-13) # (N,N)-matrix of r-derivatives

        # Using the chain rule, we turn the (N,N)-matrix of r-derivatives into
        # the (N,3)-array of derivatives to Cartesian coordinate: the gradient.
        # (Automatically sets diagonal to (0,0,0) = X[i]-X[i])
        u = d/r[:,:,NA]       # u is (N,N,3)-array of unit vectors in direction of X[i] - X[j]
        return 4*epsilon*sum(T[:,:,NA] * u, axis=1)

    return gradV
```

[1] NB: You don't need to understand these derivations to solve the problem, as the actual code for the gradient is provided to you. The derivations are here to show you how to do it yourself in the future.

Finding the minima in $\mathbb{R}^{3N}$ of the potential involves finding points where its $3N$-dimensional gradient is $\mathbf{0}$. An important component needed in next week's work is called *line search*, where we search for a point along a single direction $\mathbf{d} \in \mathbb{R}^{3N}$ at which the gradient *along this line* is zero. That is, we want to start in a point $\mathbf{X}_0 \in \mathbb{R}^{3N}$, and then find $\alpha \in [0; b]$ such that $0 = \mathbf{d} \cdot \nabla V_{LJ}(\mathbf{X}_0 + \alpha\mathbf{d})$, i.e., the gradient has no component in the direction of $\mathbf{d}$. This finds an optimum for the one-dimensional function $V_{LJ}(\mathbf{X}_0 + \alpha\mathbf{d})$.

e. Look at the gradient of the 2-particle system in question (a) with $\mathbf{x}_1 = (0, 0, 0)$ and $\mathbf{x}_0 = (x, 0, 0)$, $x \in [3; 10]$. Why are exactly two components nonzero? Why are they equal and opposite? Plot the nonzero component for the derivative of the x-coordinate of $\mathbf{x}_0$ ($0, 0$-coordinate of gradient) together with the potential, and notice the relationship between the zero of the derivative and the minimum of the potential.

   Next look at the gradient for the 4-particle system from (a) at one of the minima of your plot. Why is the gradient not zero?

f. Write a function `x0, ncalls = linesearch(F,X0, d, alpha_max, tolerance, max_iterations)` that takes a function $\mathbf{F} : \mathbb{R}^{N\times 3} \to \mathbb{R}^{N\times 3}$, a start position $\mathbf{X}_0 \in \mathbb{R}^{N\times 3}$ and finds the zero along the line-segment $\mathbf{X}_0 + \alpha\mathbf{d}$ of $\mathbf{d} \cdot \mathbf{F}(\mathbf{X}_0 + \alpha\mathbf{d})$.[2] Use your bisection solver at this point, as using Newton-Rhapson requires second derivatives when $\mathbf{F}$ is the gradient.

   Test your function by finding the mimimum along $\mathbf{X}_0 + \alpha\mathbf{d}$ with

$$\mathbf{X}_0 = \begin{bmatrix} 4 & 0 & 0 \\ 0 & 0 & 0 \\ 14 & 0 & 0 \\ 7 & 3.2 & 0 \end{bmatrix}$$

   and $\mathbf{d} = -\nabla V_{LJ}(\mathbf{X}_0)$, and with $\alpha \in [0; 1]$.

[2] You can use either a closure or a lambda expression to define the one-dimensional restriction of $\mathbf{F}$ to the line.