Scientific Computing Project 3

Jakob Schauser, pwn274

October 2023

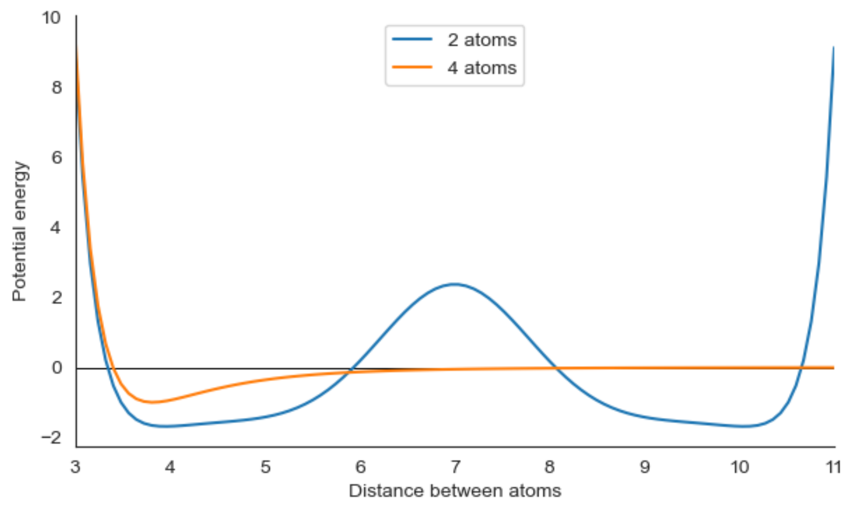
a

(1) & (2) Plot the strength of the potential between two/four particles

Using:

```python
from LJhelperfunctions import V

def pot_1(x : float) -> float:
    points = np.stack(([0,0,0],[x,0,0]))
    return V(points)

def pot_2(x : float) -> float:
    points = np.stack(([0,0,0],[x,0,0],[14,0,0],[7,3.2,0]))
    return V(points)
```

I get the following image:



b

Okay I have gone slightly above the recommended line count, but that is only because I had some good ideas I wanted to implement. Firstly I count the number of function calls in a very Object Oriented python-esque way. Secondly, I minimize the needed function calls by keeping track of what has been calculated already:

```python
def bisection_root(f, a : float, b : float, tolerance : float=1e-13 ) -> tuple[float, int]:
    def f_count(x):
        f_count.count += 1
        return f(x)
    f_count.count = 0

    a_is_previous_m = False
    while ((b - a) > tolerance):
        m = a + (b - a)/2
        fa = f_count(a) if not a_is_previous_m else fm
        fm = f_count(m)
        if np.sign(fa) == np.sign(fm):
            a = m
            a_is_previous_m = True
        else:
            b = m
            a_is_previous_m = False

    return m, f_count.count
```

Running this i get:

```python
from project_files.LJhelperfunctions import SIGMA

ans = bisection_root(pot_1, 2, 6)

print(ans)
print("Same?", same(ans[0], SIGMA))

> (3.4010000000000105, 65)
> Same? True
```

Meaning it takes 65 calls to the function to get within a tolerance of 10^{1-3}.

c

Write a Newton-Rhapson solver. How many calls were needed?

```python
def newton_root(f,df,x0,tolerance, max_iterations) -> tuple[float, int]:
    for i in range(max_iterations):
        f_val = f(x0)
        x0 -= f_val/df(x0)
        if abs(f_val) < tolerance:
            break

    return x0, i*2 # 2 calls to f(x) (or f'(x)) per iteration

ans = newton_root(pot_1, dpot_1, 2, 1e-12, 1000)
print(ans)
> (3.4009999999999994, 24)
```

It takes 24 calls to get to within 12 decimals of sigma.

d

Make a combination of Newton-Rhapson and bisection. How many calls to the LJ-energy function was needed?

```
def root(f,df,x0, tolerance, max_iterations) -> tuple[float, int]:
    x = x0

    last = abs(f(x0))
    for i in range(max_iterations):
        f_val = f(x)

        x = x - f(x)/df(x)

        if abs(f_val) < tolerance:
            # when converging use newton
            return x, i*2 # 2 calls to f(x) (or f'(x)) per iteration

        if f_val > last:
            print("Diverging")
            # when diverging use bisection
            ans = bisection_root(f, -f_val, last, tolerance)
            return ans[0], i*3 + ans[1]

        last = f_val

    return x, i*2 # 2 calls to f(x) (or f'(x)) per Newton iteration
```

I am unsure if this is the way you wanted me to combine them. It takes 24 function calls to get to an accuracy of 10^{-13}

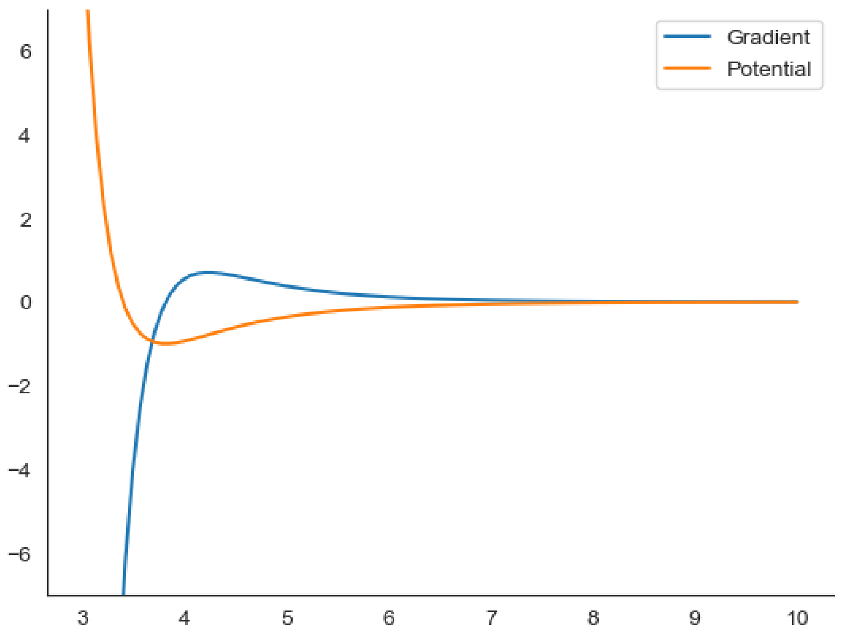
e

Why are exactly two components nonzero? Why are they equal and opposite?

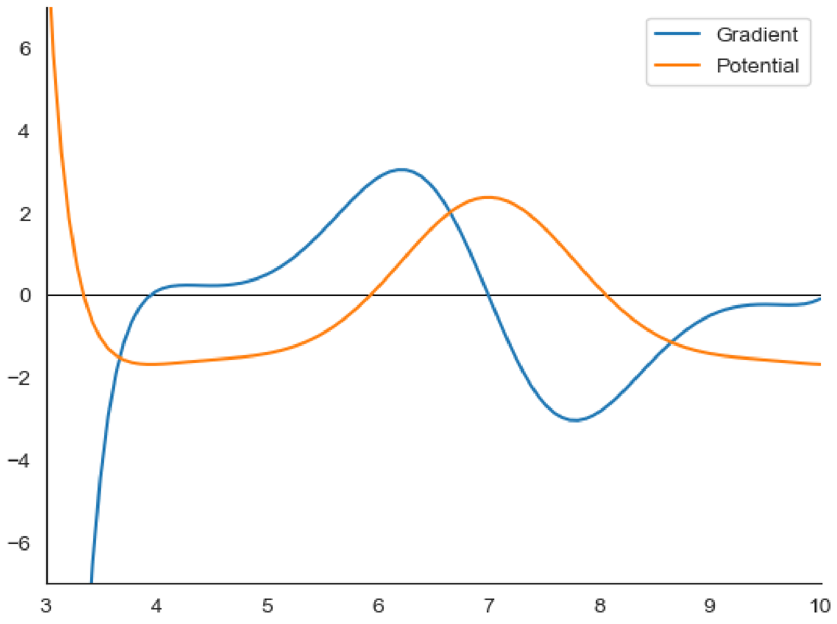
Everything happens in the x-component.

TODO: Needs more text (see feedback)

Plot the nonzero component for the derivative.



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 alpha, ncalls = linesearch

```
def make_line_func(X0, d, f):
    def f_line(alpha):
        return f(X0 + alpha * d)
    return f_line

def make_line_deriv(X0, d, f):
    lf = make_line_func(X0, d, f)
    def f_line_deriv(alpha):
        return d.dot(lf(alpha))
    return f_line_deriv

def linesearch(F, X0, d, alpha_max=1, tolerance=1e-13, max_iterations=1e4)->tuple[float, int]:
    f_line = make_line_deriv(X0, d, F)

    alpha, ncalls = bisection_root(f_line, 0, alpha_max, tolerance=tolerance)
    return alpha, ncalls
```

on x0 i get 0.45171 using 80 calls

g

Write a function x opt, n calls = golden section min(f,a,b,tolerance=1e−3) that finds the minimum of a 1D-function on a unimodal interval x [a; b] and use it to obtain the same α as you did in (f)

here is the function. Simply copied from the pseudocode in the book:

```
def golden_section_min(f,a,b,tolerance=1e-3) -> tuple[float, int]:
    phi = (np.sqrt(5) - 1)/2
    x1 = a + (1 - phi)*(b - a)
    f1 = f(x1)
    x2 = a + phi*(b - a)
    f2 = f(x2)

    count = 2 # already two calls to the function

    while ((b - a) > tolerance):
        if (f1 > f2):
            a = x1
            x1 = x2
            f1 = f2
            x2 = a + phi*(b - a)
            f2 = f(x2)
        else:
            b = x2
            x2 = x1
            f2 = f1
            x1 = a + (1 - phi)*(b - a)
            f1 = f(x1)

        count += 1

    return (a + b)/2, count
```

Running this I get 0.45167 using 17 calls. So, a lot fewer calls, but not the exact same α-value. I suspect lowering the tolerance (from the default-parameter of 10^{-3}) would make the numbers converge while executing more function calls.

Next, use your golden section function to obtain the optimal (minimal-energy) distance r0 between two Ar atoms.

This is a simple function call:

```
r0 = golden_section_min(pot_1, 2, 6)[0]
> 3.81729
```

h

Write a function X opt, N calls, converged = BFGS(f,gradf, X,tolerance=1e−6, max iterations=10000) which implements BFGS

I implement the Broydan–Fletchen–Goldfare–Shannon algorithm.

```
def BFGS(f, gradf, X, tolerance = 1e-6, max_iterations = 10000, linesearch = False) -> tuple[
    ArrayLike, int, bool]:
    x0 = flatten_gradient(X)
    x0 = X
    B0 = np.eye(len(X))
    gradfx0 = gradf(x0)
    for k in range(1,max_iterations):
        sk = np.linalg.solve(B0, -gradfx0)
        x1 = x0 + sk
        gradfx1 = gradf(x1)
        yk = gradfx1 - gradfx0

        if np.linalg.norm(yk) < tolerance:
            return x1, k, True
        B0 += np.outer(yk, yk)/np.dot(yk, sk) \
                - np.outer(np.dot(B0, sk), np.dot(B0, sk))/np.dot(sk, np.dot(B0, sk))

        x0 = x1
        gradfx0 = gradfx1

    return x0, 1 + k, False


# As the algorithm returns the positions, I will use the 'distance' helper function
ans = BFGS(flat_V, flat_gradV, ArStarts["Xstart2"])
print(distance(ans[0].reshape(-1,3)))
```

As a stopping criterion I have chosen to look at the norm of the gradient matrix, as this will approach zero as we get closer to a minimum/maximum.

Reading out the minimum distance, I get 3.81749. In full correspondence with the r_0 above.

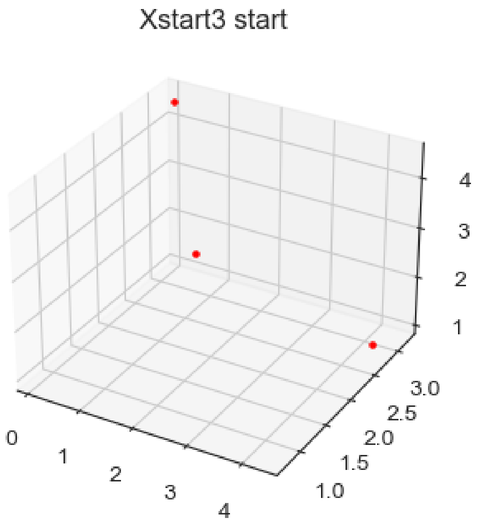
i

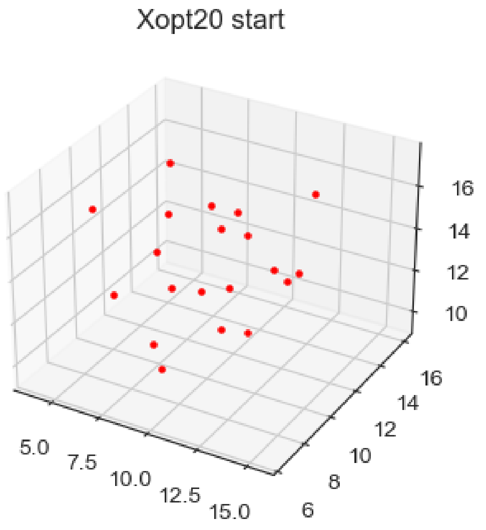
Apply your BFGS-minimizer to the starting geometries from ArStarts, starting with N = 2 and stopping when you reach an N you can't get to converge. Inspect the distance matrix. How many distances are within 1% of the two-particle optimum r0? You can count them automatically by writing sum(abs(D−r0)/r0 <= 0.02)//2 if D is the distance matrix.

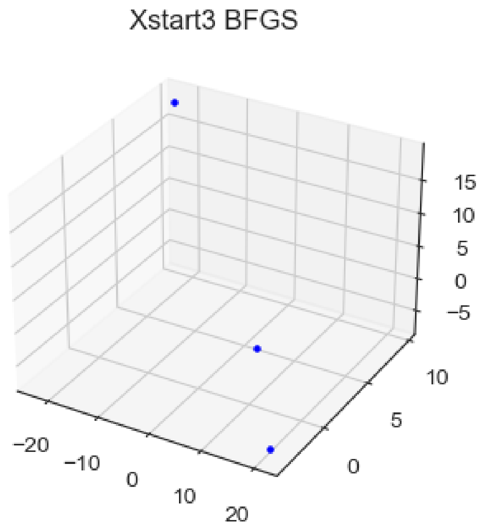
I am unsure whether I have done something wrong. Every single of the ArStart-geometries converge, but only the first one has any points that are close to r_0. For good measure here is the output (even though this seems dumb to include, I am unsure what else to write here)

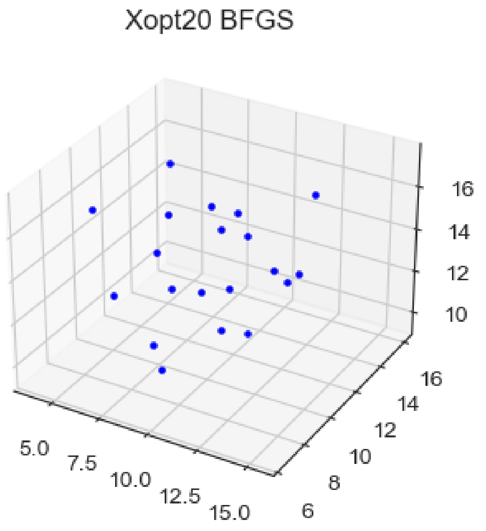
```
ArStart 0
n_close: 1
converged True

ArStart 1
n_close: 0
converged True

ArStart 2
n_close: 0
converged True

ArStart 3
n_close: 0
converged True

ArStart 4
n_close: 0
converged True

ArStart 5
n_close: 0
converged True

ArStart 6
n_close: 0
converged True

ArStart 7
n_close: 0
converged True

ArStart 8
n_close: 0
converged True

ArStart 9
n_close: 0
converged True
```

Inspect the results in 3D. The true minimum corresponds to the zero-temperature configuration of the system if left to slowly cool. A good optimum should look something like a single lattice, where as many atoms as possible are trapped in the potential well of one or more neighbours. Show the 3D picture for N = 3 and your highest converged N.









Hmm it does not exactly look like a grid to me :(