Scientific Computing Project 3

Jakob Schauser, pwn274

October 2023

a

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

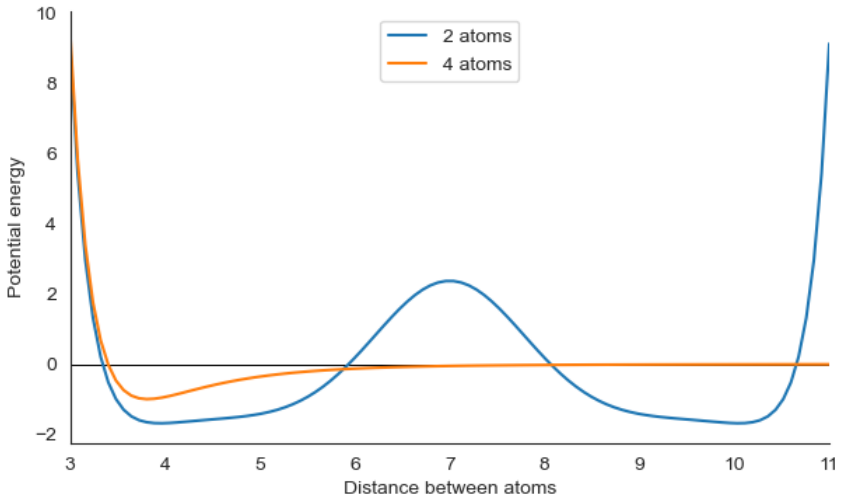
Using:

```
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:

```
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:

```
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?

```
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
```

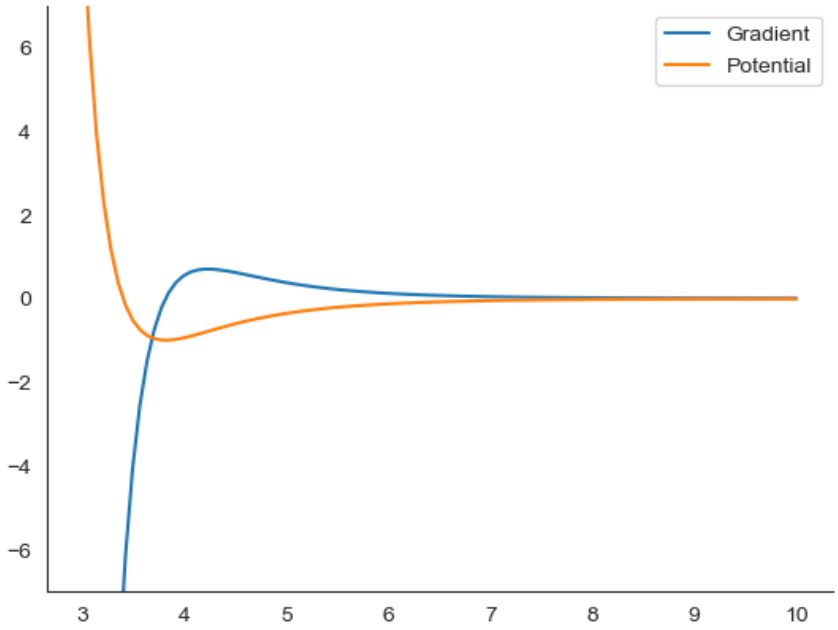
I am unsure if this is the way you wanted me to combine them. It takes 24 function calls to get to an accuracy of 10^{-13}

e

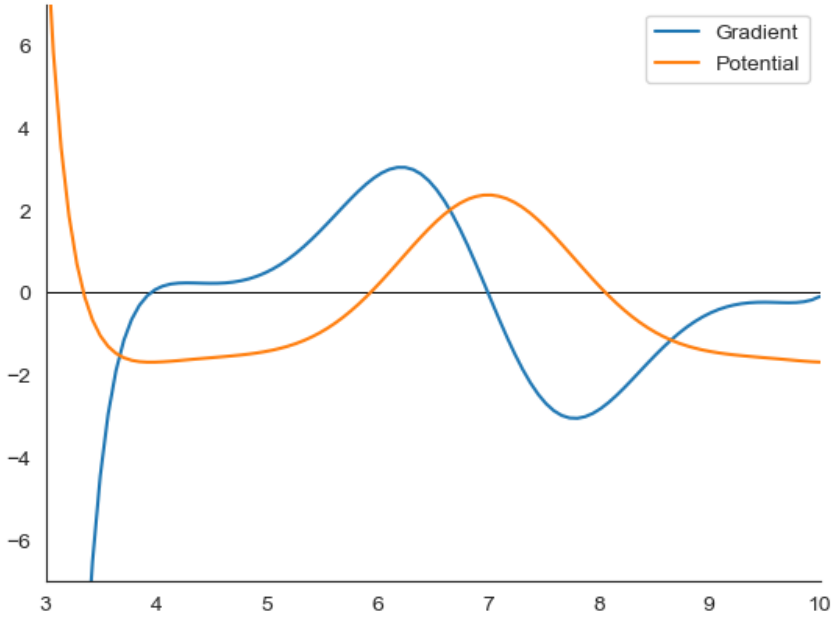
Why are exactly two components nonzero? Why are they equal and opposite?

Everything happens in the x-component.

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

0.1 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 0.451707051842277 using 80 calls

(3)

(4)