Scientific computing: Project 3

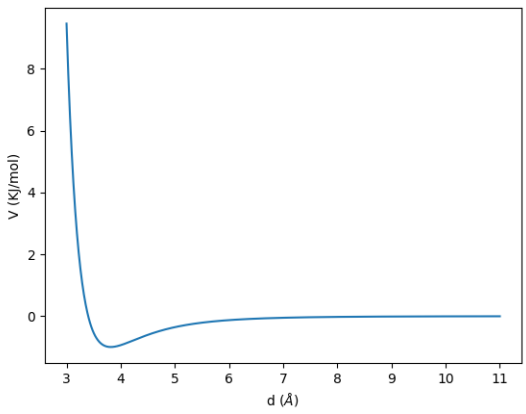
Tomás Fernández Bouvier

October 6, 2020

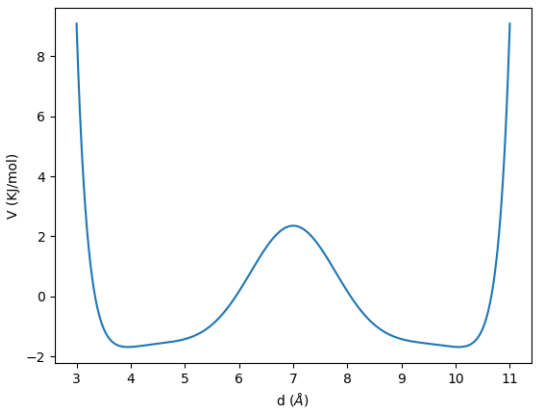
1

a)

```
def LJ(sigma,epsilon, derivative):
  def V(points):
    distance=dist(points)
    if(derivative==0):
      potential=  4*epsilon*((sigma/distance)**(12)- (sigma/distance)**(6))
    elif(derivative==1):
      potential= - 4*epsilon*(12*(sigma**(12)/distance**(13))- 6*(sigma**(6)/
   distance**(7)))
    else:
      print("derivative degree too high")
      return
    E=0
    for i in range(np.shape(potential)[0]):
      for j in range( i+1, np.shape(potential)[1]):
        E+= potential[i,j]
    return(E)
  return(V)
```



(a) 2 particles



(b) 4 particles

Figure 1: LJ potential for $\sigma = 3.401$ and $\epsilon = 0.997$ KJ/mol

b)

```
def bisection_root(f,a,b,tolerance= 1e-13):
  n_calls=0
  m= a+(b-a)/2
  while(abs(f(m))>tolerance):
    m=a+(b-a)/2.;
    if(np.sign(f(a))== np.sign(f(m))):
      a=m
    else:
      b=m
    n_calls+=1
  return(m, n_calls)
```

With this function, after 46 calls, I obtain $r_0 = 3.401$ which is equal to σ.

c)

```
def newton_root(f, df, x0, tolerance=1e-12, max_iterations=30):
  x=x0
  iter=0
  while(abs(f(x))>tolerance):
    x-=f(x)/df(x)
    iter+=1
    if(iter==max_iterations):
      break
  return(x, iter)
```

With this function, after 12 calls, I obtain $r_0 = 3.401$ which is also equal to σ.

d)

```
def super_root_finder(f,df, a,b, tolerance=1e-12):
  n_calls=0
  x= a+(b-a)/2
  while(abs(f(x))>tolerance):
    x-=f(x)/df(x)
    if(x<a or x>b):
      x=a+(b-a)/2.;
      if(np.sign(f(a))== np.sign(f(x))):
        a=x
      else:
        b=x
    n_calls+=1
  return(x, n_calls)
```

e)

The gradient of the potential represents the forces acting on the particles of the system. In our case we have two particles so the components of the gradient (as predicted by Newton's third law) will be equal and oposite in direction. In the case of 4 Argon particles, even if we are at a minimum in the potential, we are not able to fully reach a steady state so the gradient will be non-zero.

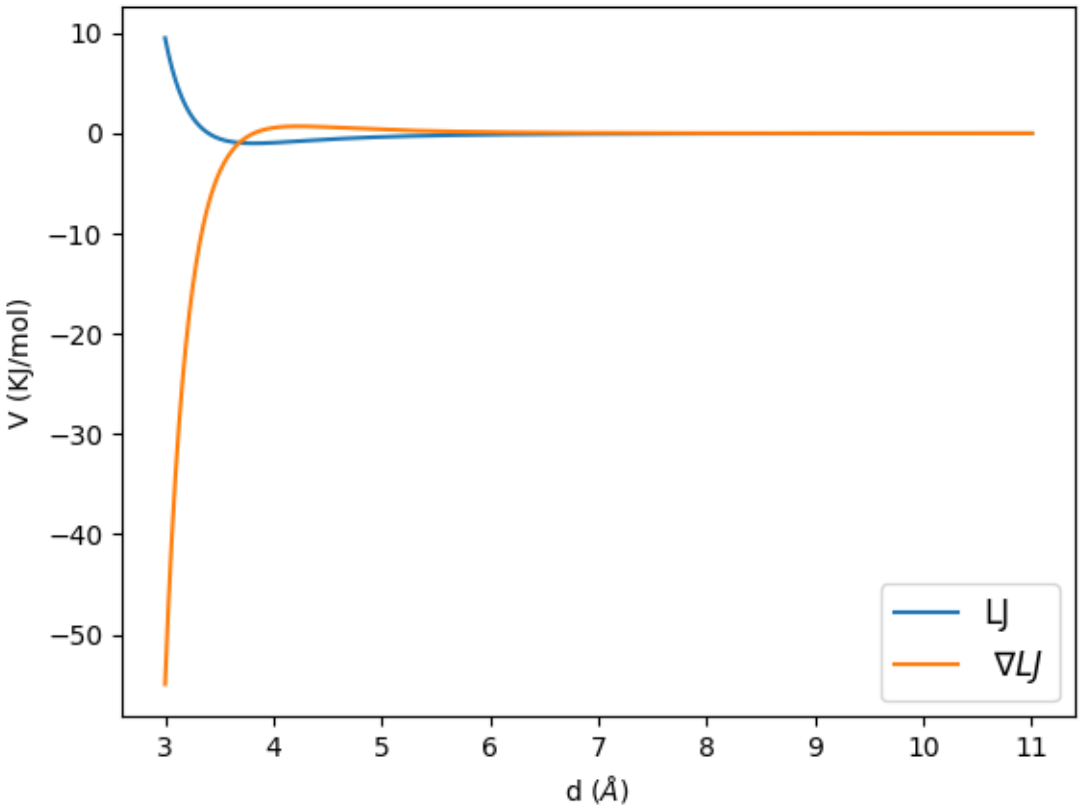


Figure 2: Potential and gradient for 2 particles

We can see that the gradient is 0 at the point that corresponds to the minimum in the potential.

f)

```
def linesearch(F, X0, d, alpha_max, tolerance, max_iterations):
  def f(alpha):
    aux= d*F(X0+alpha*d)
    l=np.sum(aux)
    l2= np.sum(abs(aux))
    return(l)
  alpha, n_iteractions = bisection_root(f,0,alpha_max,tolerance= 1e-13)
  return(alpha, n_iteractions)
```

I tested my algorithm on $\vec{X_0}$ and found that the α that minimizes $V_{LJ}(\vec{X_0} + \alpha\vec{d})$ is $\alpha = 0.4517$

2

g)

```
def golden_section_min(f,a,b,tolerance=1e-3):
```

```
  tau=(np.sqrt(5)-1)/2
  x1=a + (1-tau)*(b-a)
  f1=f(x1)
  x2=a+ tau*(b-a)
  f2=f(x2)
  while(abs(a-b)>tolerance):
    if(f1>f2):
      a=x1
      x1=x2
      f1=f2
      x2=a+tau*(b-a)
      f2=f(x2)
    else:
      b=x2
      x2=x1
      f2=f1
      x1=a+(1-tau)*(b-a)
      f1=f(x1)

  return(x2)
```

With this algorithm I found an $\alpha = 0.452$ which is quite similar to the one found with line_search (the difference is acceptable in the range of tolerance).

The optimal distance between two Ar atoms using this method is $r_0 = 3.817$ Å

2.1 h) and i)

```
def BFGS(f, gradf, X0, linesearch, tolerance=1e-9, max_iterations=10000):
  X=np.array(np.copy(X0))
  f=flattenfunction(f); gradf=flattengradient(gradf)
  B= np.identity(len(X))*np.linalg.norm(gradf(X))/0.01
  N_calls=0
  converged=True
  y=300
  while(abs(np.linalg.norm(y))>tolerance):
    s= np.linalg.inv(B)@((-gradf(X)))
    alpha=1
    if(linesearch==True):
      def f1d(alpha):
        return(f(X+alpha*s))
      alpha= golden_section_min(f1d, -1., 1.)
    s*=alpha
    X_opt= np.copy(X);
    X+= s
    y=np.array([gradf(X)-gradf(X_opt)])
    B= B+ (np.outer(y,y)/np.dot(y,s) - np.outer(B@s, B@s)/np.dot(s, B@s))
    N_calls+=1
    if(N_calls==max_iterations):
      converged=False
      break
  return(X_opt, N_calls, converged)
```

I included the improvement introduced in i) as an option in the algorithm of h). If in the parameters of the function we set linsearch=False the system will perform a full step Δx. Otherwise, after each calculation we will find an alpha that optimises the step. This α can also be negative in order to correct

an overshooting (in fact my implementation works way better with $\alpha \in [-1, 1]$.

With this algorithm and without the linesearch feature I was able to obtain the same optimal distance for a system of two Ar atoms as before, i.e $r_0 = 3.817$ Å and 6 iterations. The system with N=3 also converges after 35 iterations to a good optimum triangle with all the particles lying withing 1% of the two-particles optimum distance.

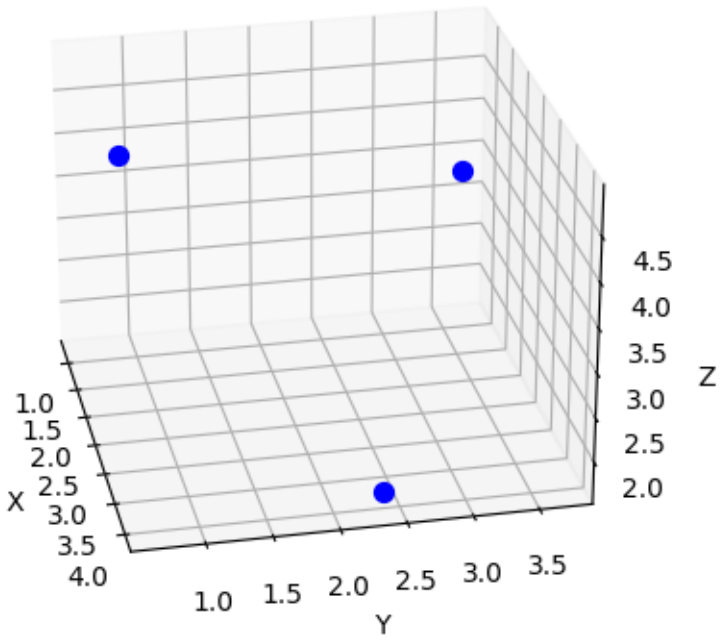


Figure 3: Molecule configuration with N=3 atoms of Argon

For N=4 and higher values of N I wasn't able to obtain good results. My systems converge to local minima but this is not the searched one and the configuration could really be improved. However, with linesearch I was able to find all the optimal configurations.

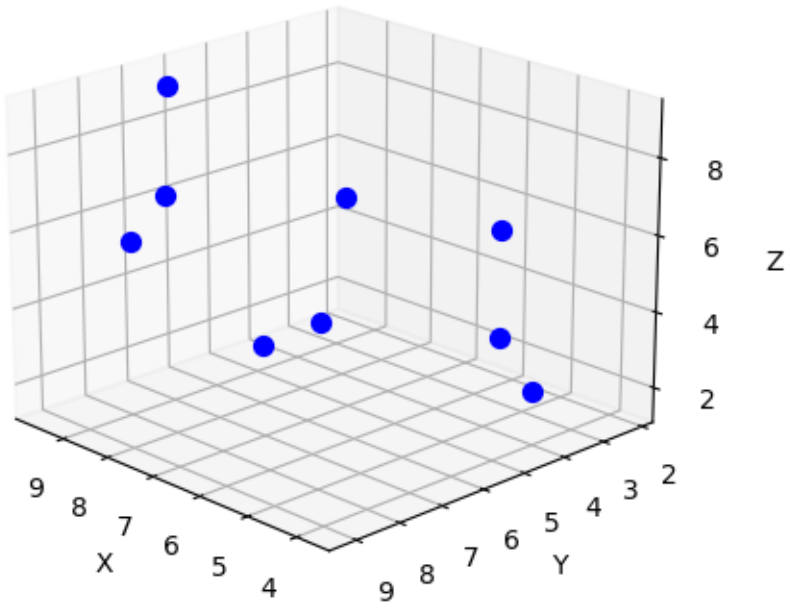


Figure 4: Molecule configuration with N=9 atoms of Argon

In the case of N=9, I obtained a minimum potential $V_{LJ} = -22.015\, KJ/mol$ which is quite near the actual potential ($\approx$ -24 KJ/mol). In this case we obtained 19 Van der Waals bonds.

For N=3, adding the linsearch feature doesn't affect the number of steps and for N=9 it does though. This is quite expected since linesearch actually makes the steps lower than the original ones so it takes longer to converge to the minimum.

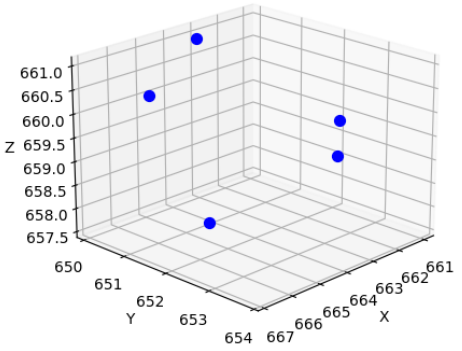
j)

```
280 def simulated_anealing(f, X0):
281   alpha=1-1e-5
282   gamma=0.5
283   x=np.array(np.copy(X0))
284   f=flattenfunction(f);
285   T=0.1
286   while(T>10**(-4)):
287     deltax=np.random.rand(len(x))
288     deltaE= gamma*(f(x+deltax)-f(x))
289     if(deltaE>0):
290       r=np.e**(-deltaE/T)
291       p=np.random.rand()
292       if(p<r):
293         x+=deltax
294     else:
```

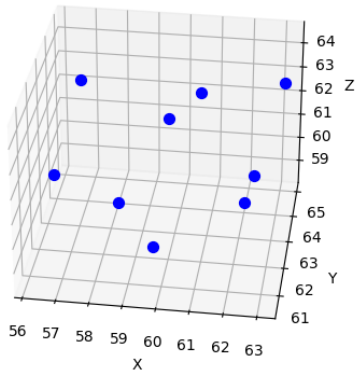
```
295         x+=deltax
296       T*=alpha
297     return(x)
```

This implementation yields decent results but worse than our customized BFGS. However, the advantage here is that I can start from a totally random configuration (as it is defined in line 255). In the case of BFGS we first needed a good starting point, i.e a initial configuration near to the real minimum. So what we can do is to combine both methods in one. Starting with a totally random configuration I used my MC function in order to find an initial guess for the optimal configuration. Then I used the output of MC as an input in BFGS to obtain the final result which is really accurate compared to the actual values.

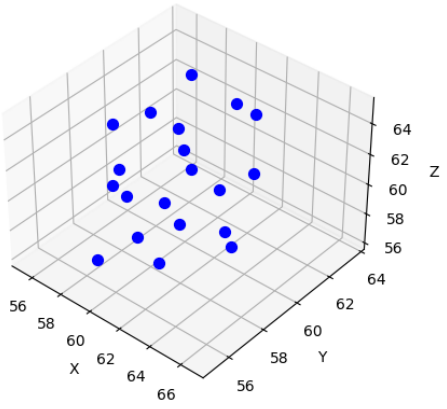
For N=5 I obtained $V_{min}^{MC} = -8.966$ and $V_{min} = -9.7065$, for N=9 I obtained $V_{min}^{MC} = -22.206$ and $V_{min} = -24.041$ and for N=20 I obtained $V_{min}^{MC} = -54.842$ and $V_{min} = -73.604$.



(a) N=5



(b) N=5



(c) N=5

Figure 5: Optimal molecule configurations obtained combining simulated anhealing with BFGS

Additionally, for N=5 we plotted the energy change after each iterations. This gives us an overview of

how does simulated anhealing yield a really good approximation starting from a very bad point:

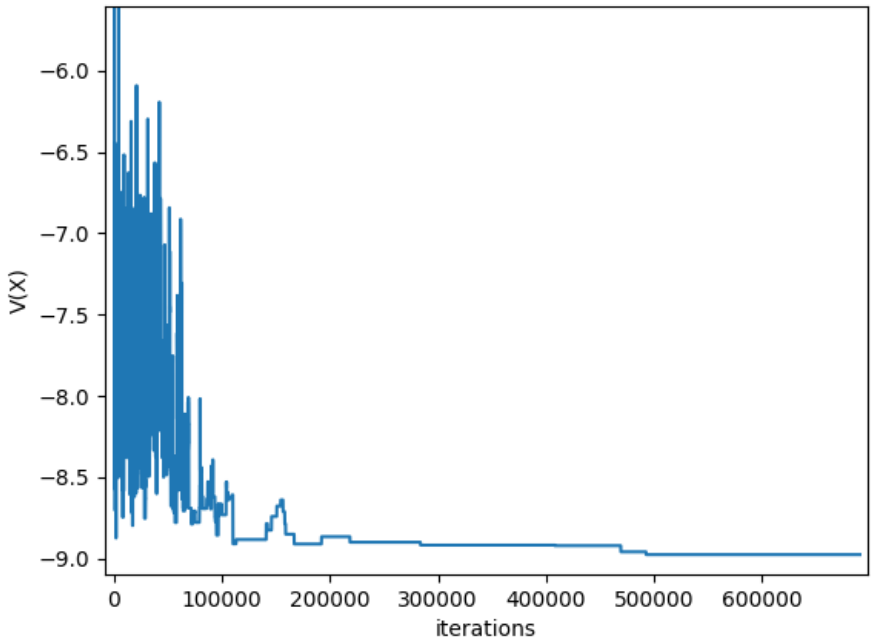


Figure 6: Energy change with iterations for simulated ANhealing

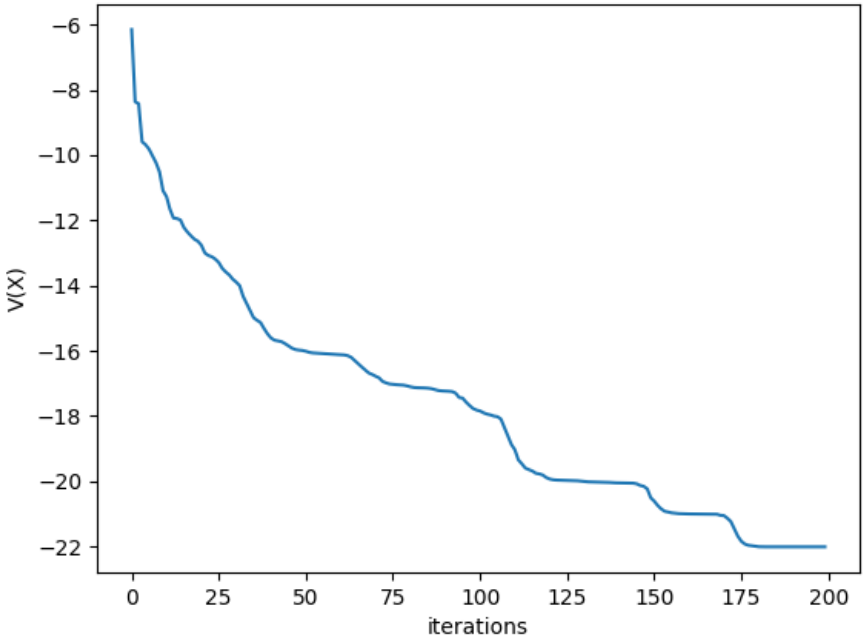


Figure 7: Energy change with iterations for BFGS