package link: https://github.com/viniciuspiccoli/simulationQP934.jl github link: https://github.com/viniciuspiccoli/QP934

Vinicius Piccoli

1 Capítulo 1

```
# Vinicius Piccoli - 24/09/2020
# exerc cios 1 e 2


# ex 1 -
  println("Ex - 1")
  a =  2.53
  b =  5.55
  c = -3.40

  println(a < b)
  println(b > c)
  println(a >= c)
  println(c <= b)


# ex 2 -
  println("Ex - 2")
  println("a" > "b")
  println("c" < "e")
  println("z" >= "w")
  println("d" <= "k" )

  println("abc" < "z")
```

[Click here to download the code]

```
# exerc cio 3

  include("./func.jl")

  for i in rand()
    println(f(i))
  end
```

[Click here to download the code]

```julia
# fun  o do exerc cio 3

  function f(x::Float64)
    return x^2 + 2*x -3
  end
```

[Click here to download the code]

Script com as demais atividades do capítulo 1.

```julia
# exerc cio 4
  function f(x::Vector)
    x[1] = x[1] + 1
    return x
  end

  function g(x::Vector)
    y = copy(x)
    y[1] = y[1] + 1
    return y
  end

# funcao 1
  x = [1,1]
  f(x)
  println(x)

# funcao 2
  y = [1,1]
  g(y)
  println(y)



###########


# exerc cio 5

  x = [2,2]
  y = copy(x)

  y[1]  = y[1] - 5

  println(x)
  println(y)



# exerc cio 6

  x = [i for i in 1:10]
  y = Vector{Int64}(undef,length(x))

  println(x)

  for i in 1:length(x)
    y[i] = x[i]
  end
```

```

y[1] = y[1] - 19
y[6] = y[6] + 5


println(x)
println(y)
```

[Click here to download the code]

2 Capítulo 2 -Simulação de partículas - métodos ingênuos

```
# exercise 7
# ep = 100 J / sig = 3.4 angs - Arg wikip dia

  function lj(d::Float64,ep, sig)
    LJ = 4 * ep * ((sig/d)^12 - (sig/d)^6)
    return LJ
  end
```

[Click here to download the code]

```
include("./ativ_7.jl")

# exercise 8, 9 and 10
 cls  = ["red", "green", "blue", "pink", "orange", "black"] # Plot
    colors
 eps  = [80, 100, 120, 140, 160, 200]                       # Well
    depth . Reference value: 120 e(J)/kb(J/K) = K
 sigs = [ 0.34, 0.36, 0.38, 0.40, 0.42, 0.44]               # sigma
    values in nm. Reference value is 0.34 nm
 dist = [i for i in 0.32:0.01:0.85]                         # Distance
    ranging from 0.32 to 0.85 nm

 using Plots, LaTeXStrings

 plot(framestyle=:box, legend=:best, grid=:false)
 plot!(xlabel=L"r\ / \mathrm{\AA}",ylabel=L"Energy\ of\ interaction\ ",
     ylims=[-130, 350])

 for i in 1:length(eps)
   energies = lj.(dist,eps[3],sigs[i])
   plot!(dist, energies, label=sigs[i], linewidth=2, color=cls[i])
 end

 annotate!( 0.70, 100, text("Ar-Ar ep=120", :center, 10))
 plot!(size=(375,375))
 savefig("LJ_Ar_potential.png")
```

[Click here to download the code]

```

# functions
  function lj(d::Float64,ep, sig)
    LJ = 4 * ep * ((sig/d)^12 - (sig/d)^6)
    return LJ
  end

  function dist(x::Vector{Float64},y::Vector{Float64})
    d = sqrt((y[1] - x[1])^2 + (y[2]-x[2])^2)
    return d
  end

# exercise 11

 # LJ potential
```

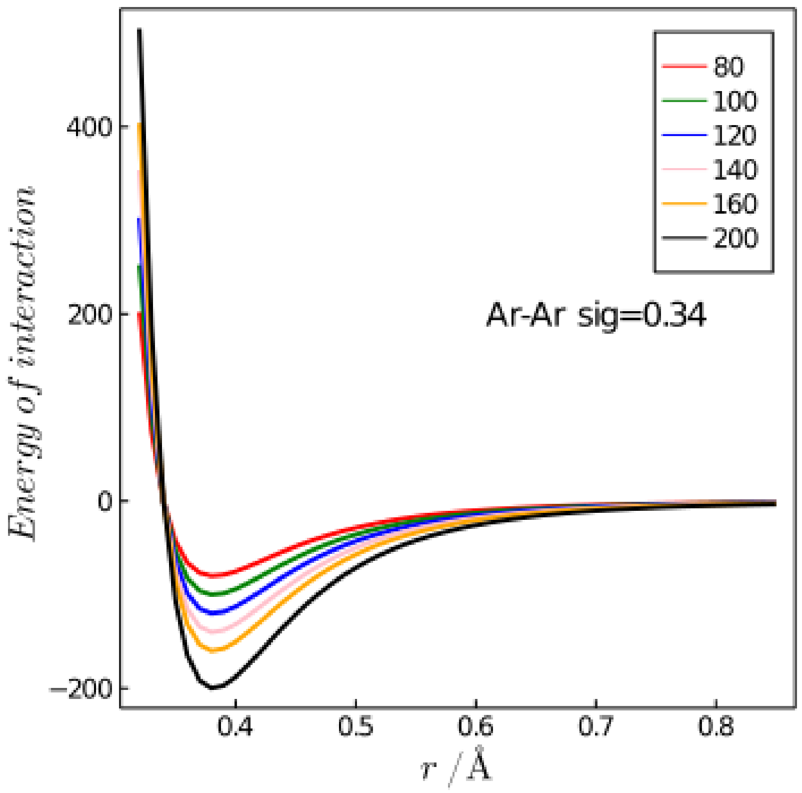


Figure 1: Curvas representando potenciais de Lennard-Jones para o argônio. É variado o valor do poço de potencial, representado pela letra grega ε.

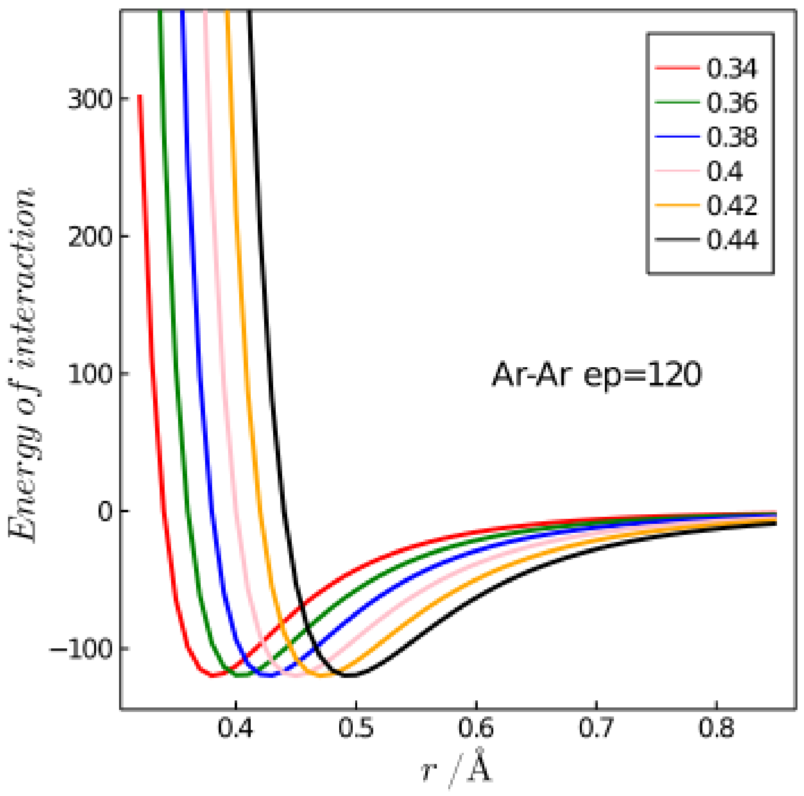


Figure 2: Curvas representando potenciais de Lennard-Jones para o argônio. É variada a distância em que o potencial é nulo, ou seja, separação mínima entre as entidades. Esta distância é dada pela letra grega σ.

```
sig = 1.
ep  = 10.

box = [10*rand(2) for i in 1:10]
# box[point][coordinates]

# function to sum energy for all pairs of atoms in a box
function box_LJ(box::Vector, sig::Float64, ep::Float64)
  Ut = 0 # energia total
  n  = 0 # n mero total de pares n o repetidos
  for i in 1:length(box)-1
    for j in i+1:length(box)
      n = n + 1
      d = dist(box[i],box[j])
      U = lj(d,ep,sig)
      Ut = Ut + U
    end
  end
  Ut = Ut / n
  return Ut, n
end

U,n = box_LJ(box,sig,ep)
```

[Click here to download the code]

```

  using BenchmarkTools

# functions
  function lj(d::Float64,ep::Float64, sig::Float64)
    LJ = 4 * ep * ((sig/d)^12 - (sig/d)^6)
    return LJ
  end

  function dist(x::Vector{Float64},y::Vector{Float64})
    d = sqrt((y[1] - x[1])^2 + (y[2]-x[2])^2)
    return d
  end

# new function pot + dist - optm

  function lj2(x::Vector{Float64},y::Vector{Float64},eps4::Float64,
    sig6::Float64,sig12::Float64)
    d = (y[1] - x[1])^2 + (y[2]-x[2])^2
    r6  = d^3
    r12 = r6^2
    LJ = eps4 * ((sig12/r12) - (sig6/r6))
    return LJ
  end

 # exercise 12

 # Func 1
 function box_LJ(box::Vector, sig::Float64, ep::Float64)
   Ut = 0. # energia total
   n  = 0 # n mero total de pares n o repetidos
   for i in 1:length(box)-1
```

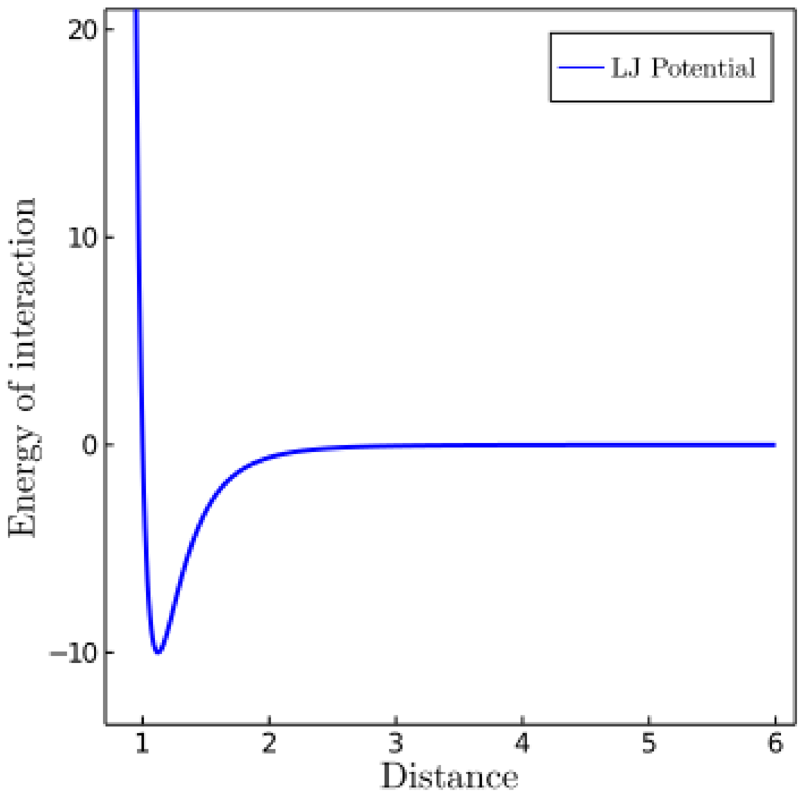


Figure 3: Potencial de Lennard-Jones com $\varepsilon = 10.0$ e $\sigma = 1$

```julia
    for j in i+1:length(box)
      n = n + 1
      d = dist(box[i],box[j])
      U = lj(d,ep,sig)
      Ut = Ut + U
    end
  end
  return Ut
end

# Func 2
function box_LJ2(box::Vector, sig::Float64, ep::Float64)
  Ut = 0
  N = length(box)
  eps4  = 4*ep
  sig6  = sig^6
  sig12 = sig6^2

  for i in 1:N-1
    for j in i+1:N
      U = lj2(box[i],box[j],eps4,sig6,sig12)
      Ut = Ut + U
    end
  end

  return Ut
end

using BenchmarkTools, Test
# LJ potential
sig = 1.
ep  = 10.

box = [10*rand(2) for i in 1:10]

@test isapprox(box_LJ(box,sig,ep), box_LJ2(box,sig,ep))
print("LJ 1"); @btime box_LJ($box,$sig,$ep)
print("LJ 2 - otimizado"); @btime box_LJ2($box,$sig,$ep)
```

[Click here to download the code]

2.0.1 Condições periódicas de contorno

```julia
# exercise 15
let
  # box
  pts = [100*rand(2) for i in 1:2]

  # distance between two potins occording to the minimal-image
    convention
  ds = sqrt((pts[1][1] - pts[2][1])^2 + (pts[1][2]-pts[2][2])^2) % 10.
  for i in 1:2
    if ds > 0.5*10
      pts[2][i] = pts[2][i] - 10
    elseif ds < -0.5*10
      pts[2][i] = pts[2][i] + 10
    end
  end
  dn = sqrt((pts[2][1] - pts[1][1])^2 + (pts[2][2]-pts[1][2])^2)

 println(dn)

end
```

[Click here to download the code]

```julia
# exercise 16

# function that will create a box with side equal to "range" and "N"
    points within.
  function point_gen(range::Int64, N::Int64)
    pts = [range*rand(2) for i in 1:N]
    return pts
  end

# Function to calculate the distance between two points
 function pbc(x::Vector{Float64},y::Vector{Float64}, L)
   ds = y - x
   for i in 1:2
     ds[i] = ds[i]%L
     if ds[i] > 0.5*L
       ds[i] = ds[i] - L
     elseif ds[i] < -0.5*L
       ds[i] = ds[i] + L
     end
   end
   return sqrt(ds[1]^2 + ds[2]^2)
 end

function r(x::Vector{Float64},y::Vector{Float64}, side)
  # dx is a vector of the difference in position of y and x
  dx = y - x
  for i in 1:2
    dx[i] = dx[i]%side
    if dx[i] > side/2
      dx[i] = dx[i] - side
    elseif  dx[i] < -side/2
      dx[i] = dx[i] + side
    end
  end
  return sqrt(dx[1]^2+dx[2]^2)
end
```

[Click here to download the code]

```julia
# exercise 17

include("./ativ_16.jl")

let

  box      = point_gen(100,100)
  pbc_side = 10
  dists    = Vector{Float64}(undef, (length(box)-1) * length(box))
  dist2    = Vector{Float64}(undef, (length(box)-1) * length(box))


  n = 0
  for i in 1:length(box)-1
    for j in i+1:length(box)
      n = n + 1
      dists[n] = r(box[i],box[j], pbc_side)
      dist2[n] = pbc(box[i],box[j], pbc_side)
    end
  end

  d_max = maximum(dists)
  d_max2= maximum(dist2)
  d_pbc = sqrt(pbc_side^2 + pbc_side^2) / 2

  println("Max distance found: $d_max")
  println("Max distance found2: $d_max2")
  println("Max distance possible: $d_pbc")

end
```

[Click here to download the code]

```julia
include("./ativ_16.jl")

# exercise 18

 function lj(d::Float64,ep, sig)
   LJ = 4 * ep * ((sig/d)^12 - (sig/d)^6)
   return LJ
 end

 box = point_gen(100,100)

 for i in 1:length(box)
   for j in i+1:length(box)
     println("Pair : $i - $j")
     dn = dist(box[i],box[j])
     d = pbcseparation(box[i],box[j],10)
     U = lj(d,10,5)
     println(" Distance    = ",dn)
     println(" Distance MI = ",d)
     println(" Energy of interaction = ",U)
```

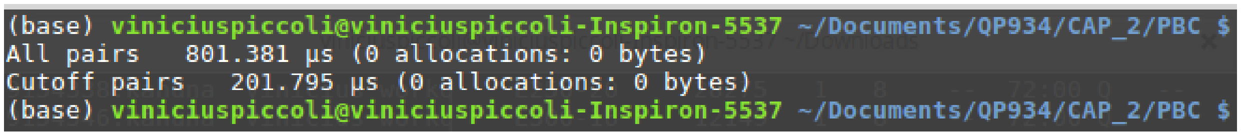
```
  end
end
```

[Click here to download the code]

```
# exercise 19 -  21

# functions

# function that will create a box with side equal to "range" and "N"
    points.
function point_gen(range::Int64, N::Int64)
  pts = [range*rand(2) for i in 1:N]
  return pts
end

# function to calculate the distance between two points according to
    the minimal-distance convention
function pbcseparation(x::Vector{Float64},y::Vector{Float64}, L::Int64
    )
  for i in 1:2
    ds = (y[i] - x[i])%L
    if ds > 0.5*L
      y[i] = y[i] - L
    elseif ds < -0.5*L
      y[i] = y[i] + L
    end
  end
  dn = sqrt((y[1] - x[1])^2 + (y[2]-x[2])^2)
  return dn
end

# Calculation of the distance between two points
function dist(x::Vector{Float64},y::Vector{Float64})
  d = sqrt((y[1] - x[1])^2 + (y[2]-x[2])^2)
  return d
end

# Lennard-Jones potential
function lj(d::Float64,ep, sig)
  LJ = 4 * ep * ((sig/d)^12 - (sig/d)^6)
  return LJ
end

#
    ###############################################################################


# first program - calculating the LJ pot of all pairs
function all_pairs(box,e,sig)
side = 10
Ut = 0.
for i in 1:length(box)
  for j in i+1:length(box)
    U = lj(pbcseparation(box[i],box[j],side),e,sig)
    Ut = Ut + U
  end
end
```

```julia
 return Ut
end

# Program that calculates the LJ pot only if the distance between particles is smaller than the cutoff.
function cutoff_pairs(box,e,sig,cutoff)
 side = 10
 Ut = 0.
 for i in 1:length(box)
   for j in i+1:length(box)
     d  = pbcseparation(box[i],box[j],side)
     if d <= cutoff
       Ut  += lj(d,e,sig)
     end
   end
 end
 return Ut
end


# box with 100 points with coordinates ranging from 0 to 10

 box = point_gen(10,100);
 cutoff = 2.
 e = 2.
 sig = 0.5

 using BenchmarkTools, Test
 @test all_pairs(box,e,sig)     cutoff_pairs(box,e,sig,cutoff)

 print("All pairs ");        @btime    all_pairs($box,$e,$sig)
 print("Cutoff pairs ");     @btime    cutoff_pairs($box,$e,$sig,$cutoff)

# println("result all pairs            = ", all_pairs(box,e,sig))
# println("result with cutoff          = ", cutoff_pairs(box,e,sig,cutoff))
```

[Click here to download the code]



```
(base) viniciuspiccoli@viniciuspiccoli-Inspiron-5537 ~/Documents/QP934/CAP_2/PBC $
All pairs   801.381 μs (0 allocations: 0 bytes)
Cutoff pairs   201.795 μs (0 allocations: 0 bytes)
(base) viniciuspiccoli@viniciuspiccoli-Inspiron-5537 ~/Documents/QP934/CAP_2/PBC $
```

Figure 4: Velocidade de execução dos códigos.

2.1 Método das células ligadas

```julia
 include("./funcoes_otm.jl")
 using BenchmarkTools


 ### exercise 25

 # function - ijcell
 ijcell(pair, cutoff) = [trunc(Int64,pair[1]/cutoff) + 1 , trunc(Int64
   , pair[2]/cutoff) + 1]


#  # function - class
#  function class(pts, cutoff, side)
#    ncellsx = trunc(Int64,side/cutoff) + 1 # we are going to start
   from 1
#    ncellsy = trunc(Int64,side/cutoff) + 1
#
#    list    = zeros()
#    for i in 1:length(p)
#      push!(,i)
#
#    end
#
#  end
# test of execution
#  N  = collect(100:50:5000);
#
#  for i in N
#
#    box = point_gen(12,i)
#    print("Time of exec with $i particles = "); @btime f($box)
#
#  end


 function f(vec;cutoff=3.)
   cls = Vector{Vector{Int64}}(undef,length(vec))
   for i in 1:length(vec)
     x = ijcell(vec[i],cutoff)
     cls[i] = x
   end
   return cls
 end

 box = point_gen(12, 10000)
 cls = f(box)

#  println(cls)
```

[Click here to download the code]

```julia
# exercise 26
 include("ativ_25.jl")

 box   = point_gen(12,50) # 50 points with coordinates from 0 to 12
 cells = f(box)           # vector that relates particle with its cell


 # function to classify
 function assign(data)
   pts = maximum(data)
   M   = [ [] for i in 1:pts[1], j in 1:pts[2] ] # Array to save
   particles in a cell
```

```
    N   = zeros(Int64,pts[1],pts[1])  # Array to save the number of
    particles in each cell

    # saving the particles that are inside of the same cell
    for i in 1:length(data)
      var = data[i]
      push!(M[var[1],var[2]],i)
    end

    # couting the number of particles in each cell
    for i in 1:pts[1], j in 1:pts[2]
      N[i,j] = count(i->(i!=0), M[i,j])
    end

   return M, N

  end


  pts, ns =  assign(cells)
```

[Click here to download the code]

```
# exercise 28

  # using push
  function assign(data)
    pts = maximum(data)
    M   = [ Int[] for i in 1:pts[1], j in 1:pts[2] ] # Array to save
    particles in a cell
    N   = zeros(Int64,pts[1],pts[1])  # Array to save the number of
    particles in each cell

    # saving the particles that are inside of the same cell
    for i in 1:length(data)
      var = data[i]
      push!(M[var[1],var[2]],i)
    end

    # couting the number of particles in each cell
    for i in 1:pts[1], j in 1:pts[2]
      N[i,j] = count(i->(i!=0), M[i,j])
    end

    return M, N
  end


  # using pre-allocated vectors
  function assign_vec(data)
    pts = maximum(data)

    M = zeros(Int64, pts[1], pts[2],20) # 10 is the maximum number of
    positions for each cell
    N = zeros(Int64, pts[1], pts[2])

    for i in 1:length(data)
      icell = data[i][1]
```

```julia
      jcell = data[i][2]
      N[icell,jcell] = N[icell,jcell] + 1
      M[icell,jcell,N[icell, jcell]] = i
    end

    return M, N
  end


# testing
 using BenchmarkTools
 include("./ativ_25.jl")

 # assign_vec alt
 pts = maximum(cls)

 list  = zeros(Int64, pts[1], pts[2],20) # 10 is the maximum number of
    positions for each cell
 nlist = zeros(Int64, pts[1], pts[2])

 function assign_vec!(data,M,N)

   for i in 1:length(data)
     icell = data[i][1]
     jcell = data[i][2]
     N[icell,jcell] = N[icell,jcell] + 1
     M[icell,jcell,N[icell, jcell]] = i
   end

   return M, N
 end

 println("Bechmarks")
 print("execution time (push)  = ") ; @btime assign($cls)
 print("execution time (alloc)  = "); @btime assign_vec($cls)
 print("execution time (alloc2) = "); @btime assign_vec!($cls,list,
    nlist) setup=(list=zeros(Int64, pts[1], pts[2],20); nlist=zeros(
    Int64, pts[1], pts[2]))  evals=1
```

[Click here to download the code]

```julia
  # exercise 29

  include("./funcoes_otm.jl")

  box = point_gen(15,50)
  cls = f(box)

  N = length(box);

  # creating vectors
  pts = maximum(cls) # max number of i and j
  list  = zeros(Int64, pts[1], pts[2],N) # 20 - maximum number of cell
    positions
  nlist = zeros(Int64, pts[1], pts[2])

  # function to save particles in each cell
  function assign!(data, M, N)
  for i in 1:length(data)
    icell = data[i][1]
    jcell = data[i][2]
    N[icell,jcell] = N[icell,jcell] + 1
    M[icell,jcell,N[icell, jcell]] = i
  end
end



function mindist(p,cellparticles,cutoff)
  np = length(p) # Number of particles
  nc = size(cellparticles,1) # Dimension of the grid
  d = +Inf
  U = 0
  for ip in 1:np
    icell = trunc(Int64,p[ip][1]/cutoff)+1
    jcell = trunc(Int64,p[ip][2]/cutoff)+1
    for i in icell-1:icell+1
      if ( i < 1 || i > nc ) continue end # Border
      for j in jcell-1:jcell+1
        if ( j < 1 || j > nc ) continue end # Border
        # Loop over the particles of this cell
        for jp in cellparticles[i,j,:]
          if jp > ip # Skip repeated
            # Compute distance and keep minimum
            d = min(d,sqrt( (p[ip][1]-p[jp][1])^2 + (p[ip][2]-p[jp
  ][2])^2) )
            U = U + lj(d,5,2)
          end
        end
      end
    end
  end
  return U
end


# using push
function assign(data)
  pts = maximum(data)
  M   = [ Int[] for i in 1:pts[1], j in 1:pts[2] ]
  N   = zeros(Int64,pts[1],pts[1])

  # saving the particles that are inside of the same cell
  for i in 1:length(data)
    var = data[i]
    push!(M[var[1],var[2]],i)
  end

  # couting the number of particles in each cell
  for i in 1:pts[1], j in 1:pts[2]
    N[i,j] = count(i->(i!=0), M[i,j])
  end

  return M, N
end
```

```julia


 function mindist2(p,cellparticles,cutoff)
   np = length(p) # Number of particles
   nc = size(cellparticles,1) # Dimension of the grid
   d = +Inf
   U = 0
   for ip in 1:np
     icell = trunc(Int64,p[ip][1]/cutoff)+1
     jcell = trunc(Int64,p[ip][2]/cutoff)+1
     for i in icell-1:icell+1
       if ( i < 1 || i > nc ) continue end # Border
       for j in jcell-1:jcell+1
         if ( j < 1 || j > nc ) continue end # Border
         # Loop over the particles of this cell
         for jp in cellparticles[i,j]
           if jp > ip # Skip repeated
             # Compute distance and keep minimum
             d = min(d,sqrt( (p[ip][1]-p[jp][1])^2 + (p[ip][2]-p[jp
    ][2])^2) )
             U = U + lj(d,5,2)
           end
         end
       end
     end
   end
   return U
 end

 using BenchmarkTools
 pts = maximum(cls)

#print("assign alloc")#; @btime assign!($cls,list,nlist) setup=(list=
    zeros(Int64, pts[1], pts[2],N); nlist=zeros(Int64, pts[1], pts[2]))
      evals=1
;
#print("assign push")#; @btime  assign($cls)


#print("TIme of ex for alloc = ")#; @btime mindist($box,$list,3.)
#print("Time of ex for push  = ")#; @btime mindist2($box,$plist,3.)


assign!(cls,list,nlist) ;
plist, pnlist = assign(cls) ;

U  = mindist(box,list,3.)
println(U)
U2 = mindist2(box,plist,3.)
println(U2)

# exercise 30
```

[Click here to download the code]

```julia
# exercise 30

function wrap_cell(dms,id,jd)
```

```julia
  i = dms[1]
  j = dms[2]

  if id < 1
    id = id  + i
  elseif id > i
    id = id - i
  end
  if jd < 1
    jd = jd + j
  elseif jd > j
    jd = jd - j
  end

 return id, jd
end



function check_wrap(nc)
  list = Matrix{Int}(undef,0,4)
  for i in 1:nc[1]
    for j in 1:nc[2]
      for ic in i-1:i+1
        for jc in j-1:j+1
          iw, jw = wrap_cell(nc,ic,jc)
          list = vcat(list,[ic jc iw jw])
        end
      end
    end
  end
  return list
end


nc = [3,3]
list = check_wrap(nc)
for i in 1:length(list)
  println(list[i])
end
```

[Click here to download the code]

```julia
import Random

# Box
function point_gen(range::Int64, N::Int64)
  Random.seed!(321)
  pts = [range*rand(2) for i in 1:N]
  return pts
end

function cutoff_pairs(box,e,sig,cutoff;side=10)
  Ut = 0.
  d  = +Inf
  for i in 1:length(box)
    for j in i+1:length(box)
      d = pbcseparation(box[i],box[j],side)
      if d <= cutoff
        Ut  = Ut +  lj(d,e,sig)
      end
    end
  end
  return Ut
end

# box with 10000 particles with coordinates ranging from 0 to 100

box = point_gen(100,10000);
cutoff = 2.
e = 2.
sig = 0.5

A = cutoff_pairs(box,e,sig,cutoff)
println(A)
```

[Click here to download the code]

```
# exercise 32
 include("funcoes_otm.jl")

# box with 10000 particles with coordinates ranging from 0 to 100

box = point_gen(100,10000);
side = 100
cutoff = 10.

e = 2.
sig = 0.5

cls = f(box,cutoff)
list = assign(cls);

list2 = assign_data(box, side, cutoff)

using BenchmarkTools, Test

print("Cutoff pairs ");     @btime    cutoff_pairs($box,$e,$sig,$cutoff,$side)
print("Linked cells");      @btime    pot_ener($box,$list, $cutoff,$e,$sig,$side)

A = cutoff_pairs(box,e,sig,cutoff,side)
println(A)

B = pot_ener(box,list,cutoff,e,sig,side)
println(B)
```

[Click here to download the code]

2.2 Listas ligadas

```
 include("../CELULA_LIGADA/funcoes_otm.jl")

 box = point_gen(12,10)
 cutoff = 3
 side = 12
```

```julia

  Nc = trunc(Int64 ,side / cutoff)  # Number of cells
  N  = length(box)                          # Number of particles

  # stating linked lists
  first_atom = zeros(Int64, Nc, Nc)

  #first_atom = [ Int64[] for i in 1:Nc, j in 1:Nc ]
  next_atom  = zeros(Int64, N)


#  for iat in 1:N
#
#    icell = trunc(Int64, box[iat][1]/cutoff) + 1
#    jcell = trunc(Int64, box[iat][2]/cutoff) + 1
#
#    next_atom[iat] = first_atom[icell, jcell]
#
#    first_atom[icell, jcell] = iat
#
#
#  end


  # function
  function linkedlist!(box, side, cutoff, first_atom, next_atom)

    for iat in 1:N

      icell = trunc(Int64, box[iat][1]/cutoff) + 1
      jcell = trunc(Int64, box[iat][2]/cutoff) + 1

      next_atom[iat] = first_atom[icell, jcell]

      first_atom[icell, jcell] = iat

    end
  end

  linkedlist!(box,side,cutoff,first_atom, next_atom)
```

[Click here to download the code]

```julia
  include("../CELULA_LIGADA/funcoes_otm.jl")

  # box, cutoff and side of the system
  box = point_gen(12,1000)
  cutoff = 3
  side = 12

  # for push method
  cls=f(box,cutoff)

  #linked list parameters
 # Nc = trunc(Int64 ,side / cutoff)  # Number of cells
 # N  = length(box)                          # Number of particles

 # first_atom = zeros(Int64, Nc, Nc) # stating linked lists
 # next_atom  = zeros(Int64, N)      # next atoms

  using BenchmarkTools

  print("Performance of the push method") ;@btime assign_data($box,$
    side,$cutoff)
  print("My push method") ;@btime assign($cls)
  print("Performance of the linked lisrt method") ;@btime linkedlist!($
    box,$side,$cutoff) # , $first_atom,$next_atom)
```

[Click here to download the code]

```julia
# exercise 36
function energy(box,cutoff, first_atom, next_atom,ep,sig,side)

  vec = [length(first_atom[:,1]), length(first_atom[1,:])]          #
    vector with maximum dimensions
  N   = length(box)                                                   #
     number of atoms
  #d   = +Inf
  Ut  = 0.

  for iat in 1:N
    icell = trunc(Int64,box[iat][1]/cutoff) + 1
    jcell = trunc(Int64,box[iat][2]/cutoff) + 1

    for i in icell-1:icell+1
      for j in jcell-1:jcell+1
        iw, jw = wrap_cell(vec,i, j)
        jat = first_atom[iw,jw]

        while jat > 0

          # Compute distance and keep minimum
          #d = min(d,sqrt( (box[iat][1]-box[jat][1])^2 + (box[iat][2]-
    box[jat][2])^2) )
          #d = (box[iat][1]-box[jat][1])^2 + (box[iat][2]-box[jat][2])
    ^2
          d = pbcseparation(box[iat], box[jat],side)

          #if d==0
          # println("fdp")
          # println(iat, "  ", jat)
          #end

          if jat > iat
            Ut = Ut + U(d,ep,sig)
          end
          jat = next_atom[jat]

        end
      end
    end
  end

  return Ut
end

```

```julia
function energy(box,cutoff,ep,sig,side)

  first_atom, next_atom = linkedlist(box, side, cutoff)

  vec = [length(first_atom[:,1]), length(first_atom[1,:])]          # vector with maximum dimensions
  N   = length(box)                                                  # number of atoms
  #d   = +Inf
  Ut  = 0.

  for iat in 1:N
    icell = trunc(Int64,box[iat][1]/cutoff) + 1
    jcell = trunc(Int64,box[iat][2]/cutoff) + 1

    for i in icell-1:icell+1
      for j in jcell-1:jcell+1
        iw, jw = wrap_cell(vec,i, j)
        jat = first_atom[iw,jw]

        while jat > 0

          # Compute distance and keep minimum
          #d = min(d,sqrt( (box[iat][1]-box[jat][1])^2 + (box[iat][2]-box[jat][2])^2) )
          #d = (box[iat][1]-box[jat][1])^2 + (box[iat][2]-box[jat][2])^2

          #if d==0
          # println("fdp")
          # println(iat, "  ", jat)
          #end

          if jat > iat
            d = pbcseparation(box[iat], box[jat],side)
            Ut = Ut + U(d,ep,sig)
          end
          jat = next_atom[jat]

        end
      end
    end
  end

  return Ut
end




# exercise 37
 include("../funcoes_otm.jl")
 include("../initial-point.jl")


 side = 100
 cutoff = 2
 e = 5.0
 sig = 0.5
 box = initial_point(10_000,side,0.9)


# import Random
# Random.seed!(321)
# box = [ 100*rand(2) for i in 1:10_000 ]
#
# side = 100
# cutoff = 10.
# e = 2.
# sig = 0.5


 using BenchmarkTools, Test

# println("Bechmarks for the lists gen")
# print("Time to create lists using linkedlists method") ;@btime linkedlist($box,$side, $cutoff)
# print("Time to create a matrix using push method")     ;@btime assign_data($box, $side, $cutoff)



 fatm, natm = linkedlist(box,side, cutoff)
 list2 = assign_data(box, side, cutoff) # linked cells direct

  N   = 10000 # number of molecules
  vec = [length(fatm[:,1]), length(fatm[1,:])] # maximum number of cells


#print("naive method") ; @btime cutoff_pairs($box,$e,$sig,$cutoff,$side)
#Uz = cutoff_pairs(box,e,sig,cutoff,side)
#println(Uz)
#println("  ")
#
#print("Linked cells method") ;@btime pot_ener($box,$list2,$cutoff,$e,$sig,$side)
#Ut = pot_ener(box,list2,cutoff,e,sig,side)
#println(Ut)
#println("   ")

print("Linked lists method") ;@btime energy($box,$cutoff, $fatm, $natm,$e,$sig,$side)
U1=energy(box,cutoff, fatm, natm,e,sig,side,vec,N)  #,N,vec
println(U1)
#println("  ")

#print("Linked listsi (2) method") ;@btime energy($box,$cutoff,$e,$sig,$side)
#U2=energy(box,cutoff,e,sig,side)
#println(U2)
#println("  ")
```

[Click here to download the code]

```julia
include("../initial-point.jl")

#box = (10000,100,0.9)


import Random
Random.seed!(321)
box = [ 100*rand(2) for i in 1:10_000 ]

side = 100
cutoff = 10.
#e = 2.
#sig = 0.5

function celllist1(p,side,cutoff)
  n = round(Int64,side/cutoff)
  cellparticles = [ Int64[] for i in 1:n, j in 1:n ]

  for i in 1:length(p)
    icell = trunc(Int64,p[i][1]/cutoff) + 1
    jcell = trunc(Int64,p[i][2]/cutoff) + 1
    push!(cellparticles[icell,jcell],i)
  end
    return cellparticles
end

function celllist2(p,side,cutoff,nmax)
  n = round(Int64,side/cutoff)
  nparticles = zeros(Int64,n,n)
  cellparticles = Array{Int64}(undef,n,n,nmax)
  for i in 1:length(p)
    icell = trunc(Int64,p[i][1]/cutoff) + 1
    jcell = trunc(Int64,p[i][2]/cutoff) + 1
    nparticles[icell,jcell] += 1
    if nparticles[icell,jcell] > nmax
      error("n > nmax")
    end
    cellparticles[icell,jcell,nparticles[icell,jcell]] = i
  end
  return cellparticles
end

function celllist3(box, side, cutoff)  #  , first_atom, next_atom
  Nc = trunc(Int64 ,side / cutoff)  # Number of cells
  N  = length(box)                          # Number of particles
  first_atom = zeros(Int64,Nc, Nc)
  next_atom  = zeros(Int64, N)

  for iat in 1:N
    icell = trunc(Int64, box[iat][1]/cutoff) + 1
    jcell = trunc(Int64, box[iat][2]/cutoff) + 1
    next_atom[iat] = first_atom[icell, jcell]
    first_atom[icell, jcell] = iat
  end

  return first_atom, next_atom
end
```

```julia

  println("Benchmarks")

  using BenchmarkTools

  print("celllist 1") ;@btime celllist1($box,$side,$cutoff)
  print("celllist 2") ;@btime celllist2($box,$side,$cutoff,10000)
  print("celllist 3") ;@btime celllist3($box,$side,$cutoff)
```

[Click here to download the code]

3 Cálculo das forças

```julia
# function to calcule the force between a pair of particles
function fpair(x,y,r,data)

  # r values for F calculation
  r6  = r^6
  r7  = r6 * r
  r12 = r6^2
  r13 = r12 * r

  # x and y components
  # Fz = -( dU(r) / dz) = - (dr/dz) * (dU(r)/dr), r = sqrt(dx^2,dy^2)
  # calculation for -(dr/dr)

  drdx1 = -(x[1]-y[1])/r
  drdx2 = -(x[2]-y[2])/r

  # calculation for (dU(r)/dr)
  dudr1 =  -(data.rep / r13)
  dudr2 =  -(data.att / r7)


  dfdr = - (dudr1 - dudr2)

  # calculatons of force components
  f1   =   dfdr * drdx1
  f2   =   dfdr * drdx2

  # or

#  f1 =   ((data.rep / r13) - (data.att / r7))*drdx1
#  f2 =   ((data.rep / r13) - (data.att / r7))*drdx2

  f = (-f1,-f2)

  # calculation of energy of interaction
  upair = data.eps4*(data.sig12/r12 - data.sig6/r6)


  return upair, f

end



```

```

function forcepair(x,y,r,data)
  r6 = r^6
  r12 = r6^2
  r7 = r6*r
  r13 = r12*r

  drdx1 = -(x[1]-y[1])/r
  drdx2 = -(x[2]-y[2])/r

  sigfac1 = -12*(data.sig12/r13)
  sigfac2 = -6*(data.sig6/r7)

  dfacdr = -data.eps4*(sigfac1 - sigfac2)

  upair = data.eps4*(data.sig12/r12 - data.sig6/r6)
  fx = ( -dfacdr*drdx1, -dfacdr*drdx2 )

  return upair, fx
end
export forcepair
```

[Click here to download the code]

```
# force calculation using linked lists method


#prreciso acertar isso daqui


# f = [ [0. , 0.] for i in 1:data.N ]
 fvec(data) = [ Vector{Float64}(undef,2) for i in 1:data.N ]

function force!(box, DATA, frc, first_atom, next_atom, nc)
  ut = 0.

  for i in 1:length(frc)
    frc[i][1] = 0.
    frc[i][2] = 0.
  end

  for iat in 1:DATA.N
    icell = trunc(Int64,box[iat][1]/DATA.cutoff) + 1
    jcell = trunc(Int64,box[iat][2]/DATA.cutoff) + 1
    for i in icell-1:icell+1
      for j in jcell-1:jcell+1
        iw, jw = wrapcell(nc, i, j)
        jat = first_atom[iw,jw]
        while jat > 0
          if jat > iat
            rij = pbcseparation(box[jat],box[iat],DATA.side)
            if rij <= 2.
              up,fp = fpair(box[jat], box[iat], rij, DATA)
              frc[iat] .= frc[iat] .+ fp
              frc[jat] .= frc[jat] .- fp
              ut += up
            end
          end
```

```julia
          jat = next_atom[jat]
        end
      end
    end
  end
  return ut
end

export fvec, force!
```

[Click here to download the code]

```julia
using simulationQP934, Test


data = Data();
box = initial_point(data);
nc, fatm, natm = linkedlist(box,data);
force_vec = fvec(data)


utotal = force!(box, data, force_vec, fatm, natm, nc)

# test if the force is zero
vecx = 0.
vecy = 0.

for i in 1:length(force_vec)
  vecx = vecx + force_vec[i][1]
  vecy = vecy + force_vec[i][2]
end


@test vecx \approx 0.
@test vecy \approx 0.
```

[Click here to download the code]

```julia
function initial_velocity(N::Int64;T=298.15,R=8.3145,m=1.0)
  vel = [ [0.,0.] for i in 1:N]
  angle = 2*pi*rand()

  for i in 1:N
    px = rand()
    py = rand()
    vel[i][1] = sqrt(-(2*R*T/m)*log(1-px)) * angle       # velocity in
    the x axis
    vel[i][2] = sqrt(-(2*R*T/m)*log(1-py)) * angle       # velocity in
    the y axis
  end

  return vel
end

function norm(x::Vector{Float64})
  s = 0.
  for i in 1:length(x)
    s = s + x[i]^2
  end
  sqrt(s)
end

function normVEL(x)
  s = zeros(length(x))
  for i in 1:length(x)
    s[i] = norm(x[i])
  end
  return s
end

#using Plots


# p(v)dv

p(v;m=1.,R=8.3145,T=298.15) = m/(R*T) * v * exp(-m*v^2/(2R*T))

# probability v < x
pv(x;m = 1., T = 298.15, R = 8.3145) = -exp(-m*x^2/(2*R*T)) + 1




#

#vels = initial_velocity(15000)

#histogram(p.(vels),vels)
```

[Click here to download the code]

```

function md(data::Data,MDinputs::MDinput)

  @unpack mass   = data
  @unpack dt, nsteps, total_time, iprint = MDinputs

  box            = initial_point(data);
  nc, fatm, natm = linkedlist(box,data);
  force_vec      = fvec(data)
  vel            = initial_velocity(data.N);
  times = 0.

  for i in 1:nsteps
    ut = force!(box, data, force_vec, fatm, natm, nc)
    times = times + dt

    for ip in 1:data.N
      # positions update @. box[ip] = box[ip] + vel[ip]*dt + force_vec[
    ip] * (dt*dt)/(2*mass)
      @. box[ip] = box[ip] + vel[ip]*dt + force_vec[ip] * (dt*dt)/(2*
    mass)
      # velocity update
      @. vel[ip] = vel[ip] + force_vec[ip] * dt/mass
    end

    if i%iprint ==0
```

```
      println("Step #$i")
      println("Total energy = ",ut)
    end

    #if i%iprint == 0
    #  println("Total energy at $i = ",ut)
    #end

  end
end
```

[Click here to download the code]

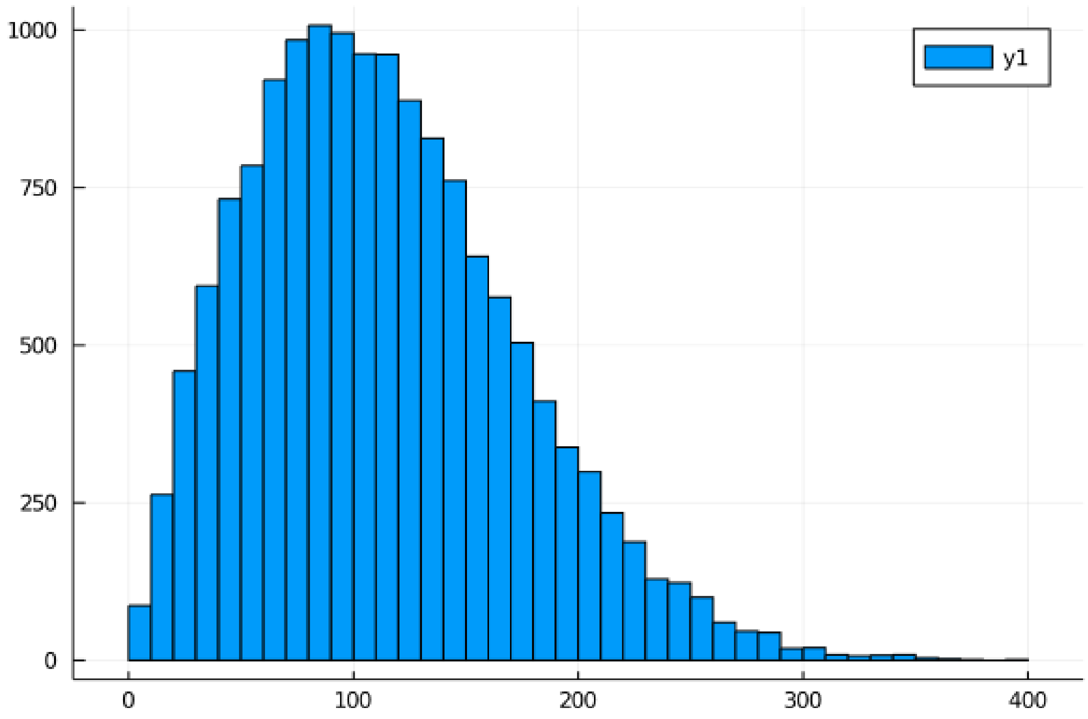


Figure 5: Velocidades geradas usando a distribuição de Maxwell.