

package link:

<https://github.com/viniciuspiccoli/simulationQP934.jl>

github link:

<https://github.com/viniciuspiccoli/QP934>

Vinicius Piccoli

1 Capítulo 1

```
1 # Vinicius Piccoli - 24/09/2020
2 # exerc cios 1 e 2
3
4
5 # ex 1 -
6 println("Ex - 1")
7 a = 2.53
8 b = 5.55
9 c = -3.40
10
11 println(a < b)
12 println(b > c)
13 println(a >= c)
14 println(c <= b)
15
16
17 # ex 2 -
18 println("Ex - 2")
19 println("a" > "b")
20 println("c" < "e")
21 println("z" >= "w")
22 println("d" <= "k" )
23
24 println("abc" < "z")
```

[\[Click here to download the code\]](#)

```
1 # exerc cio 3
2
3 include("./func.jl")
4
5 for i in rand()
6     println(f(i))
7 end
```

[\[Click here to download the code\]](#)

```

1 # fun o do exerc cio 3
2
3 function f(x::Float64)
4     return x^2 + 2*x -3
5 end

```

[\[Click here to download the code\]](#)

Script com as demais atividades do capítulo 1.

```

1 # exerc cio 4
2 function f(x::Vector)
3     x[1] = x[1] + 1
4     return x
5 end
6
7 function g(x::Vector)
8     y = copy(x)
9     y[1] = y[1] + 1
10    return y
11 end
12
13 # funcao 1
14 x = [1,1]
15 f(x)
16 println(x)
17
18 # funcao 2
19 y = [1,1]
20 g(y)
21 println(y)
22
23
24
25 #####
26
27
28 # exerc cio 5
29
30 x = [2,2]
31 y = copy(x)
32
33 y[1] = y[1] - 5
34
35 println(x)
36 println(y)
37
38
39
40 # exerc cio 6
41
42 x = [i for i in 1:10]
43 y = Vector{Int64}(undef, length(x))
44
45 println(x)
46
47 for i in 1:length(x)
48     y[i] = x[i]
49 end

```

```
50
51   y[1] = y[1] - 19
52   y[6] = y[6] + 5
53
54
55   println(x)
56   println(y)
```

[\[Click here to download the code\]](#)

2 Capítulo 2 -Simulação de partículas - métodos ingênuos

```

1 # exercise 7
2 # ep = 100 J / sig = 3.4 angstroms - Arg wikipedia
3
4 function lj(d::Float64,ep, sig)
5     LJ = 4 * ep * ((sig/d)^12 - (sig/d)^6)
6     return LJ
7 end

```

[\[Click here to download the code\]](#)

```

1 include("./ativ_7.jl")
2
3 # exercise 8, 9 and 10
4 cls = ["red", "green", "blue", "pink", "orange", "black"] # Plot
5 colors
6 eps = [80, 100, 120, 140, 160, 200] # Well
7 depth . Reference value: 120 e(J)/kb(J/K) = K
8 sigs = [ 0.34, 0.36, 0.38, 0.40, 0.42, 0.44] # sigma
9 values in nm. Reference value is 0.34 nm
10 dist = [i for i in 0.32:0.01:0.85] # Distance
11 ranging from 0.32 to 0.85 nm
12
13 using Plots, LaTeXStrings
14
15 plot(framestyle=:box, legend=:best, grid=:false)
16 plot!(xlabel=L"\mathrm{r}", ylabel=L"Energy of interaction ",
17       ylims=[-130, 350])
18
19 for i in 1:length(eps)
20     energies = lj.(dist,eps[i],sigs[i])
21     plot!(dist, energies, label=sigs[i], linewidth=2, color=cls[i])
22 end
23
24 annotate!( 0.70, 100, text("Ar-Ar ep=120", :center, 10))
25 plot!(size=(375,375))
26 savefig("LJ_Ar_potential.png")

```

[\[Click here to download the code\]](#)

```

1 # functions
2 function lj(d::Float64,ep, sig)
3     LJ = 4 * ep * ((sig/d)^12 - (sig/d)^6)
4     return LJ
5 end
6
7
8 function dist(x::Vector{Float64},y::Vector{Float64})
9     d = sqrt((y[1] - x[1])^2 + (y[2]-x[2])^2)
10    return d
11 end
12
13 # exercise 11
14
15 # LJ potential

```

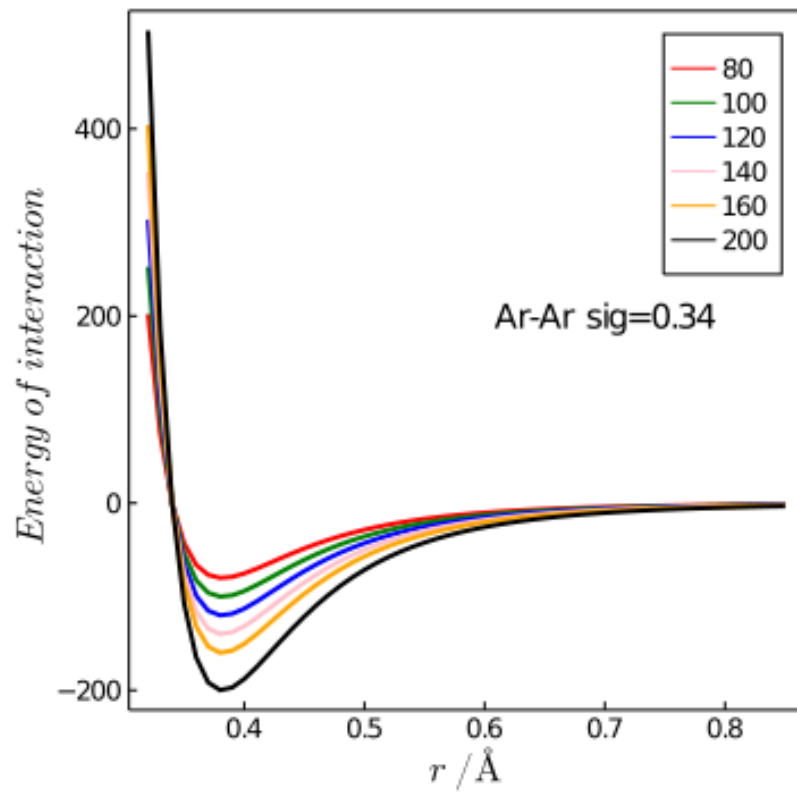


Figure 1: Curvas representando potenciais de Lennard-Jones para o argônio. É variado o valor do poço de potencial, representado pela letra grega ε .

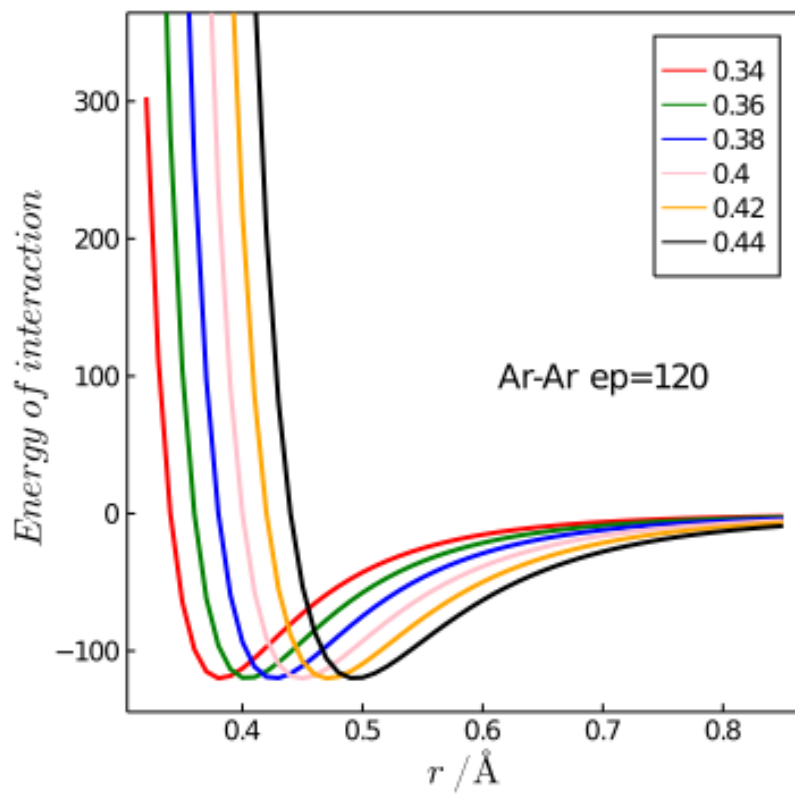


Figure 2: Curvas representando potenciais de Lennard-Jones para o argônio. É variada a distância em que o potencial é nulo, ou seja, separação mínima entre as entidades. Esta distância é dada pela letra grega σ .

```

16 sig = 1.
17 ep = 10.
18
19 box = [10*rand(2) for i in 1:10]
20 # box[point][coordinates]
21
22 # function to sum energy for all pairs of atoms in a box
23 function box_LJ(box::Vector, sig::Float64, ep::Float64)
24     Ut = 0 # energia total
25     n = 0 # n mero total de pares n o repetidos
26     for i in 1:length(box)-1
27         for j in i+1:length(box)
28             n = n + 1
29             d = dist(box[i],box[j])
30             U = lj(d,ep,sig)
31             Ut = Ut + U
32         end
33     end
34     Ut = Ut / n
35     return Ut, n
36 end
37
38 U,n = box_LJ(box,sig,ep)

```

[\[Click here to download the code\]](#)

```

1
2 using BenchmarkTools
3
4 # functions
5 function lj(d::Float64,ep::Float64, sig::Float64)
6     LJ = 4 * ep * ((sig/d)^12 - (sig/d)^6)
7     return LJ
8 end
9
10 function dist(x::Vector{Float64},y::Vector{Float64})
11     d = sqrt((y[1] - x[1])^2 + (y[2]-x[2])^2)
12     return d
13 end
14
15 # new function pot + dist - optm
16
17 function lj2(x::Vector{Float64},y::Vector{Float64},eps4::Float64,
18     sig6::Float64,sig12::Float64)
19     d = (y[1] - x[1])^2 + (y[2]-x[2])^2
20     r6 = d^3
21     r12 = r6^2
22     LJ = eps4 * ((sig12/r12) - (sig6/r6))
23     return LJ
24 end
25
26 # exercise 12
27
28 # Func 1
29 function box_LJ(box::Vector, sig::Float64, ep::Float64)
30     Ut = 0. # energia total
31     n = 0 # n mero total de pares n o repetidos
32     for i in 1:length(box)-1

```

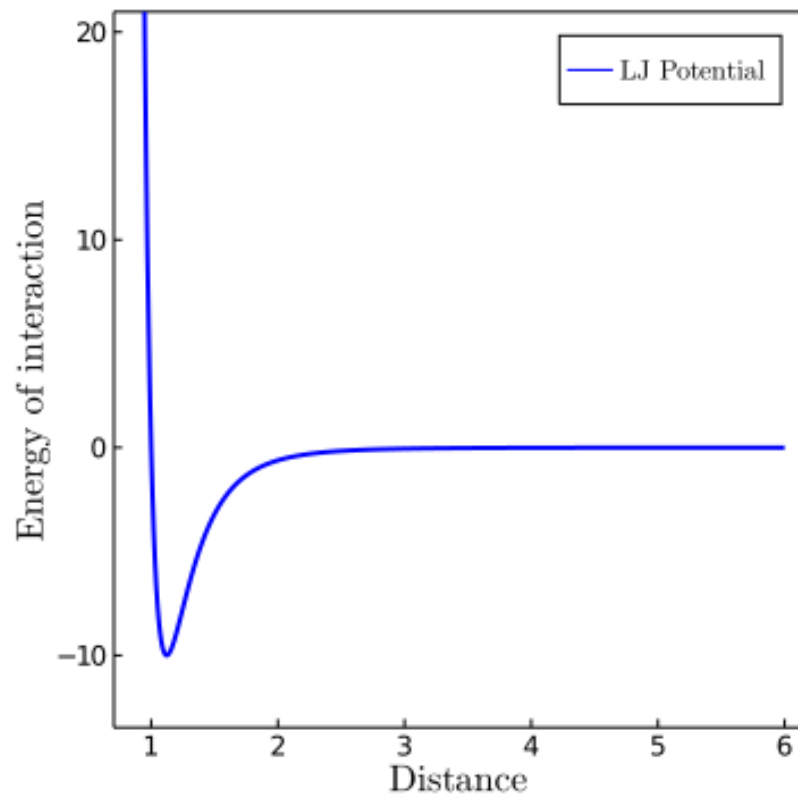


Figure 3: Potencial de Lennard-Jones com $\varepsilon = 10.0$ e $\sigma = 1$


```

32     for j in i+1:length(box)
33         n = n + 1
34         d = dist(box[i],box[j])
35         U = lj(d,ep,sig)
36         Ut = Ut + U
37     end
38 end
39 return Ut
40 end
41
42 # Func 2
43 function box_LJ2(box::Vector, sig::Float64, ep::Float64)
44     Ut = 0
45     N = length(box)
46     eps4 = 4*ep
47     sig6 = sig^6
48     sig12 = sig6^2
49
50     for i in 1:N-1
51         for j in i+1:N
52             U = lj2(box[i],box[j],eps4,sig6,sig12)
53             Ut = Ut + U
54         end
55     end
56
57     return Ut
58 end
59
60 using BenchmarkTools, Test
61 # LJ potential
62 sig = 1.
63 ep = 10.
64
65 box = [10*rand(2) for i in 1:10]
66
67 @test isapprox(box_LJ(box,sig,ep), box_LJ2(box,sig,ep))
68 print("LJ 1"); @btime box_LJ($box,$sig,$ep)
69 print("LJ 2 - otimizado"); @btime box_LJ2($box,$sig,$ep)

```

[\[Click here to download the code\]](#)

2.0.1 Condições periódicas de contorno

```
1 # exercise 15
2 let
3   # box
4   pts = [100*rand(2) for i in 1:2]
5
6   # distance between two potins according to the minimal-image
   convention
7   ds = sqrt((pts[1][1] - pts[2][1])^2 + (pts[1][2]-pts[2][2])^2) % 10.
8   for i in 1:2
9     if ds > 0.5*10
10      pts[2][i] = pts[2][i] - 10
11    elseif ds < -0.5*10
12      pts[2][i] = pts[2][i] + 10
13    end
14  end
15  dn = sqrt((pts[2][1] - pts[1][1])^2 + (pts[2][2]-pts[1][2])^2)
16
17 println(dn)
18
19 end
```

[\[Click here to download the code\]](#)

```
1 # exercise 16
2
3 # function that will create a box with side equal to "range" and "N"
   points within.
4 function point_gen(range::Int64, N::Int64)
5   pts = [range*rand(2) for i in 1:N]
6   return pts
7 end
8
9 # Function to calculate the distance between two points
10 function pbc(x::Vector{Float64},y::Vector{Float64}, L)
11   ds = y - x
12   for i in 1:2
13     ds[i] = ds[i]%L
14     if ds[i] > 0.5*L
15       ds[i] = ds[i] - L
16     elseif ds[i] < -0.5*L
17       ds[i] = ds[i] + L
18     end
19   end
20   return sqrt(ds[1]^2 + ds[2]^2)
21 end
22
23 function r(x::Vector{Float64},y::Vector{Float64}, side)
24   # dx is a vector of the difference in position of y and x
25   dx = y - x
26   for i in 1:2
27     dx[i] = dx[i]%side
28     if dx[i] > side/2
29       dx[i] = dx[i] - side
30     elseif dx[i] < -side/2
31       dx[i] = dx[i] + side
32     end
33   end
34 end
```

```

33     end
34     return sqrt(dx[1]^2+dx[2]^2)
35 end

```

[\[Click here to download the code\]](#)

```

1  # exercise 17
2
3  include("./ativ_16.jl")
4
5  let
6
7      box      = point_gen(100,100)
8      pbc_side = 10
9      dists    = Vector{Float64}(undef, (length(box)-1) * length(box))
10     dist2     = Vector{Float64}(undef, (length(box)-1) * length(box))
11
12
13     n = 0
14     for i in 1:length(box)-1
15         for j in i+1:length(box)
16             n = n + 1
17             dists[n] = r(box[i],box[j], pbc_side)
18             dist2[n] = pbc(box[i],box[j], pbc_side)
19         end
20     end
21
22     d_max = maximum(dists)
23     d_max2= maximum(dist2)
24     d_pbc = sqrt(pbc_side^2 + pbc_side^2) / 2
25
26     println("Max distance found: $d_max")
27     println("Max distance found2: $d_max2")
28     println("Max distance possible: $d_pbc")
29
30 end

```

[\[Click here to download the code\]](#)

```

1  include("./ativ_16.jl")
2
3  # exercise 18
4
5  function lj(d::Float64,ep, sig)
6      LJ = 4 * ep * ((sig/d)^12 - (sig/d)^6)
7      return LJ
8  end
9
10 box = point_gen(100,100)
11
12 for i in 1:length(box)
13     for j in i+1:length(box)
14         println("Pair : $i - $j")
15         dn = dist(box[i],box[j])
16         d = pbcseparation(box[i],box[j],10)
17         U = lj(d,10,5)
18         println(" Distance      = ",dn)
19         println(" Distance MI = ",d)
20         println(" Energy of interaction = ",U)

```

```

21 end
22 end

```

[\[Click here to download the code\]](#)

```

1 # exercise 19 - 21
2
3 # functions
4
5 # function that will create a box with side equal to "range" and "N"
  points.
6 function point_gen(range::Int64, N::Int64)
7     pts = [range*rand(2) for i in 1:N]
8     return pts
9 end
10
11 # function to calculate the distance between two points according to
    the minimal-distance convention
12 function pbcseparation(x::Vector{Float64},y::Vector{Float64}, L::Int64
    )
13     for i in 1:2
14         ds = (y[i] - x[i])%L
15         if ds > 0.5*L
16             y[i] = y[i] - L
17         elseif ds < -0.5*L
18             y[i] = y[i] + L
19         end
20     end
21     dn = sqrt((y[1] - x[1])^2 + (y[2]-x[2])^2)
22     return dn
23 end
24
25 # Calculation of the distance between two points
26 function dist(x::Vector{Float64},y::Vector{Float64})
27     d = sqrt((y[1] - x[1])^2 + (y[2]-x[2])^2)
28     return d
29 end
30
31 # Lennard-Jones potential
32 function lj(d::Float64,ep, sig)
33     LJ = 4 * ep * ((sig/d)^12 - (sig/d)^6)
34     return LJ
35 end
36
37 #
    #####
38
39 # first program - calculating the LJ pot of all pairs
40 function all_pairs(box,e,sig)
41     side = 10
42     Ut = 0.
43     for i in 1:length(box)
44         for j in i+1:length(box)
45             U = lj(pbcseparation(box[i],box[j],side),e,sig)
46             Ut = Ut + U
47         end
48     end

```

```

49 return Ut
50 end
51
52 # Program that calculates the LJ pot only if the distance between
    particles is smaller than the cutoff.
53 function cutoff_pairs(box,e,sig,cutoff)
54     side = 10
55     Ut = 0.
56     for i in 1:length(box)
57         for j in i+1:length(box)
58             d = pbcseparation(box[i],box[j],side)
59             if d <= cutoff
60                 Ut += lj(d,e,sig)
61             end
62         end
63     end
64     return Ut
65 end
66
67
68 # box with 100 points with coordinates ranging from 0 to 10
69
70 box = point_gen(10,100);
71 cutoff = 2.
72 e = 2.
73 sig = 0.5
74
75 using BenchmarkTools, Test
76 @test all_pairs(box,e,sig)      cutoff_pairs(box,e,sig,cutoff)
77
78 print("All pairs ");          @btime    all_pairs($box,$e,$sig)
79 print("Cutoff pairs ");      @btime    cutoff_pairs($box,$e,$sig,$
    cutoff)
80
81 # println("result all pairs          = ", all_pairs(box,e,sig))
82 # println("result with cutoff      = ", cutoff_pairs(box,e,sig,
    cutoff))

```

[\[Click here to download the code\]](#)

```

(base) viniuspicoli@viniuspicoli-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) viniuspicoli@viniuspicoli-Inspiron-5537 ~/Documents/QP934/CAP_2/PBC $

```

Figure 4: Velocidade de execução dos códigos.

2.1 Método das células ligadas

```

1 include("./funcoes_otm.jl")
2 using BenchmarkTools
3
4
5 ### exercise 25

```

```

6
7 # function - ijcell
8 ijcell(pair, cutoff) = [trunc(Int64,pair[1]/cutoff) + 1 , trunc(Int64
  , pair[2]/cutoff) + 1]
9
10
11 # # function - class
12 # function class(pts, cutoff, side)
13 #   ncellsx = trunc(Int64,side/cutoff) + 1 # we are going to start
    from 1
14 #   ncellsy = trunc(Int64,side/cutoff) + 1
15 #
16 #   list = zeros()
17 #   for i in 1:length(p)
18 #     push!(,i)
19 #
20 #   end
21 #
22 # end
23 # test of execution
24 # N = collect(100:50:5000);
25 #
26 # for i in N
27 #
28 #   box = point_gen(12,i)
29 #   print("Time of exec with $i particles = "); @btime f($box)
30 #
31 # end
32
33
34 function f(vec;cutoff=3.)
35   cls = Vector{Vector{Int64}}(undef,length(vec))
36   for i in 1:length(vec)
37     x = ijcell(vec[i],cutoff)
38     cls[i] = x
39   end
40   return cls
41 end
42
43 box = point_gen(12, 10000)
44 cls = f(box)
45
46 # println(cls)

```

[\[Click here to download the code\]](#)

```

1 # exercise 26
2 include("ativ_25.jl")
3
4 box = point_gen(12,50) # 50 points with coordinates from 0 to 12
5 cells = f(box)         # vector that relates particle with its cell
6
7
8 # function to classify
9 function assign(data)
10   pts = maximum(data)
11   M = [ [] for i in 1:pts[1], j in 1:pts[2] ] # Array to save
    particles in a cell

```

```

12 N = zeros(Int64,pts[1],pts[1]) # Array to save the number of
    particles in each cell
13
14 # saving the particles that are inside of the same cell
15 for i in 1:length(data)
16     var = data[i]
17     push!(M[var[1],var[2]],i)
18 end
19
20 # counting the number of particles in each cell
21 for i in 1:pts[1], j in 1:pts[2]
22     N[i,j] = count(i->(i!=0), M[i,j])
23 end
24
25 return M, N
26
27 end
28
29
30 pts, ns = assign(cells)

```

[\[Click here to download the code\]](#)

```

1 # exercise 28
2
3 # using push
4 function assign(data)
5     pts = maximum(data)
6     M = [ Int[] for i in 1:pts[1], j in 1:pts[2] ] # Array to save
    particles in a cell
7     N = zeros(Int64,pts[1],pts[1]) # Array to save the number of
    particles in each cell
8
9     # saving the particles that are inside of the same cell
10    for i in 1:length(data)
11        var = data[i]
12        push!(M[var[1],var[2]],i)
13    end
14
15    # counting the number of particles in each cell
16    for i in 1:pts[1], j in 1:pts[2]
17        N[i,j] = count(i->(i!=0), M[i,j])
18    end
19
20    return M, N
21 end
22
23
24 # using pre-allocated vectors
25 function assign_vec(data)
26     pts = maximum(data)
27
28     M = zeros(Int64, pts[1], pts[2],20) # 10 is the maximum number of
    positions for each cell
29     N = zeros(Int64, pts[1], pts[2])
30
31     for i in 1:length(data)
32         icell = data[i][1]

```

```

33     jcell = data[i][2]
34     N[icell,jcell] = N[icell,jcell] + 1
35     M[icell,jcell,N[icell, jcell]] = i
36     end
37
38     return M, N
39 end
40
41
42 # testing
43 using BenchmarkTools
44 include("./ativ_25.jl")
45
46 # assign_vec alt
47 pts = maximum(cls)
48
49 list = zeros{Int64, pts[1], pts[2],20} # 10 is the maximum number of
    positions for each cell
50 nlist = zeros{Int64, pts[1], pts[2]}
51
52 function assign_vec!(data,M,N)
53
54     for i in 1:length(data)
55         icell = data[i][1]
56         jcell = data[i][2]
57         N[icell,jcell] = N[icell,jcell] + 1
58         M[icell,jcell,N[icell, jcell]] = i
59     end
60
61     return M, N
62 end
63
64 println("Bechmarks")
65 print("execution time (push) = "); @btime assign($cls)
66 print("execution time (alloc) = "); @btime assign_vec($cls)
67 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

```

[\[Click here to download the code\]](#)

```

1  # exercise 29
2
3  include("./funcoes_otm.jl")
4
5  box = point_gen(15,50)
6  cls = f(box)
7
8  N = length(box);
9
10 # creating vectors
11 pts = maximum(cls) # max number of i and j
12 list = zeros{Int64, pts[1], pts[2],N} # 20 - maximum number of cell
    positions
13 nlist = zeros{Int64, pts[1], pts[2]}
14
15 # function to save particles in each cell
16 function assign!(data, M, N)

```



```

17     for i in 1:length(data)
18         icell = data[i][1]
19         jcell = data[i][2]
20         N[icell,jcell] = N[icell,jcell] + 1
21         M[icell,jcell,N[icell, jcell]] = i
22     end
23 end
24
25
26
27 function mindist(p,cellparticles,cutoff)
28     np = length(p) # Number of particles
29     nc = size(cellparticles,1) # Dimension of the grid
30     d = +Inf
31     U = 0
32     for ip in 1:np
33         icell = trunc(Int64,p[ip][1]/cutoff)+1
34         jcell = trunc(Int64,p[ip][2]/cutoff)+1
35         for i in icell-1:icell+1
36             if ( i < 1 || i > nc ) continue end # Border
37             for j in jcell-1:jcell+1
38                 if ( j < 1 || j > nc ) continue end # Border
39                 # Loop over the particles of this cell
40                 for jp in cellparticles[i,j,:]
41                     if jp > ip # Skip repeated
42                         # Compute distance and keep minimum
43                         d = min(d,sqrt( (p[ip][1]-p[jp][1])^2 + (p[ip][2]-p[jp]
44                             ][2])^2 ) )
45                         U = U + lj(d,5,2)
46                     end
47                 end
48             end
49         end
50     return U
51 end
52
53
54 # using push
55 function assign(data)
56     pts = maximum(data)
57     M = [ Int[] for i in 1:pts[1], j in 1:pts[2] ]
58     N = zeros(Int64,pts[1],pts[1])
59
60     # saving the particles that are inside of the same cell
61     for i in 1:length(data)
62         var = data[i]
63         push!(M[var[1],var[2]],i)
64     end
65
66     # counting the number of particles in each cell
67     for i in 1:pts[1], j in 1:pts[2]
68         N[i,j] = count(i->(i!=0), M[i,j])
69     end
70
71     return M, N
72 end

```

```

73
74
75 function mindist2(p,cellparticles,cutoff)
76     np = length(p) # Number of particles
77     nc = size(cellparticles,1) # Dimension of the grid
78     d = +Inf
79     U = 0
80     for ip in 1:np
81         icell = trunc(Int64,p[ip][1]/cutoff)+1
82         jcell = trunc(Int64,p[ip][2]/cutoff)+1
83         for i in icell-1:icell+1
84             if ( i < 1 || i > nc ) continue end # Border
85             for j in jcell-1:jcell+1
86                 if ( j < 1 || j > nc ) continue end # Border
87                 # Loop over the particles of this cell
88                 for jp in cellparticles[i,j]
89                     if jp > ip # Skip repeated
90                         # Compute distance and keep minimum
91                         d = min(d,sqrt( (p[ip][1]-p[jp][1])^2 + (p[ip][2]-p[jp
92                             ][2])^2) )
93                         U = U + lj(d,5,2)
94                     end
95                 end
96             end
97         end
98     return U
99 end
100
101 using BenchmarkTools
102 pts = maximum(cls)
103
104 #print("assign alloc"); @btime assign!($cls,list,nlist) setup=(list=
105     zeros(Int64, pts[1], pts[2],N); nlist=zeros(Int64, pts[1], pts[2]))
106     evals=1
107 ;
108 #print("assign push"); @btime assign($cls)
109
110 #print("Time of ex for alloc = "); @btime mindist($box,$list,3.)
111 #print("Time of ex for push = "); @btime mindist2($box,$plist,3.)
112
113 assign!(cls,list,nlist) ;
114 plist, pnlist = assign(cls) ;
115
116 U = mindist(box,list,3.)
117 println(U)
118 U2 = mindist2(box,plist,3.)
119 println(U2)
120
121 # exercise 30

```

[\[Click here to download the code\]](#)

```

1 # exercise 30
2
3 function wrap_cell(dms,id,jd)

```

```

4   i = dms[1]
5   j = dms[2]
6
7   if id < 1
8       id = id + i
9   elseif id > i
10      id = id - i
11  end
12  if jd < 1
13      jd = jd + j
14  elseif jd > j
15      jd = jd - j
16  end
17
18  return id, jd
19 end
20
21
22
23 function check_wrap(nc)
24     list = Matrix{Int}(undef,0,4)
25     for i in 1:nc[1]
26         for j in 1:nc[2]
27             for ic in i-1:i+1
28                 for jc in j-1:j+1
29                     iw, jw = wrap_cell(nc,ic,jc)
30                     list = vcat(list,[ic jc iw jw])
31                 end
32             end
33         end
34     end
35     return list
36 end
37
38
39 nc = [3,3]
40 list = check_wrap(nc)
41 for i in 1:length(list)
42     println(list[i])
43 end

```

[\[Click here to download the code\]](#)

```

1  import Random
2
3  # Box
4  function point_gen(range::Int64, N::Int64)
5      Random.seed!(321)
6      pts = [range*rand(2) for i in 1:N]
7      return pts
8  end
9
10 function cutoff_pairs(box,e,sig,cutoff;side=10)
11     Ut = 0.
12     d = +Inf
13     for i in 1:length(box)
14         for j in i+1:length(box)
15             d = pbcseparation(box[i],box[j],side)

```

```

16         if d <= cutoff
17             Ut = Ut + lj(d,e,sig)
18         end
19     end
20 end
21 return Ut
22 end
23
24 # box with 10000 particles with coordinates ranging from 0 to 100
25
26 box = point_gen(100,10000);
27 cutoff = 2.
28 e = 2.
29 sig = 0.5
30
31 A = cutoff_pairs(box,e,sig,cutoff)
32 println(A)

```

[\[Click here to download the code\]](#)

```

1 # exercise 32
2 include("funcoes_otm.jl")
3
4 # box with 10000 particles with coordinates ranging from 0 to 100
5
6 box = point_gen(100,10000);
7 side = 100
8 cutoff = 10.
9
10 e = 2.
11 sig = 0.5
12
13 cls = f(box,cutoff)
14 list = assign(cls);
15
16 list2 = assign_data(box, side, cutoff)
17
18 using BenchmarkTools, Test
19
20 print("Cutoff pairs ");      @btime    cutoff_pairs($box,$e,$sig,$
    cutoff,$side)
21 print("Linked cells");      @btime    pot_ener($box,$list, $cutoff,$e
    ,$sig,$side)
22
23 A = cutoff_pairs(box,e,sig,cutoff,side)
24 println(A)
25
26 B = pot_ener(box,list,cutoff,e,sig,side)
27 println(B)

```

[\[Click here to download the code\]](#)

2.2 Listas ligadas

```

1 include("../CELULA_LIGADA/funcoes_otm.jl")
2
3 box = point_gen(12,10)
4 cutoff = 3
5 side = 12

```

```

6
7 Nc = trunc(Int64 ,side / cutoff) # Number of cells
8 N = length(box) # Number of particles
9
10 # stating linked lists
11 first_atom = zeros(Int64, Nc, Nc)
12
13 #first_atom = [ Int64[] for i in 1:Nc, j in 1:Nc ]
14 next_atom = zeros(Int64, N)
15
16
17 # for iat in 1:N
18 #
19 # icell = trunc(Int64, box[iat][1]/cutoff) + 1
20 # jcell = trunc(Int64, box[iat][2]/cutoff) + 1
21 #
22 # next_atom[iat] = first_atom[icell, jcell]
23 #
24 # first_atom[icell, jcell] = iat
25 #
26 #
27 # end
28
29
30 # function
31 function linkedlist!(box, side, cutoff, first_atom, next_atom)
32
33     for iat in 1:N
34
35         icell = trunc(Int64, box[iat][1]/cutoff) + 1
36         jcell = trunc(Int64, box[iat][2]/cutoff) + 1
37
38         next_atom[iat] = first_atom[icell, jcell]
39
40         first_atom[icell, jcell] = iat
41
42     end
43 end
44
45 linkedlist!(box,side,cutoff,first_atom, next_atom)

```

[\[Click here to download the code\]](#)

```

1 include("../CELULA_LIGADA/funcoes_otm.jl")
2
3 # box, cutoff and side of the system
4 box = point_gen(12,1000)
5 cutoff = 3
6 side = 12
7
8 # for push method
9 cls=f(box,cutoff)
10
11 #linked list parameters
12 # Nc = trunc(Int64 ,side / cutoff) # Number of cells
13 # N = length(box) # Number of particles
14
15 # first_atom = zeros(Int64, Nc, Nc) # stating linked lists

```

```

16 # next_atom = zeros(Int64, N)      # next atoms
17
18 using BenchmarkTools
19
20 print("Performance of the push method") ;@btime assign_data($box,$
    side,$cutoff)
21 print("My push method") ;@btime assign($cls)
22 print("Performance of the linked list method") ;@btime linkedlist!($
    box,$side,$cutoff) # , $first_atom,$next_atom)

```

[\[Click here to download the code\]](#)

```

1 # exercise 36
2 function energy(box,cutoff, first_atom, next_atom,ep,sig,side)
3
4     vec = [length(first_atom[:,1]), length(first_atom[1,:])]      #
5     #vector with maximum dimensions
6     N = length(box)      #
7     #number of atoms
8     #d = +Inf
9     Ut = 0.
10
11     for iat in 1:N
12         icell = trunc(Int64,box[iat][1]/cutoff) + 1
13         jcell = trunc(Int64,box[iat][2]/cutoff) + 1
14
15         for i in icell-1:icell+1
16             for j in jcell-1:jcell+1
17                 iw, jw = wrap_cell(vec,i, j)
18                 jat = first_atom[iw,jw]
19
20                 while jat > 0
21                     # Compute distance and keep minimum
22                     #d = min(d,sqrt( (box[iat][1]-box[jat][1])^2 + (box[iat][2]-
23                     box[jat][2])^2) )
24                     #d = (box[iat][1]-box[jat][1])^2 + (box[iat][2]-box[jat][2])
25                     ^2
26                     d = pbcseparation(box[iat], box[jat],side)
27
28                     #if d==0
29                     # println("fdp")
30                     # println(iat, " ", jat)
31                     #end
32
33                     if jat > iat
34                         Ut = Ut + U(d,ep,sig)
35                     end
36                     jat = next_atom[jat]
37                 end
38             end
39         end
40     end
41     return Ut
42 end

```

```

43 function energy(box,cutoff,ep,sig,side)
44
45     first_atom, next_atom = linkedlist(box, side, cutoff)
46
47     vec = [length(first_atom[:,1]), length(first_atom[1,:])]      #
48     # vector with maximum dimensions
49     N = length(box)      #
50     # number of atoms
51     #d = +Inf
52     Ut = 0.
53
54     for iat in 1:N
55         icell = trunc(Int64,box[iat][1]/cutoff) + 1
56         jcell = trunc(Int64,box[iat][2]/cutoff) + 1
57
58         for i in icell-1:icell+1
59             for j in jcell-1:jcell+1
60                 iw, jw = wrap_cell(vec,i, j)
61                 jat = first_atom[iw,jw]
62
63                 while jat > 0
64                     # Compute distance and keep minimum
65                     #d = min(d,sqrt( (box[iat][1]-box[jat][1])^2 + (box[iat][2]-
66                     box[jat][2])^2) )
67                     #d = (box[iat][1]-box[jat][1])^2 + (box[iat][2]-box[jat][2])
68                     ^2
69
70                     #if d==0
71                     # println("fdp")
72                     # println(iat, " ", jat)
73                     #end
74
75                     if jat > iat
76                         d = pbcseparation(box[iat], box[jat],side)
77                         Ut = Ut + U(d,ep,sig)
78                     end
79                     jat = next_atom[jat]
80                 end
81             end
82         end
83     end
84     return Ut
85 end
86
87
88
89 # exercise 37
90 include("../funcoes_otm.jl")
91 include("../initial-point.jl")
92
93
94 side = 100
95 cutoff = 2

```

```

96 e = 5.0
97 sig = 0.5
98 box = initial_point(10_000,side,0.9)
99
100
101 # import Random
102 # Random.seed!(321)
103 # box = [ 100*rand(2) for i in 1:10_000 ]
104 #
105 # side = 100
106 # cutoff = 10.
107 # e = 2.
108 # sig = 0.5
109
110
111 using BenchmarkTools, Test
112
113 # println("Bechmarks for the lists gen")
114 # print("Time to create lists using linkedlists method") ;@btime
115   linkedlist($box,$side, $cutoff)
116 # print("Time to create a matrix using push method") ;@btime
117   assign_data($box, $side, $cutoff)
118
119
120 fatm, natm = linkedlist(box,side, cutoff)
121 list2 = assign_data(box, side, cutoff) # linked cells direct
122
123 N = 10000 # number of molecules
124 vec = [length(fatm[:,1]), length(fatm[1,:])] # maximum number of
125   cells
126
127
128 #print("naive method") ; @btime cutoff_pairs($box,$e,$sig,$cutoff,$side
129   )
130 #Uz = cutoff_pairs(box,e,sig,cutoff,side)
131 #println(Uz)
132 #println(" ")
133 #
134 #print("Linked cells method") ;@btime pot_ener($box,$list2,$cutoff,$e,$
135   sig,$side)
136 #Ut = pot_ener(box,list2,cutoff,e,sig,side)
137 #println(Ut)
138 #println(" ")
139
140 print("Linked lists method") ;@btime energy($box,$cutoff, $fatm, $natm
141   , $e,$sig,$side)
142 U1=energy(box,cutoff, fatm, natm,e,sig,side,vec,N) #,N,vec
143 println(U1)
144 #println(" ")
145
146 #print("Linked listsi (2) method") ;@btime energy($box,$cutoff,$e,$sig
147   ,$side)
148 #U2=energy(box,cutoff,e,sig,side)
149 #println(U2)
150 #println(" ")

```

[\[Click here to download the code\]](#)


```

1 include("../initial-point.jl")
2
3 #box = (10000,100,0.9)
4
5
6 import Random
7 Random.seed!(321)
8 box = [ 100*rand(2) for i in 1:10_000 ]
9
10 side = 100
11 cutoff = 10.
12 #e = 2.
13 #sig = 0.5
14
15 function celllist1(p,side,cutoff)
16     n = round(Int64,side/cutoff)
17     cellparticles = [ Int64[] for i in 1:n, j in 1:n ]
18
19     for i in 1:length(p)
20         icell = trunc(Int64,p[i][1]/cutoff) + 1
21         jcell = trunc(Int64,p[i][2]/cutoff) + 1
22         push!(cellparticles[icell,jcell],i)
23     end
24     return cellparticles
25 end
26
27 function celllist2(p,side,cutoff,nmax)
28     n = round(Int64,side/cutoff)
29     nparticles = zeros(Int64,n,n)
30     cellparticles = Array{Int64}(undef,n,n,nmax)
31     for i in 1:length(p)
32         icell = trunc(Int64,p[i][1]/cutoff) + 1
33         jcell = trunc(Int64,p[i][2]/cutoff) + 1
34         nparticles[icell,jcell] += 1
35         if nparticles[icell,jcell] > nmax
36             error("n > nmax")
37         end
38         cellparticles[icell,jcell,nparticles[icell,jcell]] = i
39     end
40     return cellparticles
41 end
42
43 function celllist3(box, side, cutoff) # , first_atom, next_atom
44     Nc = trunc(Int64,side / cutoff) # Number of cells
45     N = length(box) # Number of particles
46     first_atom = zeros(Int64,Nc, Nc)
47     next_atom = zeros(Int64, N)
48
49     for iat in 1:N
50         icell = trunc(Int64, box[iat][1]/cutoff) + 1
51         jcell = trunc(Int64, box[iat][2]/cutoff) + 1
52         next_atom[iat] = first_atom[icell, jcell]
53         first_atom[icell, jcell] = iat
54     end
55
56     return first_atom, next_atom
57 end

```

```

58     println("Benchmarks")
59
60
61     using BenchmarkTools
62
63     print("celllist 1") ;@btime celllist1($box,$side,$cutoff)
64     print("celllist 2") ;@btime celllist2($box,$side,$cutoff,10000)
65     print("celllist 3") ;@btime celllist3($box,$side,$cutoff)

```

[\[Click here to download the code\]](#)

3 Cálculo das forças

```

1  # function to calculate the force between a pair of particles
2  function fpair(x,y,r,data)
3
4      # r values for F calculation
5      r6 = r^6
6      r7 = r6 * r
7      r12 = r6^2
8      r13 = r12 * r
9
10     # x and y components
11     # Fz = -( dU(r) / dz) = - (dr/dz) * (dU(r)/dr), r = sqrt(dx^2,dy^2)
12     # calculation for -(dr/dr)
13
14     drdx1 = -(x[1]-y[1])/r
15     drdx2 = -(x[2]-y[2])/r
16
17     # calculation for (dU(r)/dr)
18     dudr1 = -(data.rep / r13)
19     dudr2 = -(data.att / r7)
20
21
22     dfdr = - (dudr1 - dudr2)
23
24     # calculatons of force components
25     f1 = dfdr * drdx1
26     f2 = dfdr * drdx2
27
28     # or
29
30     # f1 = ((data.rep / r13) - (data.att / r7))*drdx1
31     # f2 = ((data.rep / r13) - (data.att / r7))*drdx2
32
33     f = (-f1,-f2)
34
35     # calculation of energy of interaction
36     upair = data.eps4*(data.sig12/r12 - data.sig6/r6)
37
38
39     return upair, f
40
41 end
42
43
44

```

```

45
46 function forcepair(x,y,r,data)
47     r6 = r^6
48     r12 = r6^2
49     r7 = r6*r
50     r13 = r12*r
51
52     drdx1 = -(x[1]-y[1])/r
53     drdx2 = -(x[2]-y[2])/r
54
55     sigfac1 = -12*(data.sig12/r13)
56     sigfac2 = -6*(data.sig6/r7)
57
58     dfacdr = -data.eps4*(sigfac1 - sigfac2)
59
60     upair = data.eps4*(data.sig12/r12 - data.sig6/r6)
61     fx = ( -dfacdr*drdx1, -dfacdr*drdx2 )
62
63     return upair, fx
64 end
65 export forcepair

```

[\[Click here to download the code\]](#)

```

1 # force calculation using linked lists method
2
3
4 #prreciso acertar isso daqui
5
6
7 # f = [ [0. , 0.] for i in 1:data.N ]
8 fvec(data) = [ Vector{Float64}(undef,2) for i in 1:data.N ]
9
10 function force!(box, DATA, frc, first_atom, next_atom, nc)
11     ut = 0.
12
13     for i in 1:length(frc)
14         frc[i][1] = 0.
15         frc[i][2] = 0.
16     end
17
18     for iat in 1:DATA.N
19         icell = trunc(Int64,box[iat][1]/DATA.cutoff) + 1
20         jcell = trunc(Int64,box[iat][2]/DATA.cutoff) + 1
21         for i in icell-1:icell+1
22             for j in jcell-1:jcell+1
23                 iw, jw = wrapcell(nc, i, j)
24                 jat = first_atom[iw,jw]
25                 while jat > 0
26                     if jat > iat
27                         rij = pbcseparation(box[jat],box[iat],DATA.side)
28                         if rij <= 2.
29                             up,fp = fpair(box[jat], box[iat], rij, DATA)
30                             frc[iat] .= frc[iat] .+ fp
31                             frc[jat] .= frc[jat] .- fp
32                             ut += up
33                         end
34                     end
35                 end
36             end
37         end
38     end
39 end

```

```

35         jat = next_atom[jat]
36     end
37 end
38 end
39 end
40 return ut
41 end
42
43 export fvec, force!

```

[\[Click here to download the code\]](#)

```

1 using simulationQP934, Test
2
3
4 data = Data();
5 box = initial_point(data);
6 nc, fatm, natm = linkedlist(box,data);
7 force_vec = fvec(data)
8
9
10 utotal = force!(box, data, force_vec, fatm, natm, nc)
11
12 # test if the force is zero
13 vecx = 0.
14 vecy = 0.
15
16 for i in 1:length(force_vec)
17     vecx = vecx + force_vec[i][1]
18     vecy = vecy + force_vec[i][2]
19 end
20
21
22 @test vecx \approx 0.
23 @test vecy \approx 0.

```

[\[Click here to download the code\]](#)

```

1 function initial_velocity(N::Int64;T=298.15,R=8.3145,m=1.0)
2     vel = [ [0.,0.] for i in 1:N]
3     angle = 2*pi*rand()
4
5     for i in 1:N
6         px = rand()
7         py = rand()
8         vel[i][1] = sqrt(-(2*R*T/m)*log(1-px)) * angle    # velocity in
9         vel[i][2] = sqrt(-(2*R*T/m)*log(1-py)) * angle    # velocity in
10        the y axis
11    end
12    return vel
13 end
14
15 function norm(x::Vector{Float64})
16     s = 0.
17     for i in 1:length(x)
18         s = s + x[i]^2
19     end

```

```

20 sqrt(s)
21 end
22
23 function normVEL(x)
24     s = zeros(length(x))
25     for i in 1:length(x)
26         s[i] = norm(x[i])
27     end
28     return s
29 end
30
31 #using Plots
32
33
34 # p(v)dv
35
36 p(v;m=1.,R=8.3145,T=298.15) = m/(R*T) * v * exp(-m*v^2/(2*R*T))
37
38 # probability v < x
39 pv(x;m = 1., T = 298.15, R = 8.3145) = -exp(-m*x^2/(2*R*T)) + 1
40
41
42
43
44 #
45
46 #vels = initial_velocity(15000)
47
48 #histogram(p.(vels),vels)

```

[\[Click here to download the code\]](#)

```

1
2 function md(data::Data,MDinputs::MDinput)
3
4     @unpack mass    = data
5     @unpack dt, nsteps, total_time, iprint = MDinputs
6
7     box          = initial_point(data);
8     nc, fatm, natm = linkedlist(box,data);
9     force_vec     = fvec(data)
10    vel           = initial_velocity(data.N);
11    times = 0.
12
13    for i in 1:nsteps
14        ut = force!(box, data, force_vec, fatm, natm, nc)
15        times = times + dt
16
17        for ip in 1:data.N
18            # positions update @. box[ip] = box[ip] + vel[ip]*dt + force_vec[
19            ip] * (dt*dt)/(2*mass)
20            @. box[ip] = box[ip] + vel[ip]*dt + force_vec[ip] * (dt*dt)/(2*
21            mass)
22            # velocity update
23            @. vel[ip] = vel[ip] + force_vec[ip] * dt/mass
24        end
25
26        if i%iprint ==0

```

```
25     println("Step #$i")
26     println("Total energy = ",ut)
27 end
28
29 #if i%iprint == 0
30 #   println("Total energy at $i = ",ut)
31 #end
32
33 end
34 end
```

[\[Click here to download the code\]](#)

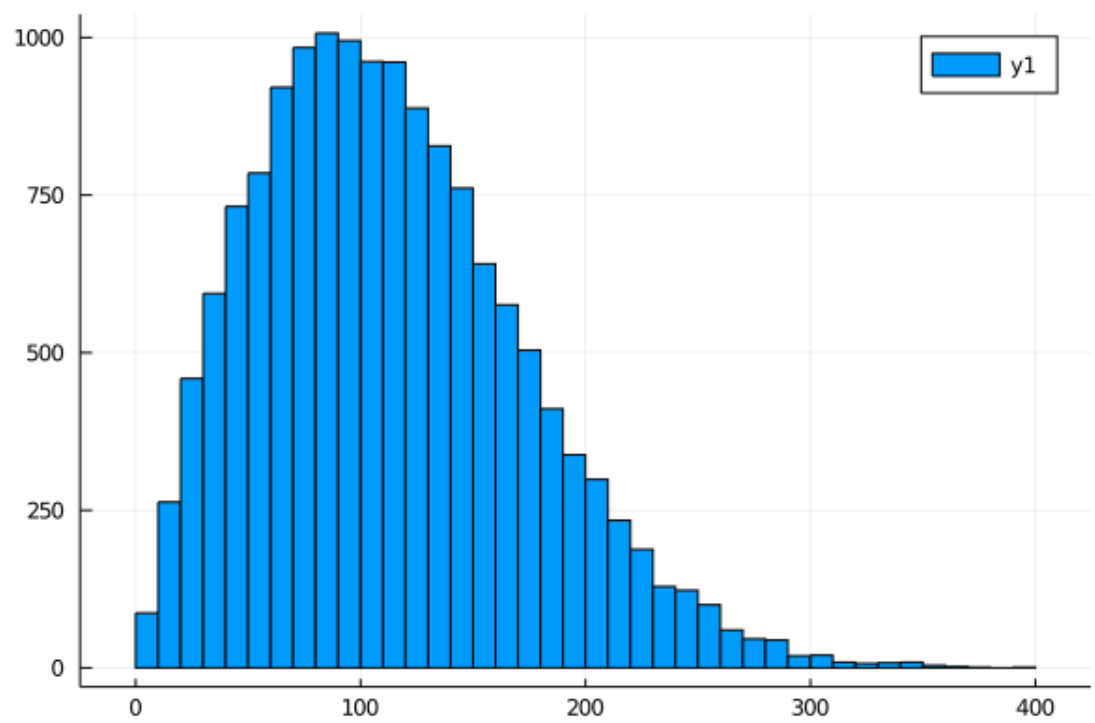


Figure 5: Velocidades geradas usando a distribuição de Maxwell.