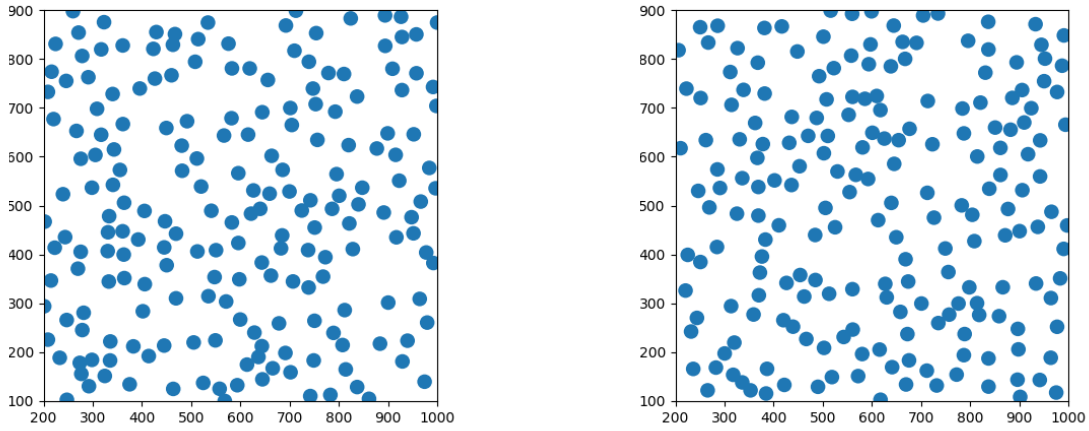
Scientific Programming Assignment 3: Simulating spatial point patterns

USN: 303039534

1 dmin2d

We start by implementing the dmin2d model as described in the assignment and plot two examples of a dmin2d simulation with parameters n=200, m=30, s=5 and ranges of x=[200:1000] and y = [100:900] (figure 1).



(a) (b)

Figure 1: Two examples of running the dmin2d model with n=200, m=30, s=5. The diameter used for plotting is the mean of the dmin model; pairs of points exhibiting overlap must thus have at least one point added with a dmin smaller than the mean.

2 Regularity Index

We proceed to implement a function *calc_RI()* to calculate the regularity index of a given pattern. The regularity index is defined as

$$RI = \frac{\text{mean}(dmin(i))}{\text{stdev}(dmin(i))} \tag{1}$$

where $dmin(i)$ is the distance between point i and its nearest neighbor. RI is a measure of how regular the local spacing of points is; if all points have similar nearest neighbor spacings, the standard deviation will be low and the regularity index high. If, on the other hand, pairwise distances differ a lot, the standard deviation will be high and the regularity index low. This is weighted by the mean nearest-neighbor distance to account for scalings of the grid.

For a given set of parameters, we can proceed to generate 1000 random patterns and calculate the regularity index of each pattern. We report the 50th largest value as a measure of regularity for this set of parameters. This is the boundary between the 5% highest and 95% lowest regularity indices we generate given these parameters, and thus tells us how regular a pattern we can expect to generate within a reasonable number of trials (here 20) with these parameters. In the following, we denote this regularity measure RI50 and calculate it with the function *run_sims()*.

We use *run_sims()* to calculate RI50 for a grid with n=200, m=0, s=0, x=[0:1000] and y=[0:1000]; that is for a square pattern of 200 points with no exclusion zone. For this set of parameters, we find that RI50 = 2.070. Repeating the calculation 10 times we find a mean of 2.0695 with a standard deviation of 0.006 across trials. This shows another benefit of the RI50 measure in that it is relatively consistent across trials in contrast to the raw RI value for a single grid which is found to have a mean and standard deviation of 1.917 and 0.125 across 10 trials.

We also see that while RI50 is significantly higher than the theoretical expectation of RI = 1.91, the mean RI across 10 trials gets relatively close. This is because RI50 reports the 95th percentile RI value rather than the expectation. If we instead generate 10^5 grids with 200 points and take the mean of the RI50 values, we arrive at an empirical expectation of 1.85 which is also similar to the theoretical expectation, although somewhat lower due to the finite grid size.

We can also investigate how RI50 changes with both the number of points added and the shape of the grid. We constrain the area of the grid to 10^6, giving us one degree of freedom specifying the shape, and we quantify this by

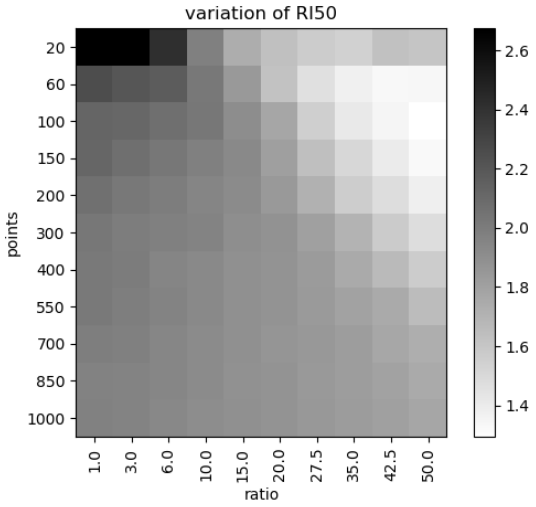


Figure 2: Dependence of RI50 on number of points added to the grid and ratio of x and y axes for a grid of area 10^6 (although RI50 is invariant of grid area when using a pointsize of 0).

the ratio of the x and y dimensions. We scan this ratio as well as the number of points added and report the results as a heatmap in figure 2.

We observe a large variation in RI50 values with ratio for small n. For ratios near 1, n=20 provides the highest RI50 value which we can rationalize since smaller n leads to more variation between grids and thus a higher 95th percentile value. Surprisingly therefore, we see that n=20 does not correspond to the maximum RI50 for ratios of 15-50.

As we move to higher n, we get less variation between trials and lower RI50 values for the near-square grids. Higher n-values also exhibit less variability between different ratios with a range of 0.20 for n=1000 compared to 1.13 for n=20 since the proportion of points directly adjacent to an edge decreases with n.

The lowest RI50 values are obtained with relatively small numbers of points (60-100) and high x:y ratios. This is because increasing x:y increases the proportion of points directly adjacent to an edge which increases nearest neighbor variability, particularly for small n.

The reason for the relatively high RI50 values still observed for n=20 is the balance between the increased number of points adjacent to an edge, and the increase in inter-trial variation as we decrease the number of points, This means that even though the median RI50 is lower for n=20 than n=60 for ratios of 20-50, the 95th percentile value is still higher.

3 Fit to Data

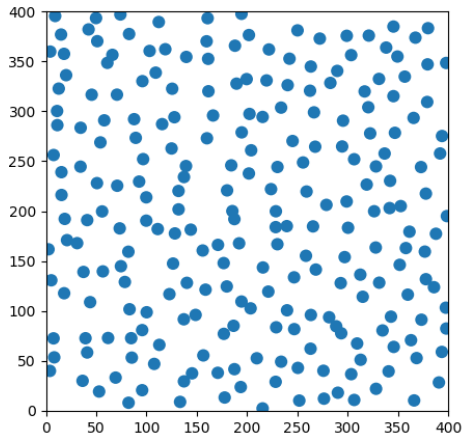
We load the file spa3_real.dat and plot the points in figure 3a. This is the set of reference points (ref) for the remainder of this section

In order to find a set of parameters that generates a distribution of points with similar spatial properties (here represented by RI), we start by defining a spatial similarity measure.

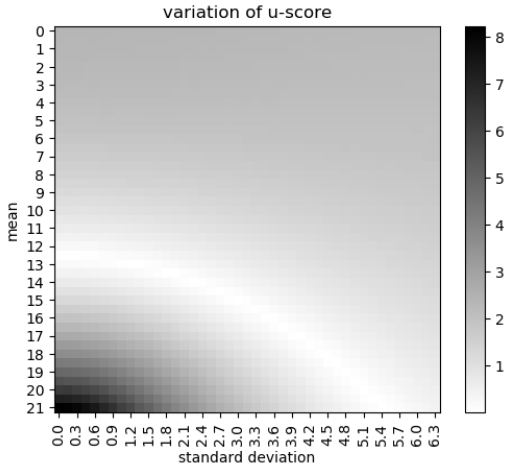
$$u_i = abs(RI(i) - \frac{1}{n-1}\sum_{i \neq j} RI(j)) \tag{2}$$

where i=1 specifies the reference grid and i =2:n specifies a set of dmin2d grids generated with a given parameter set. u_i is thus the difference between the regularity index of grid i and the 99 other grids. We therefore use u_1 as a measure of how similar the spatial properties (in this case RI) of the reference grid are to the spatial properties of points positioned with a given set of parameters. We implement the function *calc_sim()* which uses this measure to quantify similarity of spatial properties between the reference grid and a set of parameters.

We start by coarsely investigating the parameter space by letting m vary from 0 to 21 and s from 0 to 6.3, quantifying u_1 at each point (figure 3b).



(a) set of reference points



(b) u_1 for different parameters

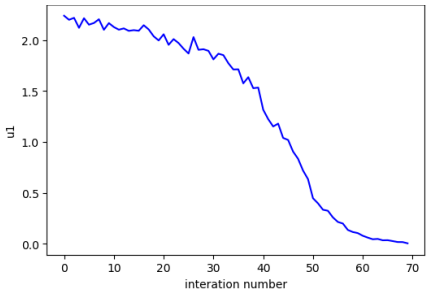
Figure 3: different packings.

We see that rather than having a single minimum, there appears to be a line of maximum similarity running from (0.0, 12.5) to (5.5, 21) in a roughly parabolic shape.

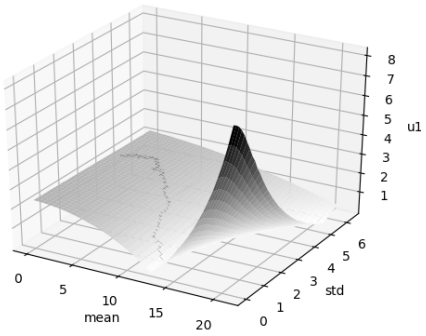
Since the u_1-landscape appears to be quite well-behaved, we proceed to write a steepest-descent algorithm to find an optimum combination of m and sd to reproduce the spatial properties of the reference grid.

For this minimization, we let the number of grids n used in *calc_sim()* at each iteration be adaptive rather than fixed at n=100 since we require a higher number of points for an accurate empirical gradient near the minimum.

Since we approach a minimum with $u_1 = 0$, we let $n = Int(max(5000 * exp(-7 * u), 20))$ which runs from a minimum value of 20 for $u_1 > 0.55$ to a maximum value of 5000 at $u_1 = 0$. We also use an adaptive delta ($\delta = u_1/2$) for calculating empirical gradients and an adaptive learning rate ($\epsilon = u_1$). The standard deviation of u_1 across 10 calculations with n=1000 is 0.007 and we thus set a convergence threshold of 0.01 for the optimization.



(a) u_1 as a function of iteration number. The adaptive δ, ϵ and n result in early steps being rapid and coarse and later steps slower but direct.



(b) projection of steepest descent path onto u_1-landscape

Figure 4

The result of this optimization is m = 12.64 and s = 0.56 giving $u_1 = 0.005$. We see from figure 4a that the optimization gives a near-monotonic decrease in u_1 with each iteration. We take this optimization path and project it onto the u_1 landscape from figure 3b and see that it does indeed proceed in a relatively direct path towards the minimum-u1 valley. Within this valley, it is unlikely that there is a single distinct minimum, and even if there is it is likely not discernable within our error margin.

The steepest descent approach thus allows us to identify a set of parameters yielding similar spatial properties to

the real pattern and could be repeated to identify multiple sets of parameters. However, since we only have two free parameters and the parameter space is well behaved, in the present case we could identify the line of similarity from an exhaustive search as in figure 3b.

For a more fine-grained determination of which parameter sets lead to $u_1 = 0$, we note that each s from 0.0 to 5.5 appears to have a minimum u_1 near 0 for an appropriate m. We therefore adapt our steepest descent algorithm to allow for optimization of only the mean given a fixed standard deviation. This allows us to find for each s the m that minimizes u_1, and we plot this in figure 5.

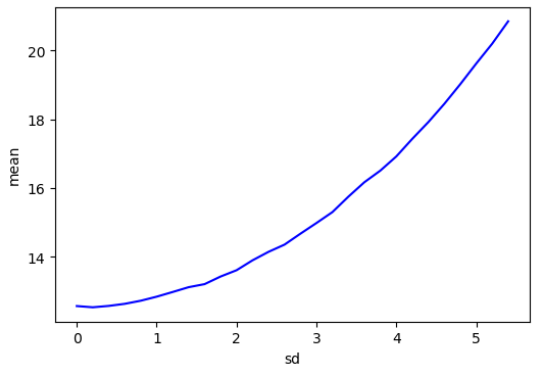


Figure 5: optimum mean for different values of s; parameter sets specifying $u_1 = 0$ path

At values of s above 5.5 where the optimum mean goes significantly beyond 20, saturation starts to become a problem and we therefore truncate the curve at 5.4. The curve from s=0.0 to s=5.4 describes a line in parameter space that generates similar spatial properties to those of the reference points.

While we can thus reproduce the regularity index of the reference points with a continuous set of possible parameters, we do note that one could also investigate other constraints to find parameters that produce 'more similar' grids. For example, we see that the optimum m for s=0 is 12.571, but plotting the reference points with a radius of 12.571/2 leads to some degree of overlap, suggesting that these are not the 'real' parameters but that there is some variation in the effective point diameter; i.e. that s is not 0.

Instead, it transpires that the minimum interpoint distance is 8.25 in the reference set of points. We can quantify how many standard deviations 8.25 is from the mean for each parameter set of our optimum-u1 line, and we find that getting dmin=8.25 is most likely for the parameter set (m=16.92, s=4.0) where it is 2.16 standard deviations from the mean. However, further investigations into more complicated similarity measure are beyond the scope of the present report.

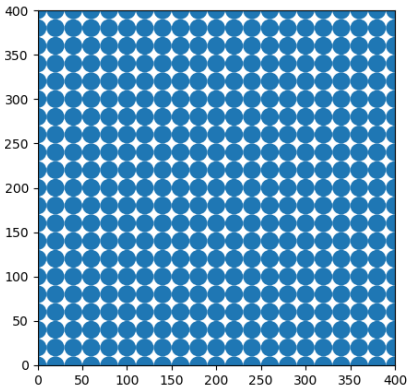
4 Packing density

We start by considering the theoretical maximum density of points in a 400x400 grid when only the center of a point has to fall in the grid (equivalent to the dmin2d model). An m of 20 in the dmin2d model with zero standard deviation corresponds to packing hard spheres of radius 10. We can imagine two simple systematic ways of doing this; using either a square packing (figure 6a) or a hexagonal packing (figure 6b), although other 2D bravais lattices also exist.

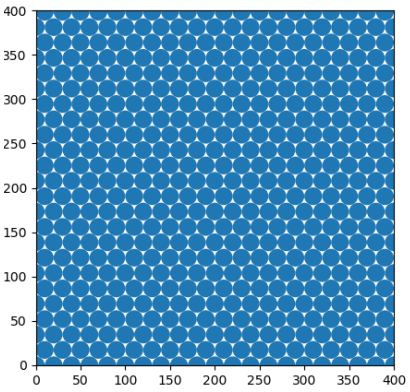
In the square packing, we can fit 400/20=20 inter-point distances and thus 21 rows of points in each dimension. This gives a total of 21*21 = 441 points.

With hexagonal packing, the separation between consecutive rows is $\sqrt{20^2 - 10^2}$. We can thus fit $floor(\frac{400}{\sqrt{300}}) = 23$ inter-row distances, corresponding to 24 rows of alternatingly 20 and 21 points in each. This gives a total of 12*(21+20) = 12*41 = 492 points. We thus find that the theoretical maximum packing density corresponds to 492 points in a 400x400 grid.

In these scenarios, the points extend beyond the 400x400 grid since the dmin2d model only requires the centers of the points to fall within the grid. The real packing density is thus given by the total area of the added points divided by the area of a grid that extends beyond the 400x400 grid by the radius of a point. This gives $\eta_{sq} = \frac{441 * 10^2 * \pi}{420^2} = 78.5\%$



(a) Square packing of 441 points in a 400x400 grid



(b) Hexagonal packing of 492 points in a 400x400 grid

Figure 6

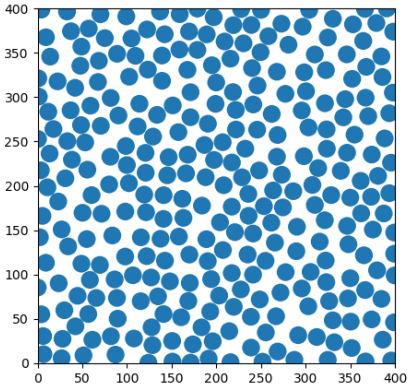
for the square packing and $\eta_{hex} = \frac{492 * 10^2 * \pi}{420^2} = 87.6\%$ for the hexagonal packing. Note that these numbers differ from conventional 2D maximum packing densities given the unusual boundary conditions.

To find an empirical measure of 'how many points can be added' with the dmin2d algorithm, we consider an attempt to generate a grid of n points to be failed if 10,000 consecutive points generated do not satisfy the dmin constraint. We then consider a particular n to be 'over-saturated' or 'unable to add n points' if we fail to generate a grid of n points in 10 consecutive trials.

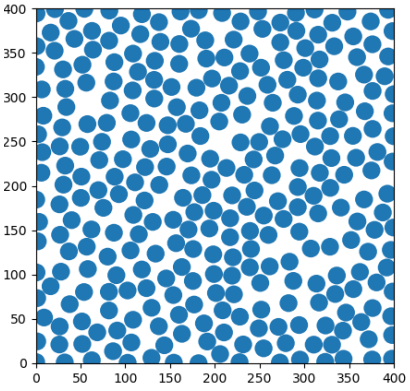
Given this measure of whether a particular n is 'too high', we write a binary search function *test_max()* with initial boundary parameters of 0 and 492 to identify n_{max}

To asses the reproducibility of this result, we run this binary search algorithm 10 times with the function *repeat_max()* and arrive at the following n_{max} values: [290, 294, 292, 290, 287, 294, 292, 289, 292, 294].
A reasonable estimate of when we cannot add more points in a dmin2d() framework is thus 294 points.

We also implement the 'birth and death' model specified in the assignment in the function *birth_death_model()*. We modify *test_max()* above to run 10 binary seaches of 'birth and death saturation', considering an attempt to generate a grid failed if we fail to add a point 10,000 consecutive times in any epoch, and an n to be saturated if we fail to generate a grid 10 consecutive times. This gives n_{max} of [300, 301, 304, 297, 302, 302, 300, 298, 302, 298].
We therefore take 302 as the maximum number of points that can be added using the birth and death model. We thus see that the birth and death model consistently packs more points than the sequential model by generating more regularly spaced points for a given n.



(a) 294 points generated by the sequential algorithm



(b) 302 points generated by the epoch algorithm

Figure 7

We plot an example of a 294-point grid generated by the dmin2d model and a 302-point grid generated by the birth

and death model in figures 7a and 7b. There appear to be small areas where an additional point could be added, suggesting that the calculated values are not hard ceilings; but they do provide an empirical estimate of what's feasible within a reasonable timeframe. The corresponding packing densities are $\eta_{dmin2d} = 52.3\%$ and $\eta_{bd} = 53.8\%$.

We also note that both the 294 points added by the dmin2d model and the 302 points added by the epoch model are very much lower than the theoretical maximum packing of 492 points. This suggests that these are not good models for simulating dense-packing systems such as crystals. However, they may be more applicable for biological systems with smaller 'penalties' for non-regularity.

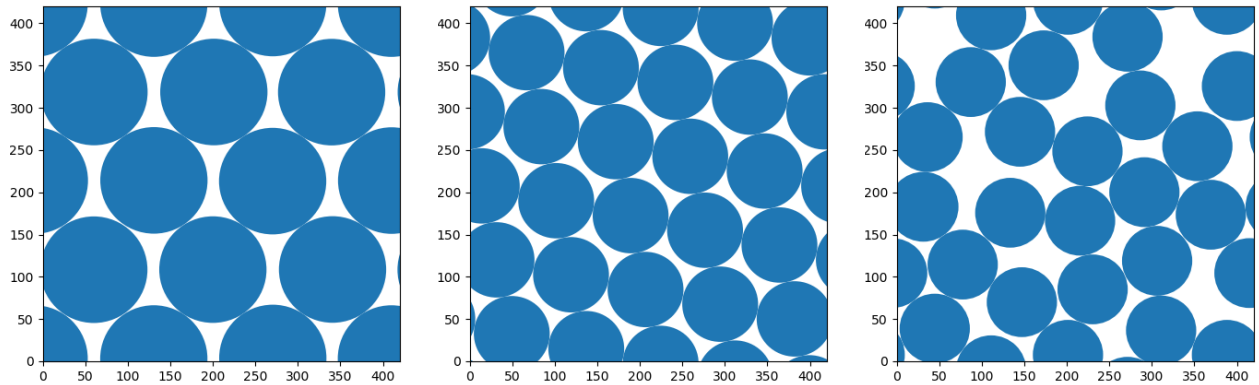
5 Moving points

We implement the Lubachevsky-Stillinger (LS) model in the function *LS_model()* as described in their 1990 paper, at every timepoint predicting the next event that will take place and updating the system accordingly. We terminate the simulation when we reach a maximum number of iterations or when the next predicted event leads to disk overlap (this can occur in jammed states given numerical inaccuracies). We also re-normalize all velocites once the mean speed of the particles exceeds 10 (this value was determined empirically). Initial velocities have components generated uniformly at random between -1 and 1, and the size of all disks grows at a constant rate such that their diameter a is given by $a(t) = a_0 t$ as described by Lubachevsky and Stillinger.

For comparison with the above results, we use a 420x420 grid for all Lubachevsky-Stillinger simulations since that is the effective space taken up by the r=10 disks in the dmin2d-generated 400x400 grid. As noted in the discussion above, it is not possible in the dmin2d model to have the center of a sphere beyond the 400x400 grid, making the present scenario slightly different. However, given the inherent differences in boundary conditions, we consider this the best approximation to a fair comparison. The overall conclusions will be invariant to small differences in the definition of packing densities.

We start by verifying the algorithm with a0=2.5 for n=12 (figure 8a) and n=24 (figure 8b) points. For n=12, this leads to a slightly elongated hexagonal packing with a final disk radius of 62.92, giving a packing density of 84.6% which is similar to the hexagonal packing density above.

For n=24, we similarly observe a rotated and slightly elongated hexagonal packing with a final disk radius of 44.59 giving a packing density of 85.0%. These values are not quite as high as the hexagonal packing density above, a consequence of the fact that we cannot achieve perfect hexagonal packing in a square grid, and the smaller disk size in figure 6b reduces the error. We note that both of the packing densities reported here are much higher than was the case for the patterns generated using dmin2d and the birth and death model, and begin to approach the upper packing density limits. When running LS simulations for small n not of the form $n = n_1 * n_2$ with $n_1 \approx n_2$, packings are generally less regular with lower packing densities as also described by Lubachevsky and Stillinger.

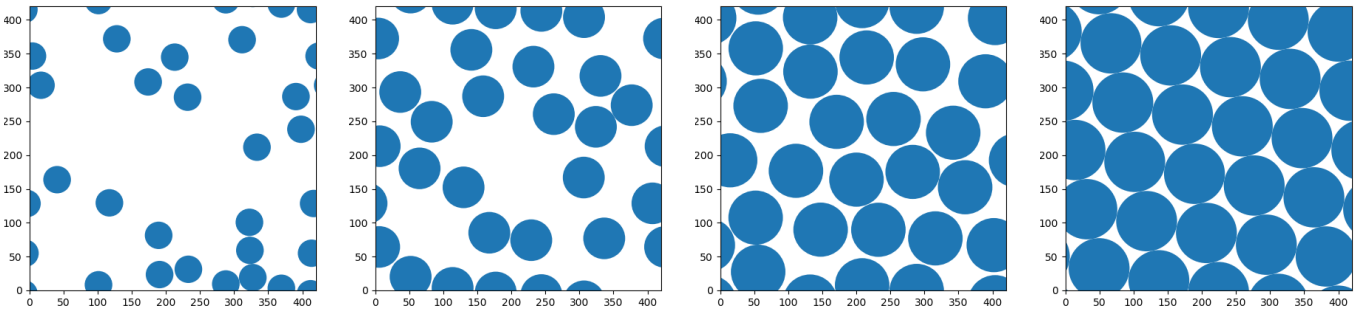


(a) n=12, a0=2.5, r=62.92 (b) n=24, a0=2.5, r=44.59 (c) n=24, a0=4.5, r=44.59

Figure 8: End-results of Lubachevsky-Stillinger simulations for different numbers of points and growth rates. Note that the grids have been plotted with periodic boundary conditions in contrast to previous plots in the report.

Similar to the observations of Lubachevsky and Stillinger, we observe that upon increasing the growth rate from 2.5 to 4.5, irregular jammed final states become more prevalent than regular packings (figure 8c). For n=24 and a0=4.5, we thus observe a final disk radius of 41.29 giving a packing density of 72.9%, which is much lower than that obtained with a smaller growth rate.

To further verify that the implementation works as expected, we can investigate in more detail the path to jammed packing in figure 8b. We thus plot the state of the simulation at four different timespoints in figures 9a-9d. We see that the points start out by being scattered at random, but as they grow and collide they form a more lattice-like structure. In figure 9c, we see the beginnings of the pseudo-hexagonal lattice that has fully formed by the end of the simulation.



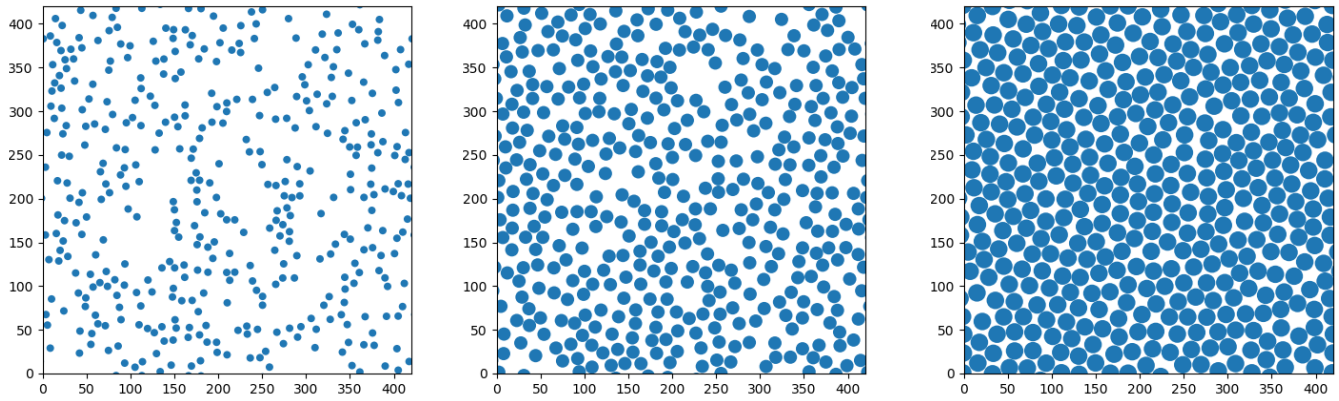
(a) pointsize 20.25 (b) pointsize 30.76 (c) pointsize 40.03 (d) pointsize 44.59

Figure 9: Freeze frames of a single Lubachevsky-Stillinger simulation with n=24 on a square lattice.

To quantify the number of radius 10 points we can fit on a grid with the LS-model compared to the dmin2d and birth and death models above, we add points to a 420x420 grid and terminate a simulation of n disks when they reach a diameter of 20 since this is equivalent to having added n points with dmin=20 to the grid.

We consider an attempt to be failed if the size of the points asymptotes before reaching 20. We use $a_0 = 1.0$ as this should lead to regular packings according to Lubachevsky & Stillinger and the simulations above.

A binary search approach similar to the one described in section 4 leads to $n_{max} = 408$, and three frames of this simulation have been plotted in figures 10a-10c.



(a) pointsize 4.3 (b) pointsize 7.6 (c) pointsize 10.0

Figure 10: different packings.

We note that for n_{max}, most of the points do not appear to be completely jammed in contrast to our observations for n=12 and n=24. This could be a result of the parameters needing further tweaking for higher n or due to numerical instabilities in the present implementation. We would imagine that if we ran the algorithm for long enough with suitable parameters (i.e. infinitesimal a0 for infinite time), we would again achieve a hexagonal packing with a packing density close to the theoretical maximum of $\frac{\pi}{2 * \sqrt{3}} = 90.7\%$ (although this is a limit as n goes to infinity for a square lattice).

The combination of growth and repulsion observed in the Lubachevsky-Stillinger Model may be more useful for modelling a range of biological systems than the dmin2d model with spontaneuous appearance of elements. For example, growth of neurons is known to involve continuous growth combined with secereton of inhibitory factors preventing excessive parallel growth of neurons.

Similarly, neuroblast development in *Drosophila* involves notch-delta-mediated lateral inhibition between nascent neuroblasts and neighboring cells, leading to a series of regularly spaced neuroblasts for subsequent generation of Ganglion Mother Cells. This could potentially also be modelled in a slightly modified Lubachevsky-Stillinger framework with the increasing disk size representing the extent of inhibition.

On an organism-wide level, we could also imagine using the Lubachevsky-Stillinger model to model growth of single celled organisms such as bacteria or yeast, with nodes representing individual colonies. In this case, we could allow for more complex interactions between points and different pointsizes.

One disadvantage of the LS model for biological systems is that it requires a fixed n which is often not the case in real systems, where the appearance or lack of appearance of an additional 'point' (e.g. cell, colony or organism) might depend on the degree of inhibition and effective size of other points. A lot of biological systems thus involve a combination of interactions between nodes, growth of nodes, and sequential addition of nodes. None of the three models considered in the present report captures all of these properties.

6 Appendix (code)

File: sp3.jl

```
#Primary file for simulations used in spa3. Calls functions implemented
#In remaining files to run simulations.

using Random, Distributions, PyCall, PyPlot, LinearAlgebra, DelimitedFiles, Plots
@pyimport matplotlib.patches as patch

#We start by loading functions from auxillary files
include("plots.jl") #functions for plotting everything
include("dmin2d.jl") #main dmin2d function
include("RI50.jl") #functions for calculating regularity indices and u values
include("saturation.jl") #functions for quantifying how many points we can add to a grid
include("LS_model.jl") #functions used in Lubachevsky-Stillinger simulations

####Now run actual code

#generate and plot grids for part I
for i in 1:2 points = dmin2d(200, 30, 5, 200, 1000, 100, 900, Print=false, Plot=true) end

RI = calc_RI(points) #test calculation of regularity index
run_sims(;N = 1000, n=200, m=0, s=0, xlo=0, xhi=1000, ylo=0, yhi=1000) #calculate RI50
results = vary_params() #investigate effect of n and shape

ref = readdlm("spa3_real.dat", Float64) #load reference points
calc_sim(10, 5, ref) #test similarity calculation
result_mat = scan_ms(ref=ref) #map u1 as a function of m and s
results = steepest_descent() #run steepest descent for optimum (m, s)
sd = test_sd() #find standard deviation of u1 for 10 simulations
plot_descent() #plot result of steepest descent and project onto u1 landscape
get_opt_means(;sds = 0:0.2:5.4) #find optimum m for for each s to minimize u1

plot_packings() #plot square and hexagonal packings
ns = repeat_max() #perform 10 bisection searches to find maximum dmin2d packing
ns = repeat_max(method="bd") #perform 10 bisection searches to find maximum birth-death packing

LS_simulation(;n=12, a0 = 2.5, xlo=0, xhi=420, ylo=0, yhi=420) #pack 12 spheres
LS_simulation(;n=24, a0 = 2.5, xlo=0, xhi=420, ylo=0, yhi=420) #pack 24 spheres
LS_simulation(;n=24, a0 = 4.5, xlo=0, xhi=420, ylo=0, yhi=420) #pack 24 spheres with high a0
repeat_max_LS() #find maximum packing with Lubachevsky-Stillinger model
```

File: dmin2d.jl

```
#function implementing dmin2d model
include("plots.jl")

function dmin2d(n, m, s, xlo, xhi, ylo, yhi; hs = false,
                Plot = false, Print = false, thresh = 10000)
  ## n: number of points to simulate
  ## m: mean of Normal distribution
  ## s: s.d. of Normal distribution
  ## xlo, xhi: possible range of X values.
  ## ylo, yhi: possible range of Y values.
```

```
  #store x, y, dmin for each Point
  points = zeros(n,3)

  check_dist = true
  if s == 0
    if m == 0
      check_dist = false #can put points arbitrarily close so may as well not check
      dmin = 0
    else
      d = [m] #julia can not draw from a normal distribution with sd 0
    end
  else
    d = Normal(m, s) #draw distances from normal distribution
  end

  i = 1; trials = 0
  while i <= n
    add = true
    coords = rand(2,1) #two floats [0:1]
    coords = coords.*[xhi-xlo, yhi-ylo]+[xlo, ylo] #convert to range
    #println(coords)
    if check_dist #only check distance if not m, s = 0, 0
      dmin = rand(d,1,1)[1]
      for j in 1:(i-1) #check distance to all other points
        if add #only check distance if we have not already found a point too close
          if norm(coords-points[j,1:2]) < max(dmin, points[j,3]) #too close to point
            add = false
            Print && println(coords, " and ", points[j,:], " too close with dmin ", dmin)
            trials += 1
            if trials > thresh
              #may be impossible/extremely unlikely to add new point so we stop
              println("Have failed to add ", thresh,
                      " points in a row. Exiting. Added ", i-1)
              return(zeros(0,0))
            end
          end
        end
      end
    end
    if add #add point to array
      if hs
        points[i, 1:3] = hcat(coords', dmin)
      else
        points[i, 1:3] = hcat(coords', 0)
      end
      i += 1
      trials = 0 #reset number of trials
    end
  end
  if Plot
    plot_points(points, pointsize = (m/2), filename="dmin",
                xlo = xlo, xhi = xhi, ylo = ylo, yhi = yhi)
  end
  return(points)
end
```

File: RI50.jl

```
#functions for calculating regularity indices and similarities
include("plots.jl")
include("dmin2d.jl")

function calc_RI(points; Print = true)
    #given an array of points, calculates regularity index as
    # mean(dmin(i))/sd(dmin(i)) (dmin(i) is distance from i to nearest neighbor)
    n = size(points)[1] #number of points
    n < 3 && return(Inf) #sd is zero in this case

    dmins = zeros(n)
    for i = 1:n #for each point, calculate distance to all other points
        point = points[i,1:2]
        dmin = Inf
```

```
        for j = vcat(1:i-1, i+1:n) #don't need self-distance. in julia, (1:0) is empty
            d = norm(point-points[j,1:2])
            if d < dmin
                dmin = d #find nearest neighbor distance
            end
        end
        dmins[i] = dmin
    end
    RI = mean(dmins)/std(dmins) #calculate RI
    Print && println("RI is ", RI, " mean ", mean(dmins), " std ", std(dmins))
    return(RI)
end

function run_sims(;N = 1000, n=200, m=0, s=0, xlo=0, xhi=1000, ylo=0, yhi=1000)
    #given a set of parameters, generates N dmin2d point arrays
    #and calculates regularity indices. Reports 50th largest value.

    RIs = zeros(N) #initialize RI vector
    for i in 1:N
        points = dmin2d(n, m, s, xlo, xhi, ylo, yhi,
                Print=false, Plot=false)
        #get points and calculate RI
        RIs[i] = calc_RI(points, Print=false)
    end
    sort!(RIs, rev=true) #sort in place
    println("50th RI ", RIs[50], " mean ", mean(RIs),
            " median ", median(RIs), " std ", std(RIs))
    return(RIs[50]) #return 5th value (95th percentile)
end

function vary_params(;ns = [20, 60, 100, 150, 200, 300, 400, 550, 700, 850, 1000],
    ratios = [1, 3, 6, 10, 15, 20, 27.5, 35, 42.5, 50], Plot=true)
    #Investigate effect of number of points and shape of grid
    #Keep grid area constant

    results = zeros(length(ns), length(ratios)) #initialize array for RI50 values

    for (i, n) in enumerate(ns)
        for (j, ratio) in enumerate(ratios)
            #consider every combination
            print("n: ", n, " ratio: ", ratio, ": ")
            #run simulation
            RI = run_sims(n=n, xhi = sqrt(10^6)*ratio, yhi=sqrt(10^6/ratio))
            results[i, j] = RI
        end
    end
    #plot and write results
    Plot && heatmap(results, ratios, ns, xlab="ratio", ylab="points",
            filename="RI50s.png", Title="variation of RI50")
    writedlm("vary_params.txt", results)
    return(results)
end


function calc_sim(m, s, ref; n = 100, xlo = 0, xhi = 400, ylo = 0, yhi = 400, Print=true)
    #calculate u-score

    L = size(ref)[1] #number of points
    RIs = zeros(n); us = zeros(n) #initialize arrays for storing RI50s and us
    RIs[1] = calc_RI(ref, Print=false) #calculate reference RI50

    for i in 2:n #generate n-1 reference grids
        points = dmin2d(L, m, s, xlo, xhi, ylo, yhi,
                    Print=false, Plot=false)
        RIs[i] = calc_RI(points, Print=false) #store RI50
    end

    for i in 1:n
        #u is magnitude of difference from RI to mean of other RIs
        u = abs( RIs[i] - 1/(n-1)*( sum(RIs[1:i-1])+sum(RIs[i+1:n]) ) )
        us[i] = u
    end
```

```
    #return results; u1 most informative
    Print && println("u1: ", us[1], " mean: ", mean(us), " sd: ", std(us))
    return us[1]
end

function scan_ms(;ms=0:0.5:21, ss=0:0.15:6.3, Plot=true, ref=ref, niter=1000)
    #preliminary investigation of the effect of mean and sd on u1 from ref

    results = zeros(length(ms), length(ss)) #initialize array for results
    result_mat = zeros(0,3) #store results as sequential array
    for (i, m) in enumerate(ms)
        for (j, s) in enumerate(ss)
            #consider every combination of mean and standard deviation
            print(m, " ", s, ": ")
            u = calc_sim(m, s, ref, n=niter)
            results[i,j]=u
            result_mat = [result_mat; [m, s, u]']
        end
    end

    #store and plot results
    writedlm("scan_results.txt", result_mat)
    writedlm("scan_result_mat.txt", results)
    Plot(&&heatmap(results, ss, ms, ylab="mean", xlab="standard deviation",
        filename="similarity.png", Title="variation of u-score")
    #report optimum parameters
    minu = findmin(results)
    println("min u is ", minu[1], " at (",
    ms[minu[2][1]], ",", ss[minu[2][2]], ")")

    return result_mat
end

function steepest_descent(; thresh=0.01, delta=0.05, rate = 1.5,
        nlim=10000, ref=ref, niter=1000, sd = "variable")
    #use a steepest descent algorithm to find the optimum set of parameters
    #use adaptive number of iterations, delta for calculating gradient
    #and step size

    params = rand(2).*[20, 6.3] #random initial parameters
    if sd != "variable" params[2] = sd end
    print("initial params ", params, " : ")
    u = calc_sim(params[1], params[2], ref, n=10)
    results = append!(copy(params), u)'
    err = thresh+1
    n = 0

    while u > thresh && n <= nlim
        niter = Int(round(max(5000*exp(-7*u), 20))) #adaptive n
        delta = u/2 #adaptive delta for calculating gradient
        rate = u/1 #adaptive learning rate
        #calculate u1 after step in m and sd
        if sd == "variable"
            us = [calc_sim(params[1]+delta, params[2], ref, Print=false, n=niter),
                calc_sim(params[1], params[2]+delta, ref, Print=false, n=niter)]
            params += ([u, u] - us)/delta*rate #update parameters
        else
            um = calc_sim(params[1]+delta, params[2], ref, Print=false, n=niter)
            params += ([u, u] - [um, u])/delta*rate #update parameters
        end
        print("new params ", round.(params, digits=3), " n=", niter, " : ")
        u = calc_sim(params[1], params[2], ref, n=niter) #calculate new u
        results = [results; append!(copy(params), u)'] #store result
        n += 1
    end
    writedlm("descent_results_temp.txt", results)
    return results
end

function test_sd(;n = 10, niter = 1000, ref=ref)
    #colculate the standard deviation of u at a given point
```

```
    #for a number of simulations
    us = zeros(10)
    for i in 1:10
        us[i] = calc_sim(12, 3, ref, n=niter)
    end
    sd = std(us)
    println("sd = ", sd) #result 0.011
    return sd
end

function get_opt_means(;sds = 0:0.2:5.4)
    means = zeros(length(sds))
    for (i, sd) in enumerate(sds)
        res = steepest_descent(sd = sd)
        means[i] = res[end, 1]
    end
    writedlm("opt_ms.txt", hcat(sds, means))
    return sds, means
end
```

File: saturation.jl

```
#functions for determining the maximum number of points
#that can be added to a grid
include("plots.jl")
include("dmin2d.jl")
include("birth_death.jl")

function saturation(n;N=10, method = "seq")
    #try constructing a grid with n points and see if it's saturated
    #consider n to be saturating if we N consecutive times fail to add a
    #single point 10,000 times

    for i in 1:N
        #try N times to add n points
        if method == "seq"
            trial = size(dmin2d(n, 20, 0, 0, 400, 0, 400, Plot=false))[1]
        else
            trial = size(birth_death_model(n, m=20, xlo=0, xhi=400, ylo=0, yhi=400))[1]
        end
        if trial == n
            return false #not saturated yet
        end
    end
    return true #failed N times in a row; saturated
end

function test_max(;nmax = 492, method = "seq")
    #perform bisection search to determine saturation limit
    #do this using the function saturation()
    #initial nmax is theoretical maximum for 400x400 grid of r=10 spheres
    nmin = 0
    while (nmax-nmin) > 1
        global n = Int(round(mean([nmin, nmax]))) #get middle value
        println("new n: ", n, "  nmin: ", nmin, "  nmax: ", nmax, " ")
        global sat = saturation(n, method=method) #check if saturated

        if sat
            nmax = n #new upper limit
            println("saturated")
        else
            nmin = n #new lower limit
            println("unsaturated")
        end
    end
    if sat
        return (n-1) #couldn't add this many points
    else
        return n #could just add this many points
    end
end

function repeat_max(N = 10; method = "seq")
```

```
    #run N bisection searches to find maximum number of points we can add
    #this allows us to look at the variation generated by this method
    ns = zeros(N)
    for i in 1:N
        ns[i] = test_max(method = method)
    end
    println("\nn's are:", ns)
    return ns
end
```

File: birth_death.jl

```
#functions for running the 'birth and death' model
include("plots.jl")

function birth_death_round(points, m, xlo, xhi, ylo, yhi)
    #I don't entirely understand what we're meant to do here...
    #pick out points in random order, remove point and try to add subject to constraints
    #hardcode a standard deviation of zero

    n = size(points)[1]
    sequence = randperm(n) #numbers 1:n in random order
    nmax = 10000 #max number points we try to add before concluding it's saturated

    for i in sequence
        ## Point i must now be killed, and a new point
        ## positioned (born) randomly subject to satisfying
        ## the minimal distance constraint.
        niter = 0
        add = false
        while (!add) & (niter < nmax)
        #keep generating new point until it satisfies constraint or we've exceeded our limit
            niter += 1
            global coords = rand(2,1).*[xhi-xlo, yhi-ylo]+[xlo, ylo] #new random coordinates
            add = true
            for j in vcat(1:(i-1), (i+1):n) #check distance to all other points
                if add
                    if norm(coords-points[j,1:2]) < m #too close to point
                        add = false #can't add; try new point
                    end
                end
            end
        end
        if add
            points[i,:] = coords' #satisfies constraint
        else
            println("Have failed to add ", nmax,
                    " points in a row. Exiting. Added ", i-1)
            return(zeros(0,0)) #failed to add a point nmax times in a row
        end
    end
    return points
end

function birth_death_model(n; m=20, xlo=0, xhi=400, ylo=0, yhi=400, nepochs = 10)
    #generate a grid of points by doing 10 consecutive rounds of
    #shuffling points subject to dmin. we hardcode here s=0
    d = [m]
    points = zeros(n,2)
    points[:,1] = (rand(n).*(xhi-xlo)).+xlo #random x coordinates
    points[:,2] = (rand(n).*(yhi-ylo)).+ylo #random y coordinates
    for epoch = 1:nepochs
        #shuffle points
        points = birth_death_round(points, m, xlo, xhi, ylo, yhi)
    end
    return points
end
```

File: LS_model.jl

```
#function for running the Lubachevsky-Stillinger model and plotting results
using Random, Distributions, PyCall, PyPlot, LinearAlgebra, DelimitedFiles
```

```
xhi = 420; yhi = 420 #specify default gridsize with global variables

function move_to_grid(state0, xlo, xhi, ylo, yhi)
    #given a point in a state, moves it onto the grid by
    #applying periodic boundary conditions
    newstate = copy(state0)
    r = newstate[1:2]; v = newstate[3:4]
    r1 = copy(r)

    #do this 'naively' by considering all directions separately
    while r1[1] < xlo
        r1[1] += (xhi-xlo)
    end
    while r1[1] > xhi
        r1[1] -= (xhi-xlo)
    end
    while r1[2] < ylo
        r1[2] += (yhi-ylo)
    end
    while r1[2] > yhi
        r1[2] -= (yhi-ylo)
    end
    newstate[1:2] = r1
    return newstate
end

function advance(state0, dt)
    #compute the state after time t ignoring collisions
    #velocity v is constant; r1 = v*(t1-t0) + r
    newstate = copy(state0)
    r=newstate[1:2]; v = newstate[3:4]
    r1 = r .+ v*dt #find new position
    newstate[1:2] = r1
    return newstate
end

function interaction_time1(state, time, k)
    #take a current time 'time' and a state,
    #return the nearest time of crossing boundary k
    #k = 1,2,3,4 ccw from right boundary
    r = state[1:2]; v = state[3:4]
    if k == 1 || k == 3
        ind = 1 #x coordinates
    else
        ind = 2 #y coordinates
    end
    if k == 1 || k ==2
        lim = xhi #right or top edge
    else
        lim = 0 #bottom edge
    end
    t = 1/v[ind] * ( lim + v[ind] * time - r[ind] ) #time to crossing
    if t > time
        return t #if happens in future that's fine
    else
        return Inf #not moving towards boundary, return Inf
    end

end

function interaction_time2(state1, state2, tstar, a0; Print = false)
    #given two states, return a list of times at which they collide
    #consider main grid and one period in either direction

    v = state2[3:4] - state1[3:4] #difference in velocity
    r20 = state2[1:2]
    Print && println("v: ", v)
    sects = [[0,0],[0,xhi],[0,-xhi],[xhi,0],[xhi,xhi],[xhi,-xhi],[-xhi,0],[-xhi,xhi],[-xhi,-xhi]]
    is = [] #store sector indices for collisions
    times = []
    for i in 1:9
        r10 = state1[1:2] + sects[i] #apply period boundary conditions to state1
```

```
        Print && println("r20: ", r20)
        Print && println("r10: ", r10)
        r = r20-r10 #difference in position
        Print && println("r: ", r)

        #solve |r+vt|^2=[a(t*)+aot]^2
        A = norm(v)^2 - a0^2
        Print && println("A: ", A)
        B = 2 * dot(r, v) - 2*a0^2*tstar
        Print && println("B: ", B)
        C = norm(r)^2-a0^2*tstar^2
        Print && println("C: ", C)

        if (B^2 - 4*A*C) > 0 #quadratic has solutions
            deltat1 = ( (-B + sqrt(B^2 - 4*A*C)) / (2*A) )
            deltat2 = ( (-B - sqrt(B^2 - 4*A*C)) / (2*A) )
            for deltat in [deltat1, deltat2]
                if deltat > -0.0001 #need event to happen in the futureallowing for numerical errors
                    Print && println("new interaction, mods: ", sects[i], " time:", tstar+deltat)
                    times = vcat(times, tstar+max(0, deltat) ) #append time of collision
                    is = vcat(is, i) #note which sector state1 is in
                end
            end
        end
    end
    Print && println("times: ", times)
    return times, is #return list of all collisions
end

function jump1(state, k, Print = false)
    #jump across edge. Given state and index of edge, returns new state
    if k == 1 || k == 3
        ind = 1 #x coordinates
    else
        ind = 2 #y coordinates
    end
    if k == 1 || k ==2
        newval = 0 #right or top edge; jump to left or bottom
    else
        newval = xhi #jump to right or top
    end
    Print && println(ind ," ", newval)
    newstate = copy(state)
    newstate[ind] = newval #update position
    return newstate
end

function jump2(state1, state2, a0, sectorind; Print = false)
    #process collision between sphere 1 and 2. Update velocities
    #need to provide information on which sector state2 is in
    sects = [[0,0],[0,xhi],[0,-xhi],[xhi,0],[xhi,xhi],[xhi,-xhi],[-xhi,0],[-xhi,xhi],[-xhi,-xhi]]
    h = a0 #h = a'(t); a(t) = a0*t

    r1, v1 = copy(state1[1:2]), copy(state1[3:4])
    r2, v2 = copy(state2[1:2]), copy(state2[3:4])
    r1 = r1 + sects[sectorind] #place in correct sector
    newstate1 = copy(state1); newstate2 = copy(state2)

    r12 = r1-r2; r12 = r12/norm(r12) #normalized inter-point vector
    Print && println("r12: ", r12)
    v1p = dot(v1, r12) * r12; v1t = v1-v1p #compose v into perpendicular and transverse components
    v2p = dot(v2, r12) * r12; v2t = v2-v2p
    Print && println("v1p: ", v1p, "  v1t: ", v1t)

    v1s = (v2p + h*r12) + v1t #transverse velocity unchanged
    v2s = (v1p - h*r12) + v2t #parallel swapped w/ additive h*(+/-)u12

    newstate1[3:4] = v1s #update velocities
    newstate2[3:4] = v2s
    return newstate1, newstate2
end
```

```
function initialize(n, xlo, xhi, ylo, yhi, Print = false)
    #initialize array of positions and velocities
    state = zeros(n, 4) #store x,y position and velocity
    for i = 1:n
        r = rand(2,1).*[xhi-xlo, yhi-ylo]+[xlo, ylo] #random positioning
        v = ( rand(2,1) .- 0.5 )*2 #initial velocities between -1 and 1 in each direction
        Print && println( hcat([[r, v]], [[r, v]]), "\n", state )
        state[i, 1:2] = r; state[i, 3:4] = v
    end
    return state
end

function LS_simulation(;n=408, a0 = 0.60, xlo=0, xhi=xhi, ylo=0, yhi=yhi, end_time=1000000,
                    maxiter = 100000, Print = false, Printall = false, nhist = 100,
                    writepoints = 200, bisection = true)
    #naive implementation scales as n^2 but given the relatively small system sizes we're working with,
    #this runs in a reasonable amount of time and is much simpler to implement. Hence
    #the preferred implementation for the present purposes.
    #a(t) = a0*t #DIAMETER GROWTH RATE
    #h = a'(t) = a0

    nroll = 0
    sects = [[0,0],[0,xhi],[0,-xhi],[xhi,0],[xhi,xhi],[xhi,-xhi],[-xhi,0],[-xhi,xhi],[-xhi,-xhi]]
    niter = 0 #number of iterations run
    current_time = 0 #start at time zero
    state = initialize(n, xlo, xhi, ylo, xhi)
    recent_events = zeros(nhist, 3) #store nhist most recent events as [i, partner, time]
    recount = 0 #keep track of where in the recent events array we are

    while (current_time <= end_time) & (niter < maxiter)
        oldstate = copy(state) #keep a copy of the old state in case we need to reverse iteration
        recount = (recount % nhist) + 1 #increment by 1
        niter += 1
        mvel = mean(abs.(state[:,3:4]))
        if mvel > 10 #reduce velocities if too high; prevent divergence
            state[:,3:4] /= (4*mvel)
            recent_events = zeros(nhist, 3)
            println("resat velocities")
        end
        Print && println("\n\nNew iteration ", niter)
        if (niter % 10 == 0)
            println("new n: ", niter, "  current size: ", current_time*a0)
            #print progress
            if niter % writepoints == 0
                #occasionally store positions of all points
                writedlm("intermediate_points/points_"*string(n)*"_"*string(niter)*
                        "_"*string(round(current_time*a0, digits=2)), state[:,1:2])
            end
        end
        Print && for i in 1:n println(i, ": ", state[i,:]) end
        collision = true; tmin = Inf; imin=NaN; partner=NaN; kmin=NaN; sectind = NaN #reset parameters
        for i in 1:n
            for j in (i+1):n
                #consider all possible pairwise collisions; find interaction times
                tcols, sectorinds = interaction_time2(state[i,:], state[j,:], current_time, a0)
                for ind in 1:length(tcols)
                    #consider collision from every sector
                    tcol = tcols[ind]; sectorind = sectorinds[ind]
                    if tcol < tmin
                        #new next event
                        add = true
                        for prev in 1:nhist
                            if i == recent_events[prev,1] && j == recent_events[prev,2] &&
                                                abs(tcol - recent_events[prev,3]) < 0.0001
                                add = false #already processed this event
                                Print && println("might have processed this event? ",
                                                    recent_events[prev,:])
                            end
                        end
                        if add
                            #store parameters
                            collision = true
```

```
                    imin = i
                    tmin = tcol
                    partner = j
                    sectind = sectorind
                    Print && println("new tmin ", tmin, " collide ", imin,
                                     " with ", partner, " sector: ", sectind)
                end
            end
        end
    end

    for k in 1:4
        #consider collisions with every edge
        tcross = interaction_time1(state[i,:], current_time, k)
        if tcross < tmin
            add = true
            for prev in 1:nhist
                if i == recent_events[prev,1] && isnan(recent_events[prev,2]) &&
                                    abs(tcross - recent_events[prev,3]) < 0.000001
                    add = false #already processed this event
                    Print && println("might have processed this event? ", recent_events[prev,:])
                end
            end
            if add
                #if new next event, store parameters
                imin = i
                collision = false
                tmin = tcross
                kmin = k
                Print && println("new tmin ", tmin, " cross ", kmin,)
            end

        end
    end
end

for i in 1:n
    state[i,:] = advance(state[i,:], tmin-current_time)
    #let simulation progress till next event
end

if collision
    Print && println("colliding ", imin, " with ", partner, " at ", tmin)
    recent_events[recount, :] = [imin, partner, tmin] #store event
    dist = norm(state[partner,1:2]-(state[imin, 1:2]+sects[sectind]))
    if round(dist) != round(a0*tmin)
        #check that the disks are in fact colliding
        println("ERROR! dists do not match. Dist: ", dist, "  a(t): ", a0*tmin)
    end
    Print && println("Distance: ", norm(state[partner,1:2]-(state[imin,1:2]+sects[sectind])),
                     "  a=", a0*tmin)
    #finally process collision by updating velocities
    state[imin,:], state[partner,:] = jump2(state[imin,:], state[partner,:], a0, sectind)
else
    #if next event is a border crossing
    Print && println(imin, " crossing ", kmin, " at ", tmin)
    recent_events[recount, :] = [imin, NaN, tmin]
end

for i in 1:n
    #move all our points to the grid using periodic boundary conditions
    state[i,:] = move_to_grid(state[i,:], xlo, xhi, ylo, yhi)
end

if !collision
    #jump border after moving to grid, otherwise numerical inaccuracies
    #can lead us to processing the same event over and over again
    state[imin,:] = jump1(state[imin,:], kmin)
    Print && println("jumped: ", state[imin,:])
end
```

```
        dmin = Inf; imin = NaN; jmin = NaN; smin = NaN
        for i in 1:n
            for j in (i+1):n
                dmini = Inf
                for sect in sects
                    #find all nearest neighbor distances
                    d = norm(state[i,1:2] + sect - state[j,1:2])
                    if d < dmini
                        dmini = d
                        if d < dmin
                            dmin = d
                            imin = i
                            jmin = j
                            smin = sect
                        end
                    end
                end
                Printall && println("distance ", i,",", j, ": ", dmini)
            end
        end
        Print && println("dmin: ", dmin, " size: ", tmin*a0, " time: ", tmin)
        #println("dmin: ", dmin, " time: ", tmin)
        if tmin*a0 > (dmin + 0.00001) #if two points are closer than their diameter, something is wrong
            #undo last simulation step and try something new
            println("rolling back event; ", imin, " and ", jmin, " too close in sector ", smin)
            println(state[imin,1:2], state[jmin,1:2])
            state = copy(oldstate) #return state to previous step
            nroll += 1
            if nroll == 50 #if we cannnot do anything without disks overlapping, simulation is saturated
                if bisection return true end
                return state, a0*current_time
            end
        else
            #nothing overlaps; update time and proceed to next event
            nroll = 0
            current_time = tmin
            if bisection
                if tmin*a0 > 20 #for binary search, exit if spheres are larger than 20
                    writedlm("intermediate_points/points_"*string(n)*"_"*string(niter)*
                                "_"*string(round(current_time*a0,digits=2)), state[:,1:2])
                    return false
                end
            end
        end
        current_time = tmin
    end

    if bisection return true end
    return state, a0*current_time
end

function plot_points_LS(points; pointsize=10, xlo=0, xhi=xhi, ylo=0, yhi=yhi,
            xlab="", ylab="", title="", filename="default.png")
    #given a list of points and some plotting parameters,
    #plots the points. Plot with periodic boundary condidions

    sects = [[0,0],[0,xhi],[0,-xhi],[xhi,0],[xhi,xhi],[xhi,-xhi],[-xhi,0],[-xhi,xhi],[-xhi,-xhi]]

    figsize = [5,5] .* [yhi-ylo, xhi-xlo] ./ mean([yhi-ylo, xhi-xlo])
    fig, ax = subplots(figsize = figsize)
    xlabel(xlab)
    ylabel(ylab)
    xlim([xlo,xhi])
    ylim([ylo,yhi])
    circles = []
    for i in 1:size(points)[1]
        for sect in sects #plot with periodic boundaries
            #plot points as circles with radius
            circles = vcat(circles,
                        matplotlib[:patches][:Circle](
                        (points[i,1]+sect[1], points[i,2]+sect[2]), radius = pointsize, facecolor="b"))
        end
```

```
    end
    p = matplotlib[:collections][:PatchCollection](circles)
    ax[:add_collection](p) #plot all points
    println("plotted")
    PyPlot.savefig(filename)
    close()

end


function bisection_LS()
    nmin, nmax = 1, 520
    #perform bisection search to determine saturation limit for LS model
    while (nmax-nmin) > 1
        global n = Int(round(mean([nmin, nmax]))) #get middle value
        println("new n: ", n, "  nmin: ", nmin, "  nmax: ", nmax, " ")
        global sat = LS_simulation( n = n ) #check if saturated

        if sat
            nmax = n #new upper limit
            println("saturated")
        else
            nmin = n #new lower limit
            println("unsaturated")
        end
    end
    if sat
        return (n-1) #couldn't add this many points
    else
        return n #could just add this many points
    end
end

function repeat_max_LS(N = 10)
    #run N bisection searches to find maximum number of points we can add
    #in LS model
    ns = zeros(N)
    for i in 1:N
        ns[i] = bisection_disks() #run bisection search
        println("new n: ", ns[i])
    end
    println("\nn's are:", ns)
    return ns
end
```