Weicheng HE

Aymane OUIJALI

Pierre-Alexandre SIMON

DM2: OPTIMIZATION

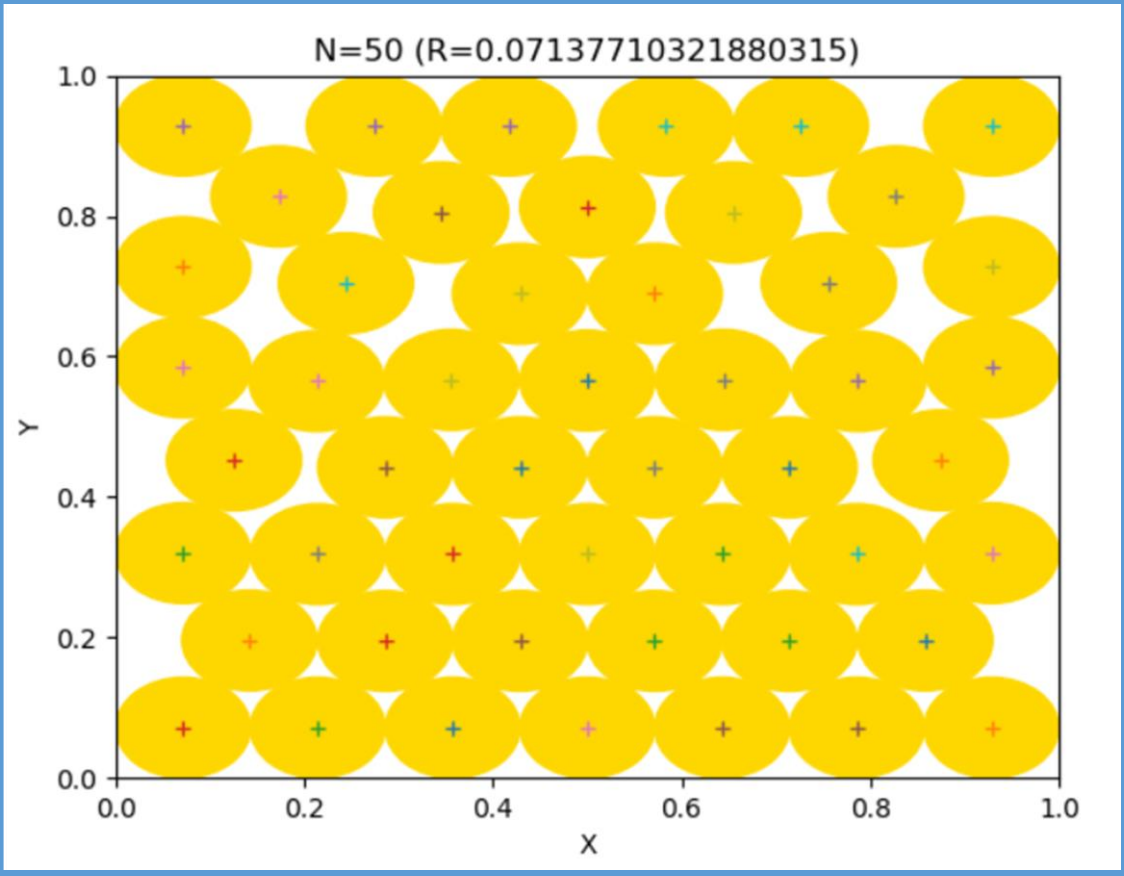


Table of Contents

Question 1 :

1) Propose a formulation

We want to pack n circles in a unit square, with the maximum possible radius.

More precisely, we have to consider the following conditions mentioned in the instructions:

- n (number of circles) is a given natural number
- r is the (variable) radius (same for all the circles)
- circles must be (fully) contained in the unit square [0, 1]2
- circles do not overlap
- we look for the maximum possible radius r

Mathematic formulation:

Let us consider two points $C_i\ (x_i, y_i)$ and $C_j\ (x_j, y_j)$ such as these latter are the center of two circles of same radius r among the n circles considered to be arranged in the unit square as depicted on the Figure 1.

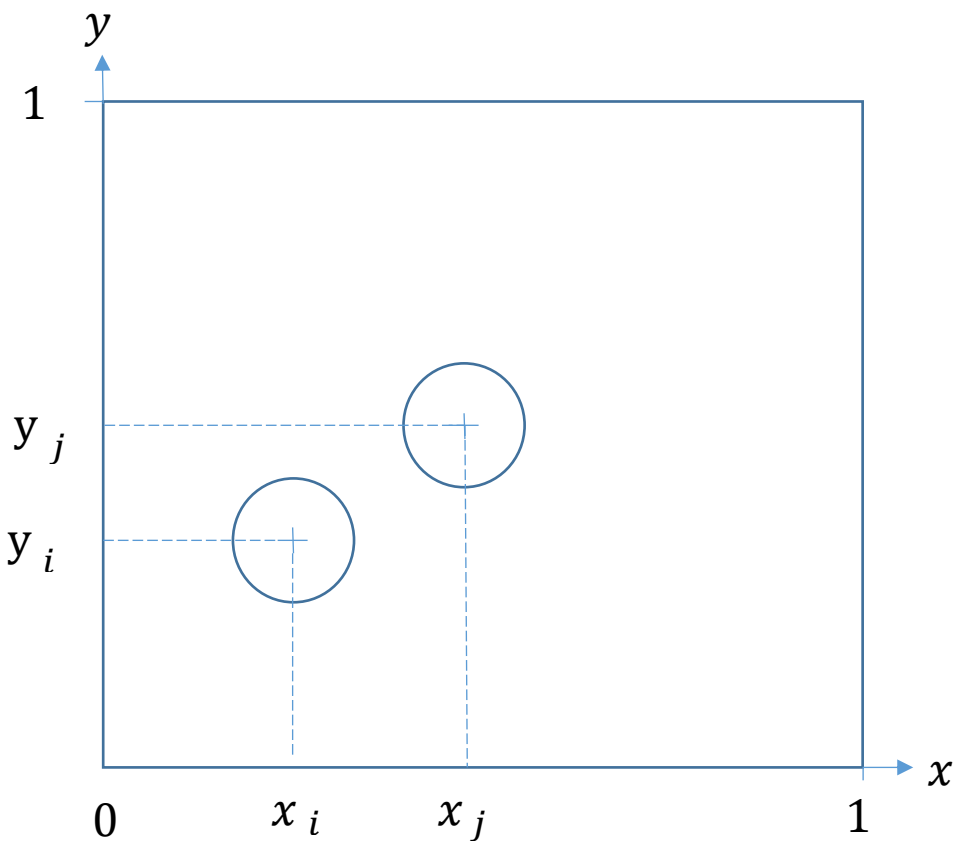


Figure 1: Draw of two circles in the unit square

We want to find the minimum radius r for n circles to be packed:

$$N = \{1, \dots, n\}$$

$$\max r \leftrightarrow \min(-r) \tag{1.1}$$

Circles should be contained in the unit square:

$$\forall\, i \in N, \qquad 1 - r \geq x_i \geq r \quad \text{and} \quad 1 - r \geq y_i \geq r \tag{1.2}$$

Circles should not overlap, so we have the following condition:

$$\forall (i,j) \in N^2, i \neq j, \qquad \sqrt{\left(x_j - x_i\right)^2 + \left(y_j - y_i\right)^2} \geq 2r \tag{1.3.1}$$

We can reformulate inequalities (1.3.1) on the form:

$$\forall (i,j) \in N^2, i \neq j, \qquad \left(x_j - x_i\right)^2 + \left(y_j - y_i\right)^2 \geq 4r^2 \tag{1.3.2}$$

Considering that the surface occupied by the n circles cannot be greater than the surface of the unit square, we have:

$$n\pi r^2 \leq 1 \tag{1.4.1}$$

$$r \leq \frac{1}{\sqrt{n\pi}} \tag{1.4.2}$$

Algorithm Formulation:

Firstly, we define an abstract model from pyomo.environ library. Then, we define a lowerbound and upperbound for the coordinates of our points to be generated (lb=0, and ub=1 as formulation above). We define as well the different variable, which will intervene in our problem respectively the number of point to be generated, x and y coordinates of the i^{th} point and the radius associated (same bounds have been applied).

```
def CirclePacking(size, lb, ub):

    #Initialisation of the model
    model = pe.AbstractModel()

    # size
    model.lb = lb
    model.ub = ub
    model.n = pe.Param(default=size)

    # set of variables, useful for sum and iterations
    model.N = pe.RangeSet(model.n)
    model.x = pe.Var(model.N,  bounds=(model.lb, model.ub))
    model.y = pe.Var(model.N,  bounds=(model.lb, model.ub))
    model.r = pe.Var(bounds=(0.0, 1.0))
```

Figure 2: Initialisation of the variables

In a second time, we define the different constraints, which applies to our variables. We define firstly the no-overlap constraint *1.3.2*: for any new point generated (new x_j and y_j), it should not be included

in previous circles of center (x_i and y_i). Finally, we define the other constraints for x_i and y_i to be included in the unit square as mentioned in equation *1.2*.

```
def no_overlap_rule(model, i, j):
    if i < j:
        return(
            (model.x[i] - model.x[j])**2
            + (model.y[i] - model.y[j])**2 >= 4*model.r**2
        )
    else:
        return pe.Constraint.Skip

model.no_overlap = pe.Constraint(model.N, model.N, rule=no_overlap_rule)

def Inside_x_min_rule(model, i):
    return model.x[i] >= model.r
model.Inside_x_min = pe.Constraint(model.N, rule=Inside_x_min_rule)

def Inside_y_min_rule(model, i):
    return model.y[i] >= model.r
model.Inside_y_min = pe.Constraint(model.N, rule=Inside_y_min_rule)

def Inside_x_max_rule(model, i):
    return model.x[i] <= 1-model.r
model.Inside_x_max = pe.Constraint(model.N, rule=Inside_x_max_rule)

def Inside_y_max_rule(model, i):
    return model.y[i] <= 1-model.r
model.Inside_y_max = pe.Constraint(model.N, rule=Inside_y_max_rule)
```

Figure 3: Definition of the constraints

Finally, we define the objective function (circles radius) as defined in equation *1.1* in the sense of maximizing as we want the biggest radius possible for our model and return the instance of the model created.

```
def radius_rule(model):
    return model.r

# then we created the objective: function and sense of optimization
model.obj = pe.Objective(rule=radius_rule, sense=pe.maximize)

model.n = size
# return instance
return model.create_instance()
```

Figure 4: Definition of the objective function

Before looking for local solver, we will retain the Monotonic Basin Hopping algorithm (hereinafter "MBH") as advised for our study. For one-dimensional problem, the algorithm is described on the Figure 6. The algorithm start generating a point anywhere in the feasible space. From this point, it generates a new point locally in a window determined (not too far from the first point generated). Then, the algorithm uses a local optimizer to find the local minimum in the neighbourhood of this new point and then compare the value of the objective function in the new point find by the local optimizer with the value on the older point. If the objective function is smaller in this new point, then this latter is considered as the best point so far and the local generation start again from this point. If not, local generation starts from the older point. In the second case, the local generation and optimization end if no better point is found after N iterations.

Algorithm 4 Monotonic Basin Hopping

```
procedure MONOTONIC BASIN HOPPING(N)
    x* = GloballyGenerate(), n = 0
    while n < N do
        z = LocallyGenerate(x*)
        z = L(z)
        if f(z) < f(x*) then
            x* = z, n = 0
        else
            n = n + 1
    return x*, f(x*)
```

Figure 6: Monotonic Basin Hopping algorithm

From this definition, we proposed a formulation of the algorithm to our circle-packing problem as decribed on the Figure 7 below.

Algorithm Monotonic Basin Hopping

1: Procedure Monotonic Basin Hopping(N)

2: $\forall i \in N,\ (x_i,\ y_i) = GloballyGenerate()$

3: $r^* = \begin{cases} max(r(x_i,\ y_i)) \\ i \in N \end{cases}$

3: ***while*** $n < N$ ***do***

4: $(x_j,\ y_j) = LocallyGenerate()$

5: $r = \begin{cases} max(r(x_j,\ y_{ij})) \\ j \in N \end{cases}$

6: ***if*** $r < r^*$ ***then***

7: $r^* = r,\ n = 0$

8: $best\_points = (x_j,\ y_j)$

9 : ***else***

10 : $n = n + 1$

return $best\_points, r^*$

Figure 7: Monotonic Basin Hopping for our circle-packing problem

This algorithm starts generating N random tuple of coordinate (x_i, y_i) anywhere in the feasible space. Then, the algorithm run a local optimization on the radius from these points. Afterwards, the algorithm start a local search by perturbing the current point's position in a determined window and running from this new state once again a local optimization on the radius (new arrangement). If the radius is higher than the previous one with old center arrangement, this latter is retained, otherwise this operation is renewed until the solution found on the radius stay stable after k iterations (max_no_improve). On the Figure 8, is depicted the working of the algorithm for a two-circle packing problem. Arrows on the corner of the square are indicating active constraint, which limit the radius size to keep increasing.

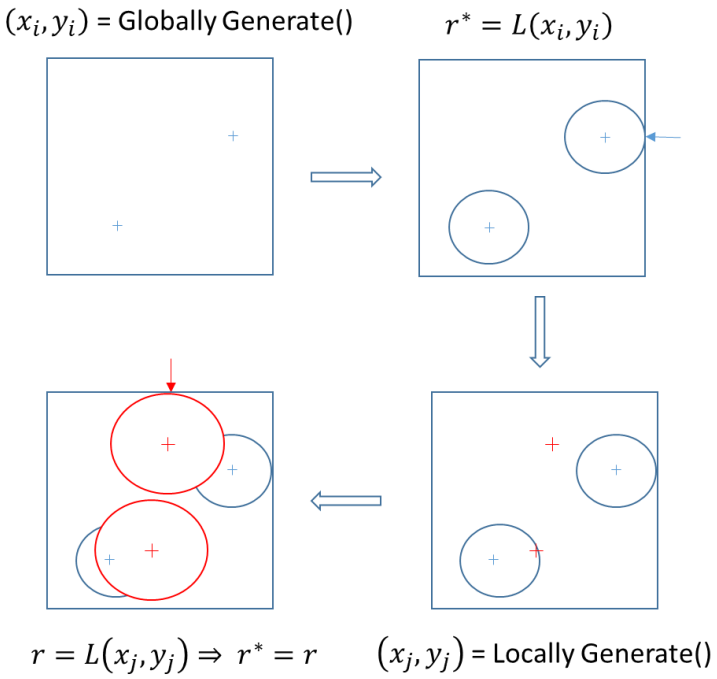


Figure 8: First steps of Monotonic Basin Hopping for two circle-packing problem

2) Determine a suitable random generation (how to generate a random point randomly)

We first define the random point's generator and call seed() method to keep the same order in the random sequence generated by the generator.

```
gen_multi = random.Random()
seed1 = 42
gen_multi.seed(seed1)
```

When using a heuristic algorithm as Multistart, it will start with a global random point generation. So, we have to define a function to generate a set of N random points. For each coordinate, we use a uniform distribution between 0 and 1 (our lowerbound and upperbound considered), which represents the bound of our problem delimited by the unit square.

```python
# random generating point keeping in [lb,ub]
def random_point(model, gen_multi):
    for i in model.N:
        model.x[i] = gen_multi.uniform(model.lb, model.ub)
        model.y[i] = gen_multi.uniform(model.lb, model.ub)
```

Figure 9: Global random point generation function

In the case of Monotonic Basin Hopping algorithm, we have a local random point generation, which allow us to decrease consequently the execution time of our algorithm. In this case, we start from points globally generated and perturb them locally. We choose a uniform distribution between -1 and 1 and configure the size of our window with the delta value such as:

$$x_i = x_i(1 + z * \delta) \ , z \sim U(-1,1)$$

$$y_i = y_i(1 + z * \delta) \ , z \sim U(-1,1)$$

Then, we applied constraints to ensure new coordinates to be inside of the unit square:

$$x_i = \max(0, \min(x_i, 1))$$

$$y_i = \max(0, \min(y_i, 1))$$

The encoded formulation is depicted on the Figure 10 below. As mentioned on comment in the code, projecting the new coordinate inside of the unit square make the perturbation capped by definition and therefore not anymore following a uniform distribution.

```python
def perturb_point(model, gen_pert, delta):
    for i in model.N:
        model.x[i] = model.x[i].value*(1+gen_pert.uniform(-1, 1) * delta)
        #project inside the box (ATTENTION: the perturbation is not anymore a uniform distribution)
        model.x[i] = max(model.lb, min(model.x[i].value, model.ub))
        model.y[i] = model.y[i].value * (1 + gen_pert.uniform(-1, 1) * delta)
        model.y[i] = max(model.lb, min(model.y[i].value, model.ub))
```

Figure 10: Local random point generation function

In a second time, we considered the perturbation mentioned in the article[1], which propose a neighbourhood structure for packing equal circles in a square. The idea is that the variable $z = x_i \ or \ y_i$ is uniformly sampled over the interval:

$$[\max\{0, \hat{z} - \xi_n\}, \min\{1, \hat{z} + \xi_n\}] \subseteq [0,1] \text{ where } \xi_n = \frac{\alpha}{\sqrt{(n)}} \ (\alpha = 0.5)$$

In addition, since the circle packing problem has another variable r, which should be perturbed in a different way with respect to the others. The author presents that the value for r has to be chosen in $[0, \bar{r}]$, where $\bar{r}$ is the minimum distance between points of the perturbed solution. However, the choice $r = 0$ turns out to be more efficient experimentally compared to $r = \bar{r}$. Because with $r = 0$, the perturbed solution lies probably in the interior of the feasible region, which gives more freedom to the local minimum search started from the perturbed solution. The implementation of this perturbation is shown below.

[1] Bernardetta Addis. A journey through optimization: from global to discrete optimization and back.. Operations Research [cs.RO]. Université de Lorraine, 2018.

```python
# perturbation
def perturb_point(model, gen_pert):
    xi=0.5/((model.n)**0.5)
    for i in model.N:
        x_lb=max(0.0,model.x[i].value-xi)
        x_ub=min(1.0,model.x[i].value+xi)
        y_lb=max(0.0,model.y[i].value-xi)
        y_ub=min(1.0,model.y[i].value+xi)
        model.x[i] = gen_pert.uniform(x_lb, x_ub)
        model.y[i] = gen_pert.uniform(y_lb, y_ub)
    model.r = 0.0
```

Figure 11: Second perturbation

After the implementation, the performance of two type of perturbation in MBH is compared. The simulation of problem size from N=5 to 50 is executed with both two perturbations respectively. The precision level and computation time are compared between each other.

From the results, we can see that in terms of precision, both two type of perturbation are at the same level. However, the second one, more simple perturbation seem to have a shorter computation time compared to the first one when the problem size gets bigger.

	N	Radius	Packomania	Ecart [%]	Time [s]
0	5	0.196438	0.207107	-5.151311	0.892622
1	10	0.147923	0.148204	-0.189917	1.691634
2	15	0.125785	0.127167	-1.086740	3.388466
3	20	0.111382	0.111382	-0.000007	6.222510
4	25	0.097221	0.100000	-2.778801	17.472818
5	30	0.089810	0.091671	-2.030343	7.469374
6	35	0.083173	0.084291	-1.326475	14.901498
7	40	0.078151	0.079187	-1.307655	13.157920
8	45	0.072553	0.074727	-2.909736	24.685288
9	50	0.069935	0.071377	-2.019763	36.013012

	N	Radius	Packomania	Ecart [%]	Time [s]
0	5	0.196438	0.207107	-5.151311	1.157242
1	10	0.147923	0.148204	-0.189917	3.947984
2	15	0.125785	0.127167	-1.086740	3.345964
3	20	0.111382	0.111382	-0.000007	4.942320
4	25	0.097221	0.100000	-2.778801	10.989604
5	30	0.089810	0.091671	-2.030343	11.293986
6	35	0.083173	0.084291	-1.326475	15.723482
7	40	0.078151	0.079187	-1.307655	19.051344
8	45	0.072553	0.074727	-2.909736	22.457816
9	50	0.069935	0.071377	-2.019763	28.755540

Figure 12: Performance of two perturbations (left: the first method, right: the second method)

3) Choose a "good" local solver (try knitro, minos, snopt, ...)

In this section, we will try different solvers to assess which one is performing the best for our maximization problem. In a first instance, let place us in a particular case where there is no obvious solution (not low number of circle to be packed and even number as for uneven number, disposition is symmetric along the axis). Therefore, we choose 10 circles to be packed, and observe the results obtained for knitro et snopt optimizers aforementioned in the Figure 13. As the Packomania results are given with a lot of decimal, we will consider a new radius as relevant if its value is higher than 10^{-8} the previous best radius found.

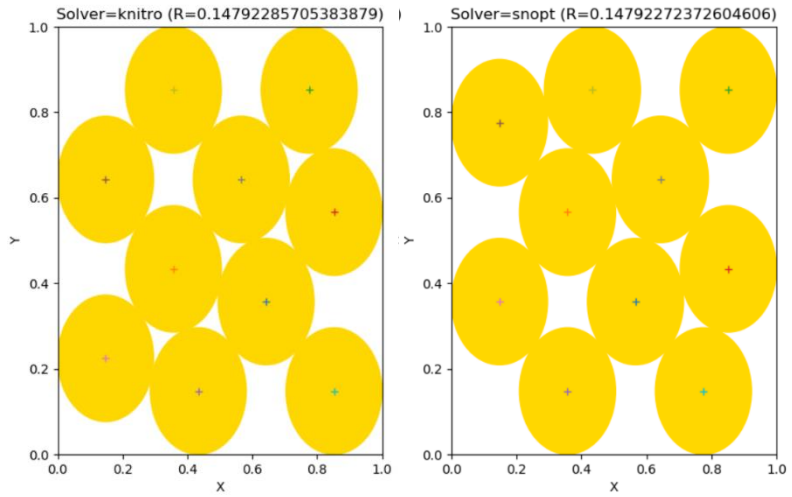


Figure 13: Packing using MBH algorithm and different solvers for 10 Circles

We can observe that the solver knitro perform better than snopt on this particular case of packing 10 circles in the unit square. In a second time, we perform the optimization on 100 consecutive cases with increasing number of circles to compare respectively the evolution of their performance when the problem become more and more complex. We observe the result on the Figure 14. We observe that both solvers are behaving quite similarly when N increase. On lower amount of circles to be packed, we see that knitro seems to obtain smoother radius than snopt. For this reason, we will retain knitro for the rest of our study.

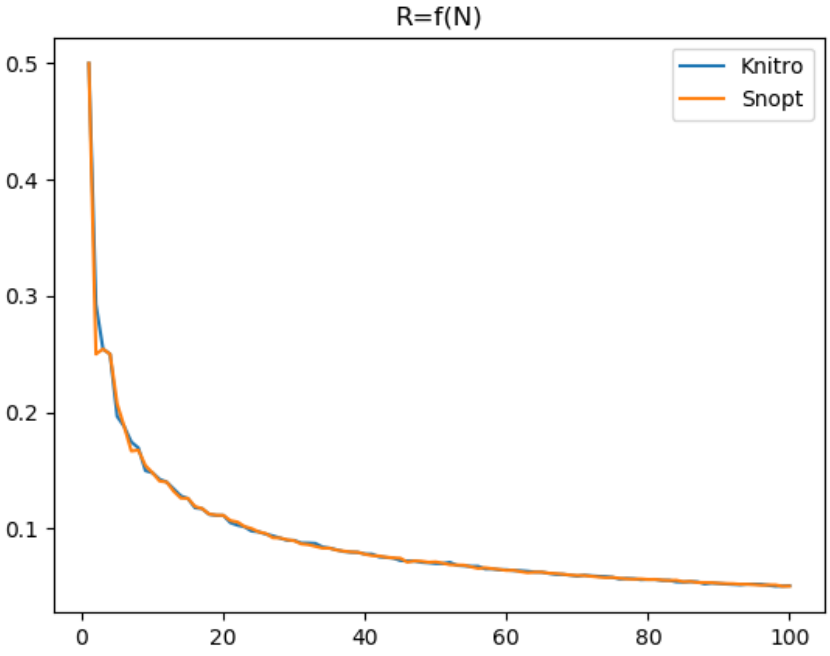


Figure 14: Performance on maximizing radius for knitro and snopt solvers

4) Determine a good neighbourhood (how and which variables must be perturbed?)

In this section, we try empirically several neighbourhood ranges using MBH. We use the local solver knitro and we set the number of circle to be packed to 10 for the same reason as mentioned previously.

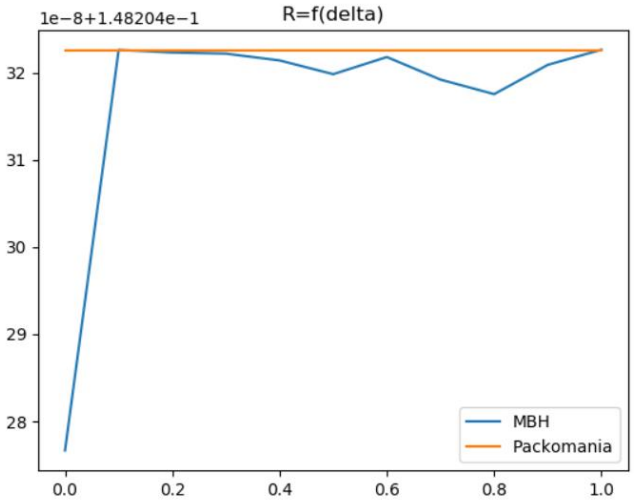


Figure 15: Radius obtained using MBH algorithm and different local generation ranges for 10 circles packing problem

As depicted in the Figure 15 above, we can observe that when delta is set to zero, radius found is quite bad in comparison with packomania results. Indeed, this means that the window is reduced to the each single point generated during the global generation. The points stay stuck in this position and do not move anymore. We can observe a severe drop from $\delta = 0.5$ in the radius values. This may be explained as well as a particular case taking a random seed as generator, which generates random points in a certain sequence which is not profitable for this 10 circles packing problem. Another reason may be the random range we took between -1 and 1. When $\delta > 0.5$, this means that above this value, the points generated are often outside the unit square and need to be projected again inside leading to similar values for the center of the circles, which finally do not allow them to move efficiently locally inside of the unit square. To avoid the first reason, we implement a boostrap of 100 samples generated randomly and calculate the mean of the radius obtained. The results are displayed on the Figure 16.

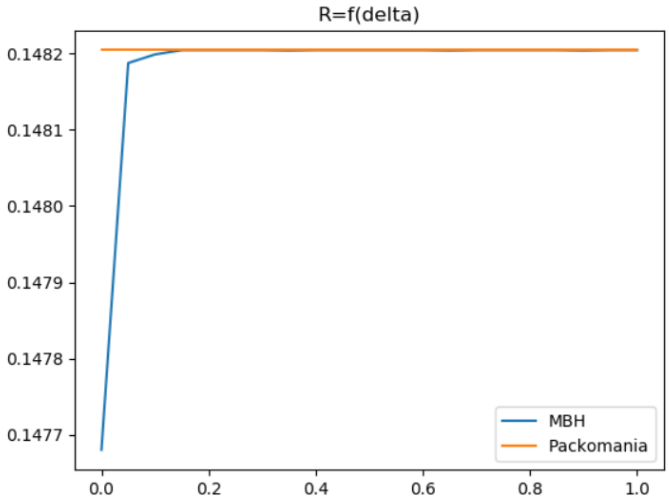


Figure 16: Radius obtained by bootstrap using MBH algorithm and different local generation ranges

We can see that the several drops in the radius obtained after delta>5 disappears. This make sense as with the bootstrap method, we generated for each value of delta, 100 different sequences of initial random points, and then proceed up to n*2=20 iterations to find a better radius. This makes 2000 maximum different points configurations possible in the feasible space, which increase therefore the probability to get to the best configuration and radius. If we reiterate the first method and decrease the number of iteration for our 10 circles packing problem, we observe that the best radius found become worst for delta>0.5 in comparison with the first simulation as depicted on the Figure 17. As our objective it to increase the complexity of the problem and obtain solution as quick as possible, we will consider a value for delta higher than 0,2 and lower than 0,5, where the value of radius found is stable in the Figure 15 previously. Then, we will set the value for delta to 0,3 for the rest of our study.

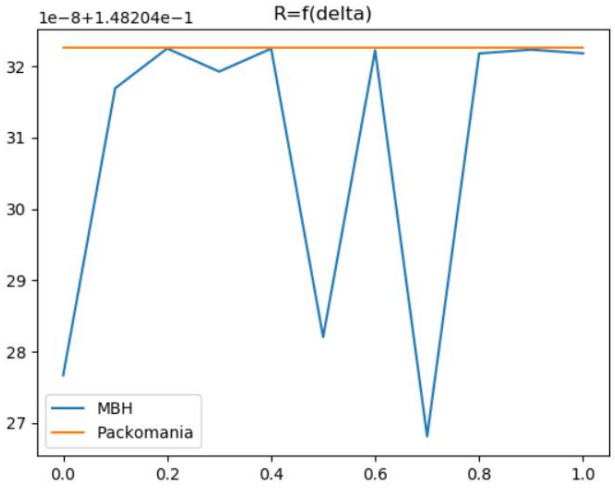


Figure 17: Radius obtained by bootstrap using MBH algorithm and different local generation ranges

5) Test your algorithm against Multistart (which one gives better results, under which conditions?)

We tried to optimize our radius with respect to Circle Packing problem with Multistart algorithm. Until N=4, Multistart algorithm find the good shape of the best arrangement of circles within the unit square, characterized by a small difference between the Packomania results as depicted on the Figure 18 below. From N=5, Multistart failed to find the best arrangement in comparison with Packomania results. We can observe considerable difference in term of radius return by Multistart, characterized by important empty space between circles as displayed on the Appendix 2. We may explain this difference by the way the Multistart is behaving. Indeed, it generates randomly points over the unit square and optimize the radius with a local optimizer. Although we set up the maximum number of iterations to $n * 100$ for Multistart against $n * 2$ for MBH, Multistart algorithm performs worst as MBH. As the complexity of the problem is small (low amount of circles to be packed), both algorithm are perform similarly. As the number of circles become more important, we can see that a noticeable difference between radius return between the two algorithms (Figure 19).

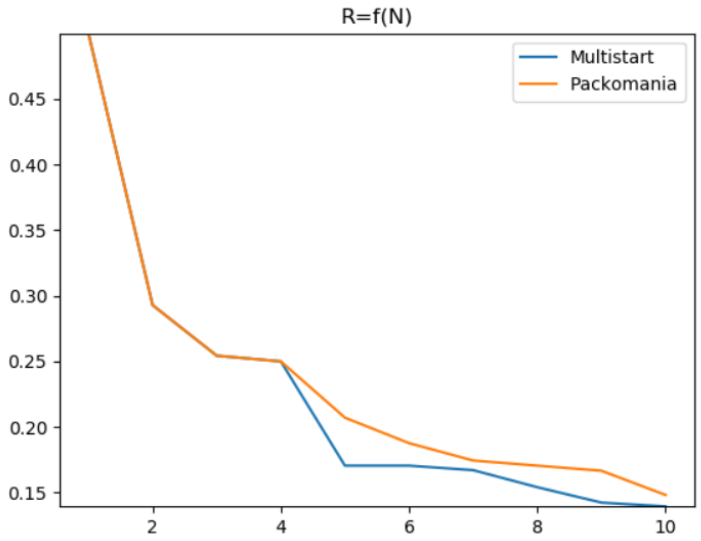


Figure 18: Comparison between Radius optimized with Multistart and Packomania

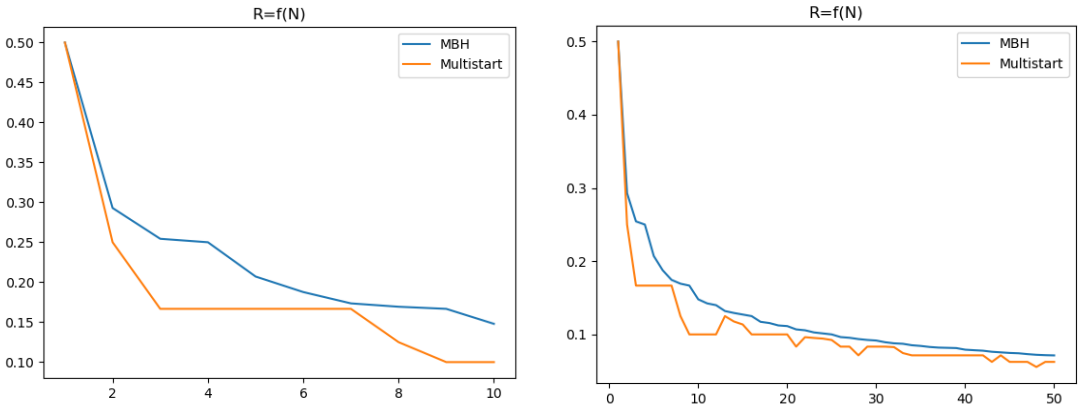


Figure 19: MBH against Multistart with N increasing

Moreover, we add the PMBH algorithm to the comparison and perform some comparison considering difference in radius returned and computation time between Multistart, MBH and PMBH. All of three algorithms seem efficient for this circle packing problem, but a further understanding on their performance is needed. So we conducted a comparison on the same conditions. The simulation is based on a complexity going from 5 to 50 by step 5. The algorithm Multistart has a number of iteration equal to N*5 (N is the problem size) while the stopping criterion of MBH is set to 30 to ensure both algorithms have the similar computation time. As for PMBH, since it is more time-comsuming, it's not very fair to compare it with the other two algorithms during the same computation time. Our aim here is to find if PMBH can get a better results than others without computation time limit.

Multistart (iter=i*5)

N	Radius	Packomania	Ecart [%]	Time [s]
5	0.207107	0.207107	-1.498550e-08	0.342204
10	0.148204	0.148204	-4.259982e-08	1.266822
15	0.127167	0.127167	-1.041900e-05	3.330290
20	0.111382	0.111382	3.578231e-08	6.732796
25	0.100000	0.100000	-1.012220e-05	12.102638
30	0.091671	0.091671	-3.107939e-07	20.116750
35	0.084087	0.084291	-2.413844e-01	31.465940
40	0.079187	0.079187	-2.623562e-07	46.809172
45	0.074727	0.074727	-6.233411e-05	68.775976
50	0.071308	0.071377	-9.676630e-02	91.926364

MBH (max_no_improve = 50)

N	Radius	Packomania	Ecart [%]	Time [s]
5	0.196438	0.207107	-5.151311	0.889052
10	0.147923	0.148204	-0.189917	1.633720
15	0.125785	0.127167	-1.086740	2.641610
20	0.111382	0.111382	-0.000007	4.243456
25	0.097221	0.100000	-2.778801	14.534146
30	0.089810	0.091671	-2.030343	7.079074
35	0.083173	0.084291	-1.326475	14.182500
40	0.078151	0.079187	-1.307655	12.677104
45	0.072553	0.074727	-2.909736	21.903590
50	0.069935	0.071377	-2.019763	33.925148

MBH_MultiTrial (iter=I*5 max_no_improve = 50)

N	Radius	Packomania	Ecart [%]	Time [s]
5	0.207107	0.207107	7.460149e-05	9.876088
10	0.148204	0.148204	3.143931e-08	21.798012
15	0.127167	0.127167	1.452086e-08	35.315872
20	0.111382	0.111382	2.157088e-08	57.447594
25	0.100000	0.100000	-4.488791e-08	92.528166
30	0.091671	0.091671	2.347728e-08	111.185452
35	0.084291	0.084291	2.458704e-08	154.502056
40	0.079187	0.079187	-2.623562e-07	210.063992
45	0.074727	0.074727	-2.502676e-05	247.584890
50	0.071377	0.071377	-2.236098e-05	311.757472

Figure 20 : Obtained results for three algorithms (left: MultiStart, middle: MBH, right: PMBH)

According to the obtained results, we observed that Multistart has a high precision up to $5 \times 10^{-8}\%$ (when N is lower than 10), much better than MBH which has always an error more than 1%. However, the computation time of Multistart increases considerably with the increase of problem size. Regarding PMBH, with the same number of iteration as Multistart, its precision is very close to that of Multistart when N is below 40 but its computation time is five times longer. Moreover, when the problem size is quiet big, like 50, PMBH can achieve a higher precision than MultiStart, which makes sense considering that fact that PMBH shares the advantages of both Multistart and MBH theoretically.

Question 2:

Packing n spheres in the unit cube.
Which is the largest size (n) of problem that you can you solve?

This problem is quite similar to the circle packing in the unit square. We add a new dimension z and we add the same constraints to this new dimension to keep the sphere in the unit cube as well as avoiding overlap as described below.

We want to find the minimum radius r for n spheres to be packed:

$$N = \{1, \dots, n\}$$

$$\max r \tag{2.1}$$

Spheres should be contained in the unit square:

$$\forall\, i \in N, \quad 1 - r \geq x_i \geq r,\ \ 1 - r \geq y_i \geq r \ and\ \ 1 - r \geq z_i \geq r \tag{2.2}$$

Spheres should not overlap, so we have the following condition:

$$\forall (i,j) \in N^2, i \neq j, \quad \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2} \geq 2r \tag{2.3.1}$$

We can reformulate inequalities (2.3.1) on the form:

$$\forall (i,j) \in N^2, i \neq j, \quad (x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2 \geq 4r^2 \tag{2.3.2}$$

Considering that the surface occupied by the n spheres cannot be greater than the volume of the unit cube, we have:

$$\frac{4n\pi r^3}{3} \leq 1 \tag{2.4.1}$$

$$r \leq \sqrt[3]{\frac{3}{4n\pi}} \tag{2.4.2}$$

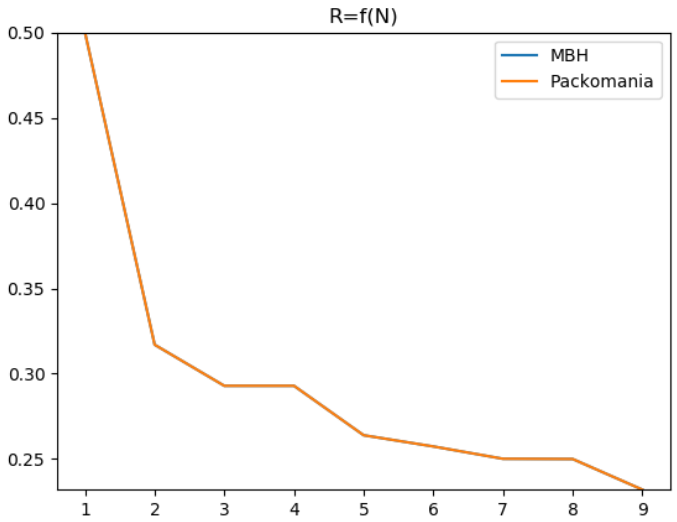


Figure 21: Comparison between MBH and Packomania results for sphere packing problem and n<10

Implementing the code, we represents graphically the packing of the different cases up to 9 spheres in the Appendix 5. The radius for these problems is displayed on the Figure 21 above. We represented as well the results up to 50 spheres on the Figure 22. We can see overall that MBH give very good results on finding optimal radius and even when the complexity become larger, with only small difference with Packomania optimal results.

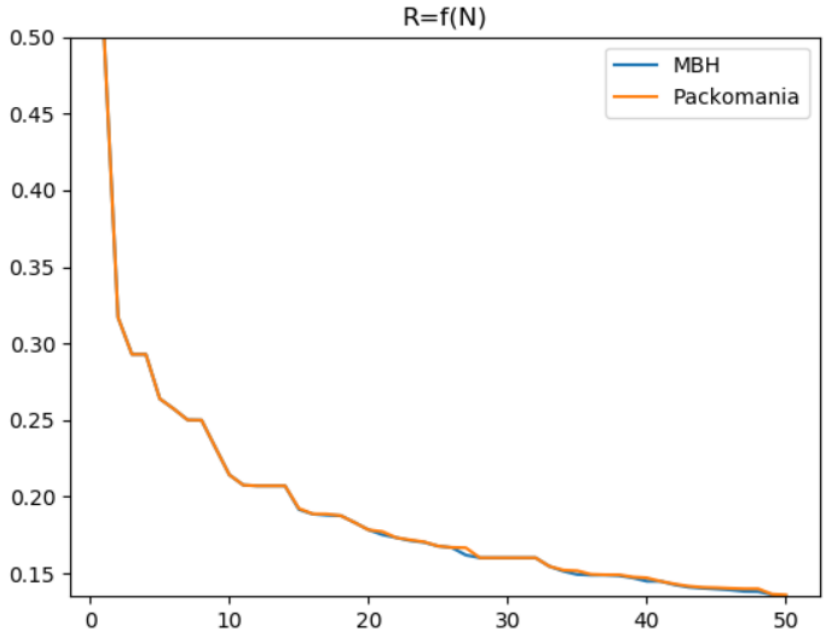


Figure 22: Comparison between MBH and Packomania results for sphere packing and n<50

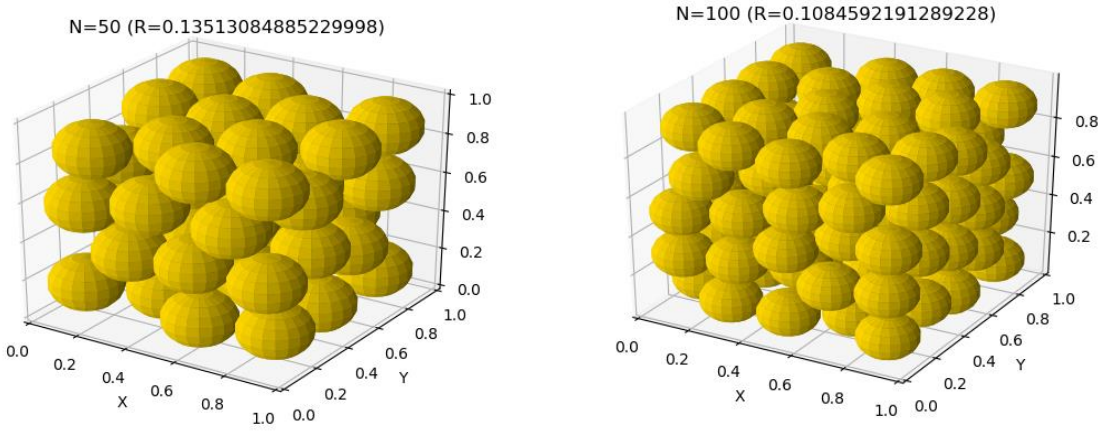


Figure 23: Sphere packing for N=50 and N=100 with MBH algorithm and knitro solver

We considered as well sphere packing for N=50 and N=100 as displayed on Figure 23. As the radius is given by Packomania with a lot of decimal, we considered the error in percentage.

	N=50	N=100
R_found	0,135130848852299	0,108459219128922
R_pack	0,135954512641663	0,111382347512479
Difference	0,61%	2,70%

We can observe that the difference is lower than 1% up to N=50 but increase up to roughly 3% for the double of sphere to be packed. We can see that as the complexity of the problem is increasing, the processing time required explode exponentially (Figure 24).

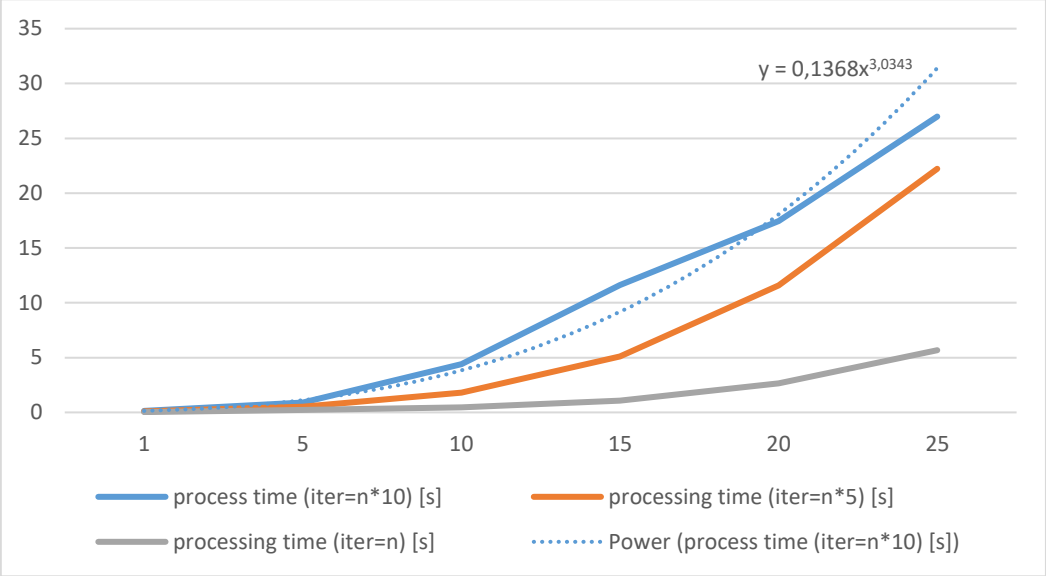


Figure 24: Processing time of MBH with number iteration=n*{1, 5, 10}

We can see on Packomania website that an optimal radius has been found until **1,024,192** spheres. By fitting on excel the processing time with an exponential law for n*10 iterations, we found that the resolution of this problem would take $0.1368 * 1{,}024{,}192^{3.0343} = 2.3626 * 10^{17} s = 6{,}56 * 10^{13} h = 2.73 * 10^{12} days = 7{,}491{,}688{,}466\ years$! To relativize this result, we suppose that researchers may use less iterations, better processors and different machines in clusters allowing this huge amount of time to be reduced to several weeks or months at maximum.

New formulation:

In this part, we implemented the method of Lopez and Beasley described in Source (2). They propose a formulation using both Cartesian and polar coordinate for the center of the circles allowing more flexibility when solving the problem. We modifiy their formulation to adapt it to our circle-packing problem in the unit square.

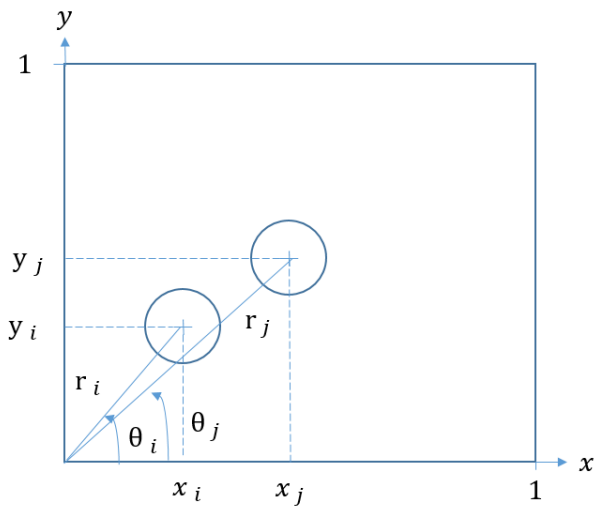


Figure 25: Draw of two circles in the unit square

Mathematic formulation:

Max R

Subject to :

$$(x_i - x_j)^2 + (y_i - y_j)^2 \geq 4R^2 \quad \forall (i,j) \in Q \text{ with } i \in C \quad j \in C \quad i < j$$

$$(x_i - r_j \cos(\theta_j))^2 + (y_i - r_j \sin(\theta_j))^2 \geq 4R^2 \quad \forall (i,j) \in Q \text{ with } i \in C \quad j \in P,$$

$$r_i^2 + r_j^2 - 2 r_i r_j \cos(\theta_i - \theta_j) \geq 4R^2 \quad \forall (i,j) \in Q \text{ with } i \in P \quad j \in P \quad i < j,$$

$$0 \leq x_i \leq 1 \quad \forall i \in C,$$

$$0 \leq y_i \leq 1 \quad \forall i \in C,$$

$$0 \leq r_i \leq 1 \quad \forall i \in P,$$

$$0 \leq \theta_j \leq \frac{\pi}{2} \quad \forall i \in P,$$

$$0 \leq R \leq Roverlap$$

In a first instance, the coordinate of the first $\frac{n(n+1)}{2}$ circles are expressed in Polar coordinates as the $n - \frac{n(n+1)}{2}$ other circles have their coordinates expressed in Cartesian coordinates. Specifics constraints are defined for Polar and Cartesian coordinates.

In a second time, as the flexibility remain in the fact and making the range of indices varying in Polar and Cartesian respectively, we implement this idea using the method $mutable = True$ in the definition of the index.

```
# size
model.lb = lb
model.ub = ub
model.n = pe.Param(default=size)
model.c = pe.Param(default=int(round(size/2,0)), mutable=True)
model.p = pe.Param(default=model.n-model.c, mutable=True)
model.r_inf=0.3

model.R_overlap=1/(size*pi)**(1/2)

# set of variables, useful for sum and iterations
model.N = pe.RangeSet(model.n)
model.C= pe.RangeSet(1, model.c)
model.P = pe.RangeSet(model.c+1, model.n)

model.x = pe.Var(model.N,  bounds=(model.lb, model.ub))
model.y = pe.Var(model.N,  bounds=(model.lb, model.ub))
model.R = pe.Var(bounds=(0, 1))
model.r = pe.Var(model.N, bounds=(0, 1))
model.theta=pe.Var(model.N, bounds=(0.0, pi/2))
```

Figure 25: Definition of the new instance of the model

At the beginning of each local perturbation, a new range are allocated for Polar and Cartesian circles in order to perform the optimization using both expression for each circles of the total range n as described in the below encoding.

```
model.c= gen_multi.randint(1, model.n)
model.p.value = model.n.value - model.c.value
model.C = pe.RangeSet(1, model.c)
model.P = pe.RangeSet(model.c + 1, model.n)
```

Figure 26: Dynamically changes of the range of the circles in polar/cartesian coordinates

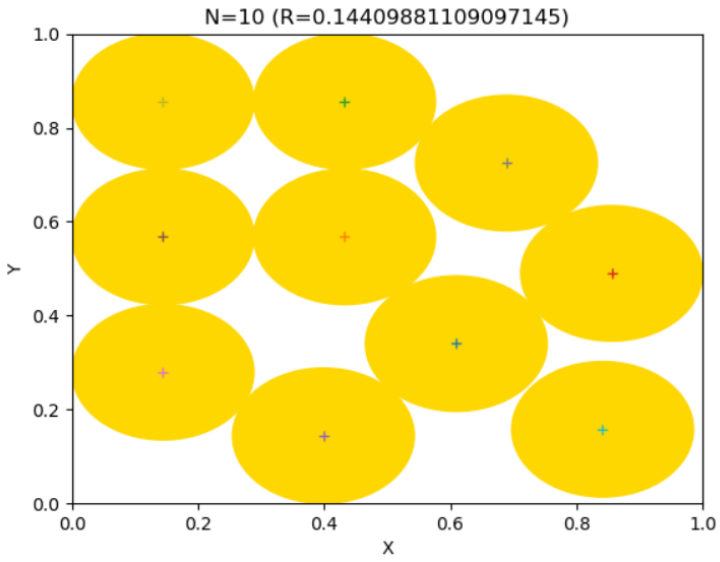


Figure 27: Packing for n=10 using new formulation

We can observe on the figure above (Figure 27) that the new formulation make the packing less efficient with the first method implemented in Exercice 1. Indeed, as we restrained the polar radius to be smaller than 1, the upper right area is not covered by the circles defined in polar coordinates. The maximum polar radius should be allow to be at maximum equal to $\sqrt{2} = 1.41$. Modifying accordingly the range of r_i, we obtained a much better result as displayed on the Figure 28 below.

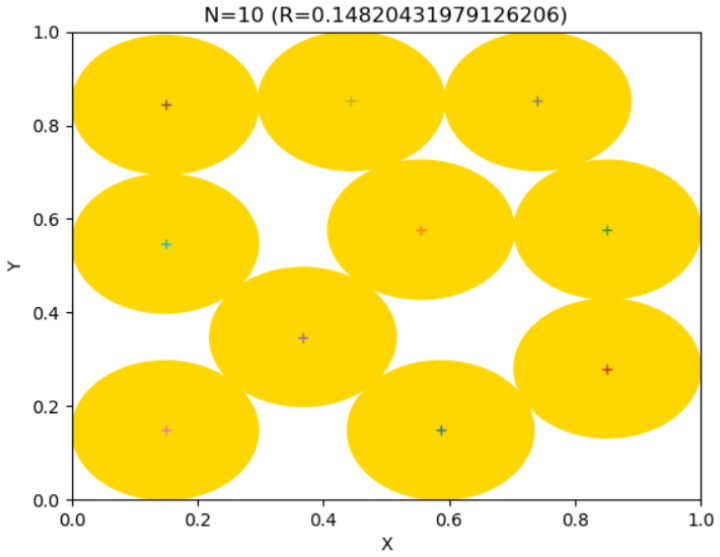


Figure 28: Packing for n=10 using new formulation with corrected polar radius

In a second time, we consider a different container as the surface contained between the unit circle and a circle of radius R_min as described in the Paper of Lopez and Beasley. We can describe the problem graphically as below:

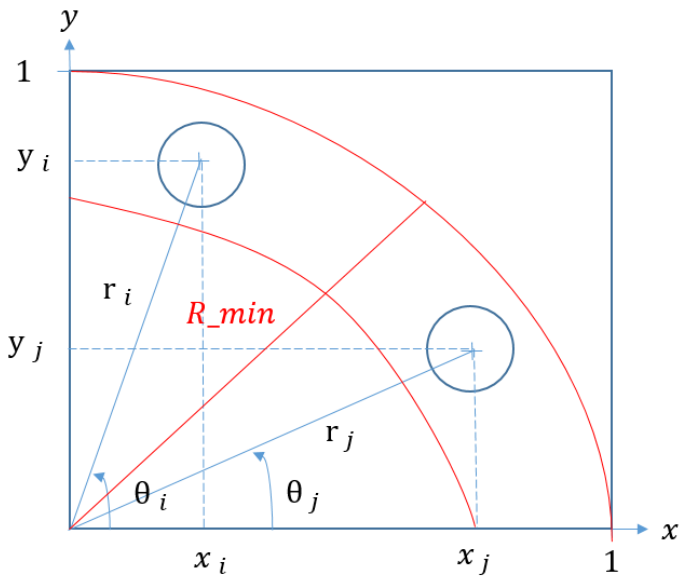


Figure 29: Packing circles between unit circle and circle of radius R_min

The mathematic formulation is given by:

$$x_i^2 + y_i^2 \leq (1 - R)^2$$

$$r_i \leq 1 - R \quad \forall\, i \in P$$

$$(x_i - x_j)^2 + (y_i - y_j)^2 \geq 4R^2 \quad \forall (i,j) \in Q \text{ with } i \in C \quad j \in C \quad i < j$$

$$(x_i - r_j \cos(\theta_j))^2 + (y_i - r_j \sin(\theta_j))^2 \geq 4R^2 \quad \forall\, (i,j) \in Q \text{ with } i \in C \quad j \in P\,,$$

$$r_i^2 + r_j^2 - 2 r_i r_j \cos(\theta_i - \theta_j) \geq 4R^2 \quad \forall (i,j) \in Q \text{ with } i \in P \quad j \in P \quad i < j\,,$$

$$R\_\min + R \leq x_i \leq 1 \quad \forall\, i \in C\,,$$

$$R\_\min + R \leq y_i \leq 1 \quad \forall\, i \in C\,,$$

$$R\_\min + R \leq r_i \leq 1 - R \quad \forall\, i \in P\,,$$

$$0 \leq \theta_j \leq \frac{\pi}{2} \quad \forall\, i \in P\,,$$

$$0 \leq R \leq Roverlap$$

We retrieve the more generic formulation of the thesis, we limited the angle of polar circles to be between 0 and $\frac{\pi}{2}$ to keep our initial square representation. We implement the algorithm for packing between a circle of radius 0.3 and the unit radius and we displayed try different complexity of circle packing up to 50 (Figure 30). We can observe that the solution returned is quite optimal as we can observe almost no space remaining between circles.

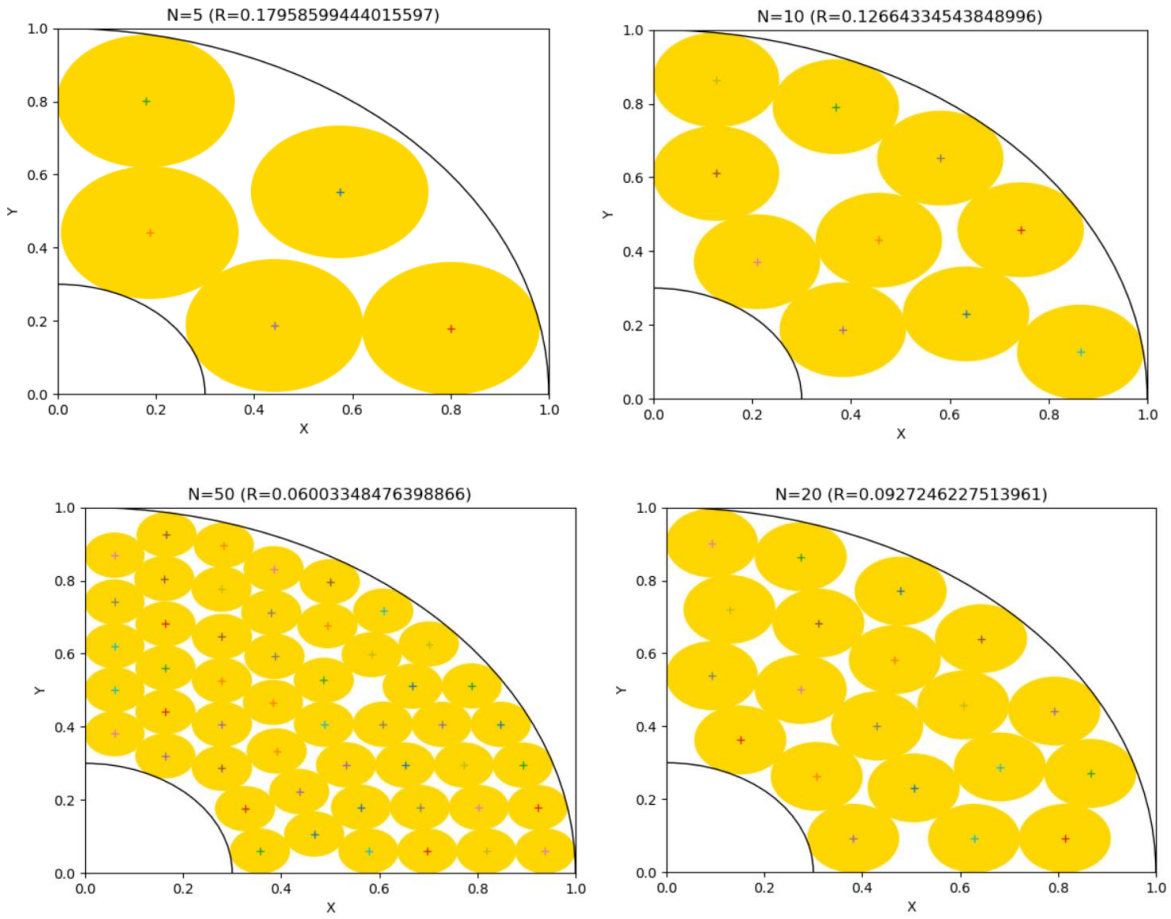
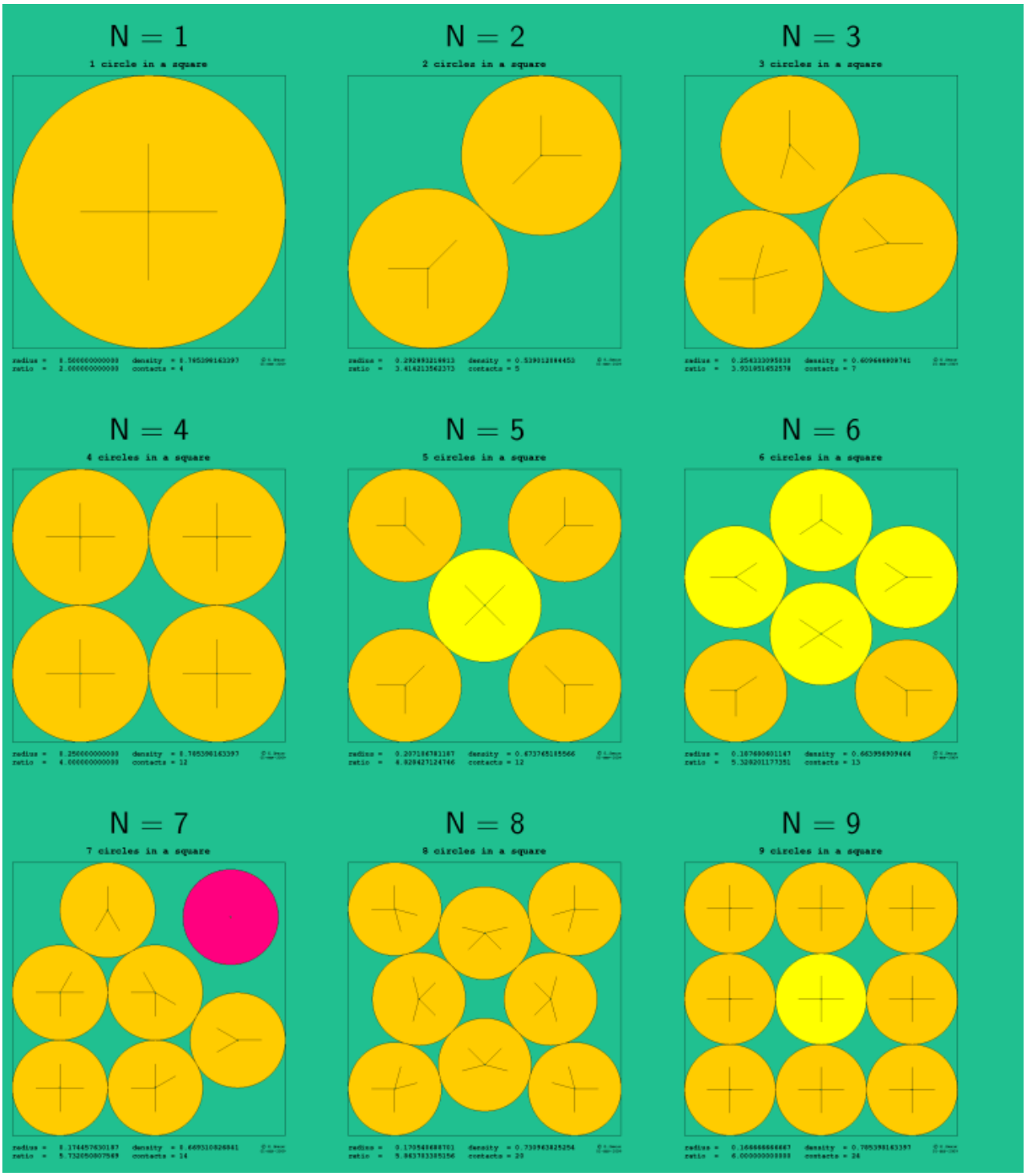
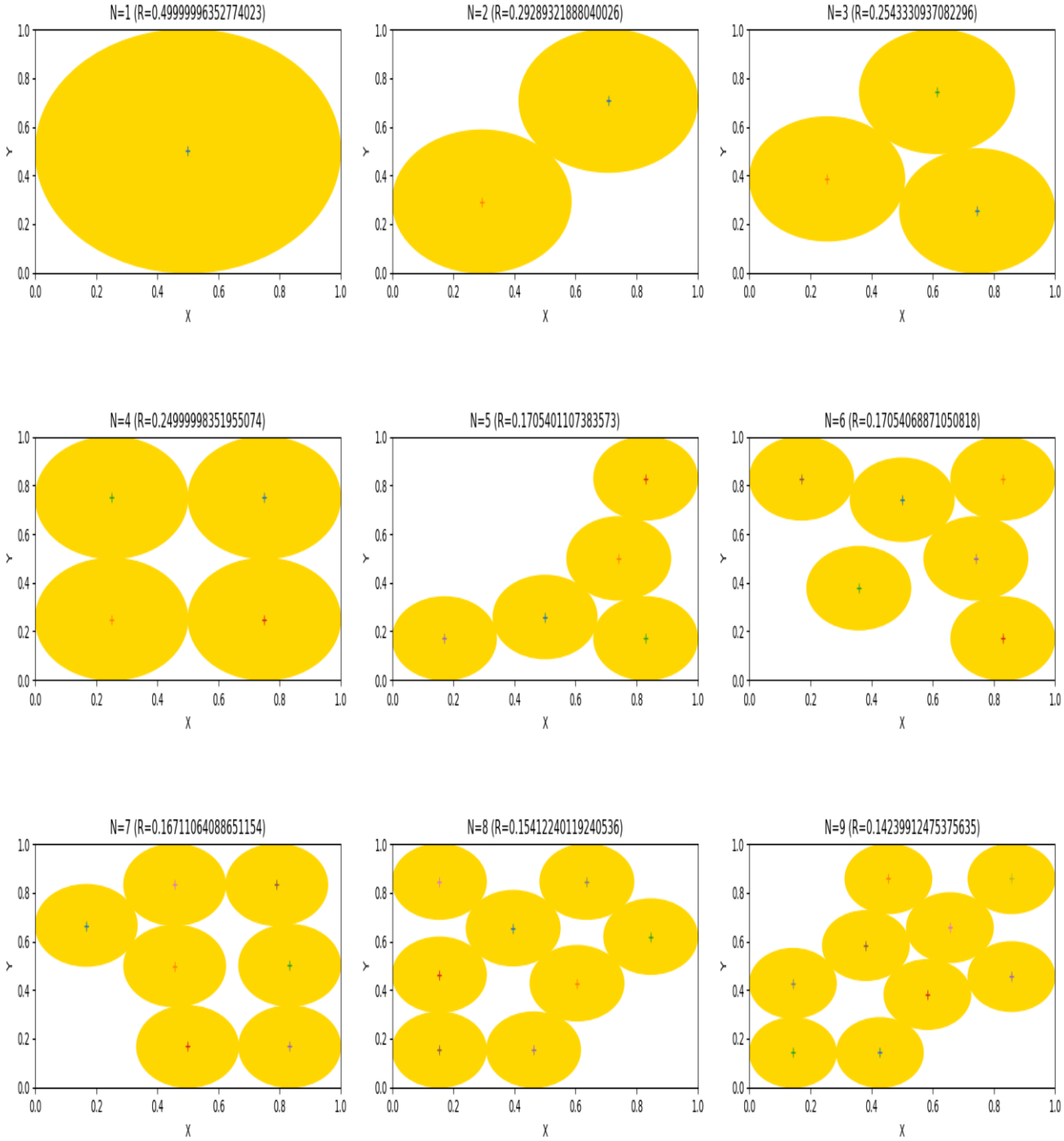


Figure 30: Packing circles between unit circle and circle of radius R_min for n={5, 10, 20, 50}

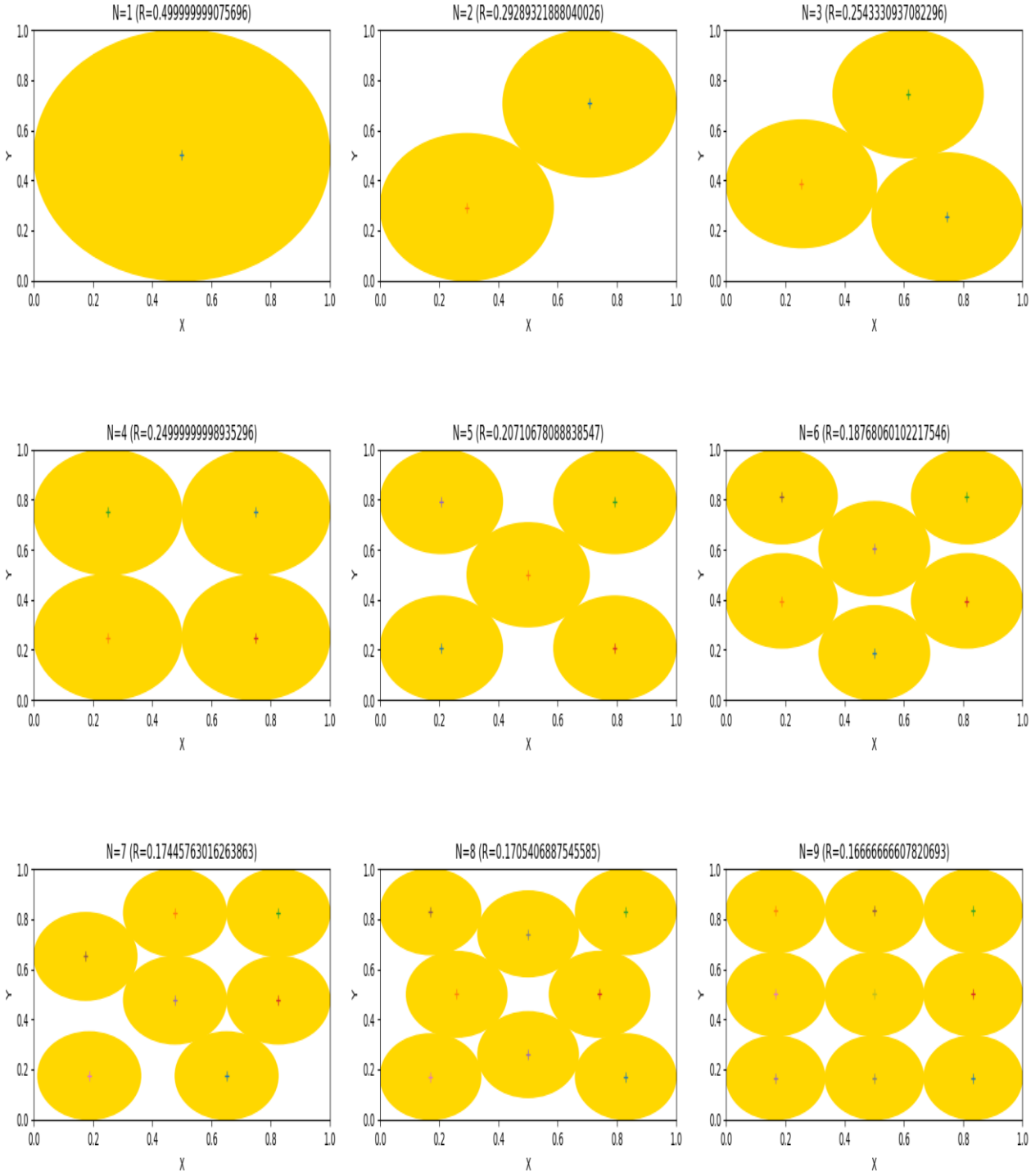
Appendix 1: Packomania results



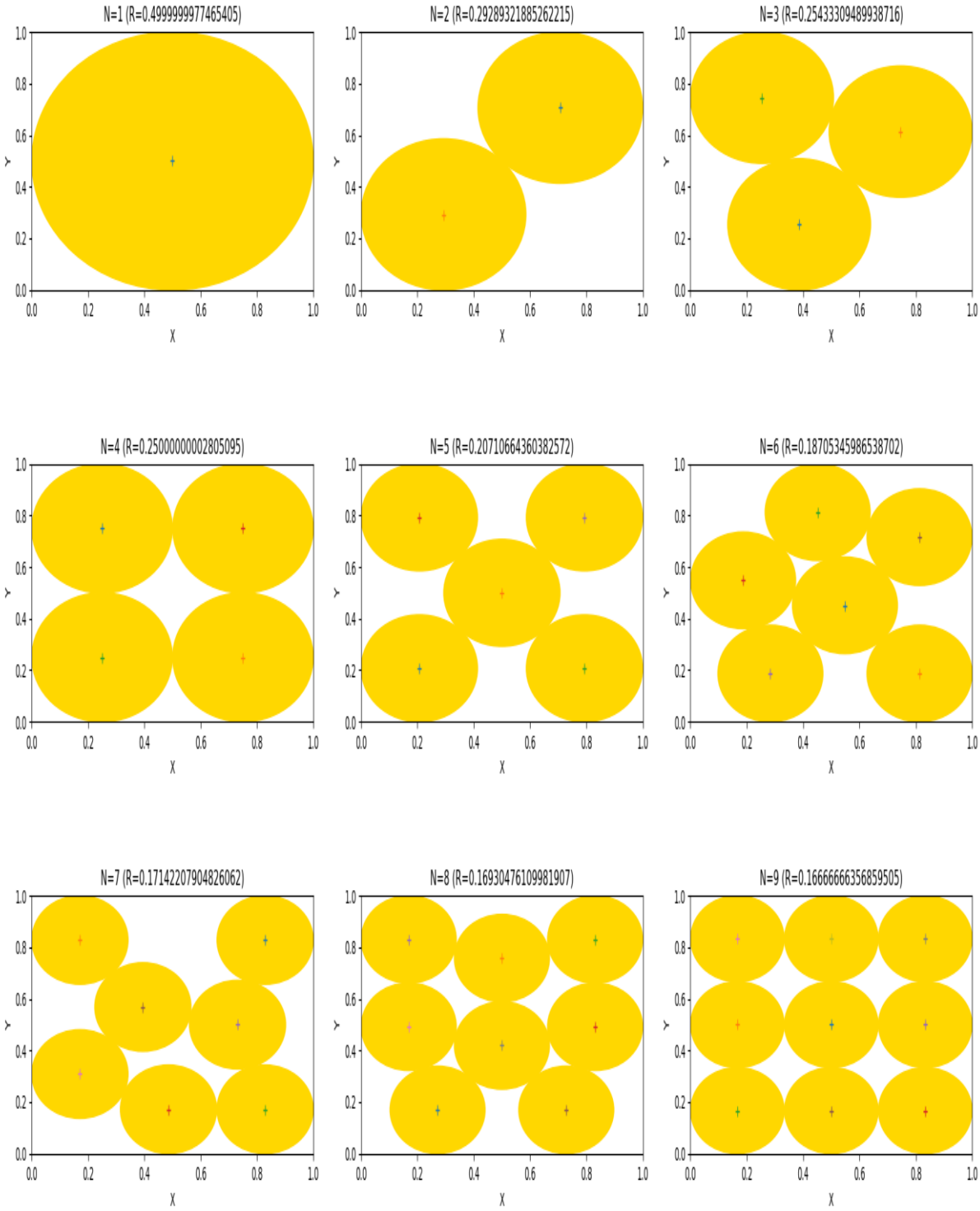
Appendix 2: Multistart



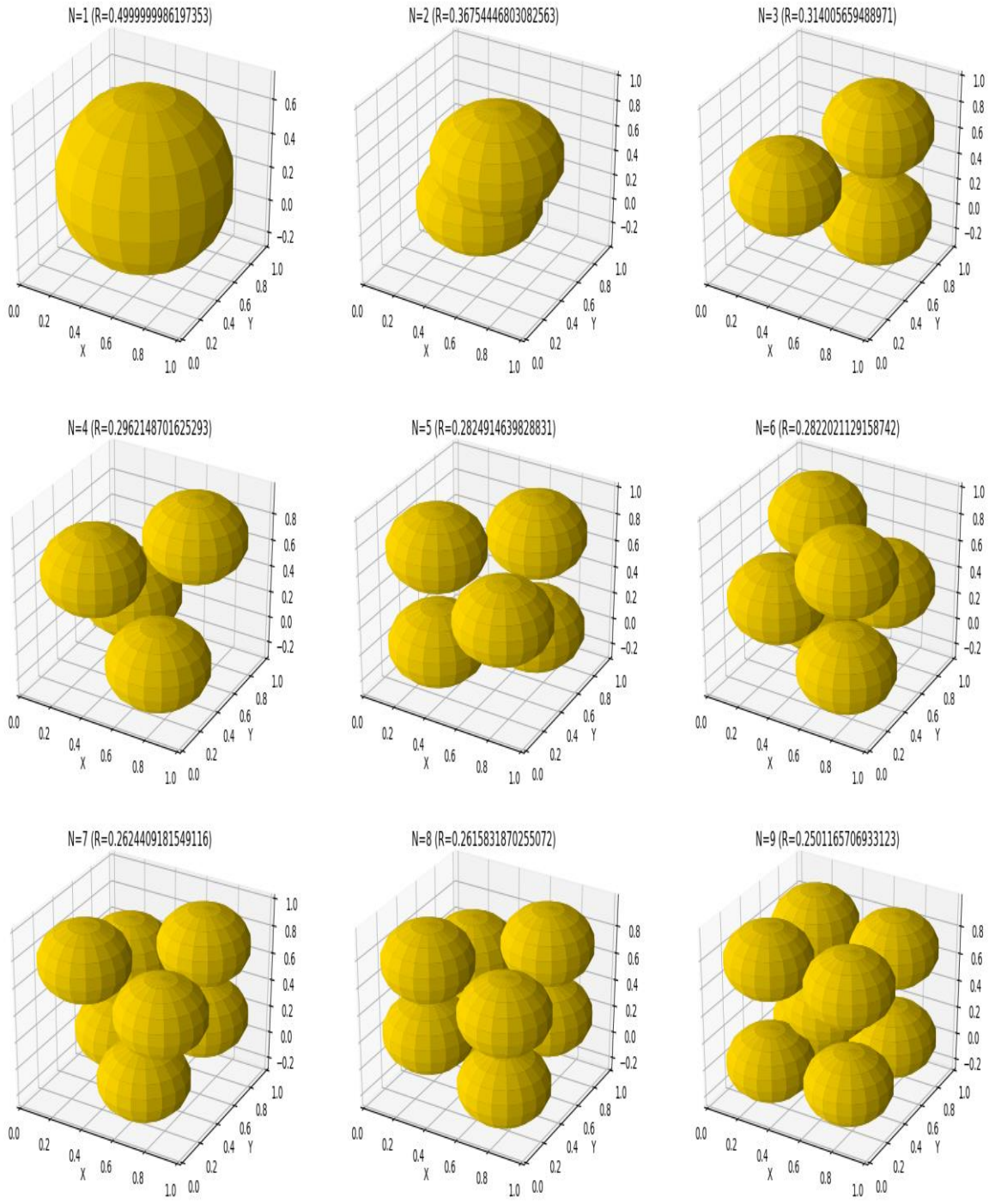
Appendix 3: MBH



Appendix 4: MBH Multitrial



Appendix 5: MBH for Sphere-packing



Source:

http://www.packomania.com/ (1)

A heuristic for the circle-packing problem with a variety of containers (2):
https://www.researchgate.net/publication/257195506_A_heuristic_for_the_circle_packing_problem_with_a_variety_of_containers

Packing circles in a square: A review and New Results (3):
https://www.researchgate.net/publication/228327423_Packing_Circles_in_a_Square_A_Review_and_New_Results