Techniques in Operations Research
Group Assignment 2018

Group 'TBC'; Group Number: 3
Brendan Law, 350173; Christian Drozdowicz, 832391
Cameron Young, 832254; and Emma Morris, 923654

May 27, 2018

1 Introduction

The d-dimensional Sphere Packing problem is one in which a number of hyper-spheres are to be arranged within a given contained space such that no hyper-spheres overlap, and where the hyper-spheres are arranged to maximise the packing density - that is, the fraction of the available space filled by the hyper-spheres. The Sphere Packing problem that we will be focusing on is the three-dimensional case. Other cases include packing circles on a plane in two dimensions, and packing line segments into a linear universe in one dimension.

The Sphere Packing problem is a well-known mathematical problem that has stymied mathematicians for more than 300 years. In the 17th century, Johannes Kepler conjectured that the densest way to pack together equal-sized spheres in space was in an arrangement called 'face-centred cubic packing', which is the familiar pyramidal piling that we see in grocery stores [1]. Recent advances in computing power have finally allowed computers to deal with the enormous calculations required to solve this problem in higher dimensions. It was not until 1998 that specific dimensional cases of the problem were finally solved, allowing the conjecture to be proved for all cases [1].

In this report we explore the motivations for solving the Sphere Packing problem and delve into the real-world benefits that these solutions provide. We examine the geometric difficulties that arise when trying to pack spheres into a cube. We determine how the sphere packing problem can be constructed into different optimisation problems, how we can apply these methods using MATLAB, and discuss the results of our approach, along with any caveats. Specifically, we aim to investigate how we can maximise the radii (r) of the spheres for a fixed number (n) of identical spheres in a three-dimensional unit cube.

2 Motivation

2.1 In Three Dimensions

2.1.1 Materials Science [2]

In the fields of engineering and materials science, there is often a duality between the strength of a material and the ease with which it can be manipulated. Stronger materials may be harder to work with and build from, while those that can easily be shaped and worked may not have the requisite strength to withstand the rigours they will be subjected to. Hence, an important challenge is to discover new materials that are both strong and ductile.

An example of this is concrete, or more specifically, High Performance Concrete (HPC) and High Strength Concrete (HSC). *High performance* refers to concrete with a number of qualities, such as long-term durability, ease of placement and working, density and permeability, all of which may be crucial for the environment in which it is to be used. *High strength*, on the other hand, refers simply to the concrete having a high level of compressive strength. In order to combine these two categories, material engineers have utilised a sphere packing model.

Concrete is comprised of two substances, aggregate (rock particles) and cement. The block of concrete is modelled as a space into which spheres are to be packed, while the aggregate particles are modelled as spheres, suspended within the cement. The objective is to minimise or eliminate pockets of air within the concrete structure. To do this, the

volume of aggregate particles must be maximised, and also be larger than the volume of air within the concrete, thus lending itself to the sphere packing problem.

2.1.2 Container Loading

Many industries want to maximise the number of materials delivered in a given shipment. For example, a ball bearing manufacturer may be limited to a single shipping container, and need to send as many identically sized bearings to a foreign port. This is essentially a physical realisation of the sphere packing problem, as the ball bearings are three-dimensional spheres, and the packing arrangement dictates the maximum number of bearings that can fit into a container.

2.1.3 Wireless Network Layouts

In a multi-storey business building, a typical problem is to determine an optimal number of wireless transmitters to ensure sufficient coverage across the whole space. Although wireless network radii can be overlapped, this is generally not cost efficient in large structures. Furthermore, this type of overlap may result in destructive interference, which can lead to dropped packets and reduced signal quality, stability and speed.

We can model the maximum distance at which a transmitter can propagate a signal as the radius of a sphere, and then attempt to optimise the density of coverage within the space without signal overlap.

However, it should be noted that this problem less strict constraints relative to our problem. For example, only the sphere centres (routers) need be within the space, but not the entire sphere (consider placing a transmitter near an external wall, with the signal propagating outside the building). Additionally, since the primary goal is complete coverage, some amount of overlap between signals will need to be permitted.

2.1.4 Molecular Dynamics [3]

In the field of Computational Chemistry, Molecular Dynamics is a useful tool for using computer simulation to study the interactions of atoms and molecules. With the increase in computing power, more complex systems are being examined. However, these simulations require starting points and adequate constraints.

Regular lattices can be used when studying simple liquids and mixtures, but for more complex systems, this approach is inefficient. This problem can be modelled on the sphere packing problem, where the goal is to place known objects in a finite domain so that the distance between any pair of points of different objects is larger than a threshold tolerance. In this case the objects are molecules and points are atoms.

Therefore, finding the starting positions can be modelled as a sphere packing problem in a box, with satisfactory results being obtained quicker than conventional methods.

2.2 Slight Variation in the 3-Dimensional Sphere Packing Problem

2.2.1 Gamma Knife Radiosurgery [4]

Gamma knife radiosurgery is performed by fixing a large helmet onto a patient's head, then a high-intensity cobalt radiation is administered that is concentrated in a small volume in order to destroy tumour tissue.

The main problem is how the surgeons can use as few shots as possible to target at least 90% of the tumour tissue volume, whilst minimising the amount of radiation hitting normal tissue and ensuring that the shots do not overlap, as the levels of radiation across the volume should be as even as possible.

Therefore, the gamma knife treatment problem can be stated as a sphere packing problem - for a given volume (most likely irregularly shaped), arrange spheres of different diameters such that at least 90% of the volume is occupied. Although the standard sphere packing considers uniform spheres within a regular shaped volume, it can still be used as a foundation to the more complicated solution.

2.3 In other dimensions

2.3.1 Facility Location Problem (P-Dispersion Problem) [5], [6]

In logistics, the Facility Location problem, also known as Location Analysis, deals with the optimal placement of facilities to minimise transport costs as well as meeting constraints such as requiring that hazardous materials are not placed near housing areas, or that competing businesses are not located too close together. This is particularly useful in telecommunications, where optimising the dispersion of tranceivers will minimise interference.

This problem can also be modelled as a p-dispersion problem. In this problem, we have p points in the feasible region, the objective is to disperse these selected points such that the minimum distance between pairs is maximised, which will in turn disperse the selected points as much as possible. This is formed as a non-linear programming problem, and if the space in a p-dispersion problem is a square, it is equivalent to packing the square with p equal circles of largest possible radius - exactly the 2-dimensional packing problem.

2.3.2 Error Correcting Code (ECC)[7]

When transmitting information across a network with a high degree of noise or interference, ensuring that the received information is still correct can become difficult. Error Correcting Codes are a method of ensuring that such information is able to be retained.

In modern communications, data is transmitted as binary, strings of ones and zeros. When using an ECC, the goal is to ensure that the encoded binary message is sufficiently unique, so that it is still possible to distinguish it from other messages, even if it is corrupted or interfered with during transmission. While it might seem simple to send each message as a long, unique string of binary, in practice we wish to limit the increase in data that an encoding causes - if we needed a hundred times as much data to encode a message as the original message contained, this would be extremely inefficient.

Hence, our goal is to maximise the 'uniqueness' of each encoding, while still fitting all encodings into a given amount of data. We can quantify this level of 'uniqueness' using what is known as Hamming distance, which is essentially the total bitwise difference between two encodings.

Given some encoding length n, we can replace Euclidean distance with Hamming distance and treat an n-hypercube as a 'container' for our encodings, and thus model this problem as an n-dimensional sphere packing.

3 Optimisation Problem - Modelling

The general form of this problem can be characterised as a problem of fitting n points in a hypercube in d-dimensional Euclidean space, whilst requiring that the minimum distance between:

a) each possible pairing of points is $2r$, and

b) between any point and boundary of the hypercube is r.

and subsequently, maximising the value of r which achieves this. In the general case, r is the distance between any point and its boundary in any direction. In three dimensions, this problem is equivalent to having spheres with a certain radii r, centred around each of the n points, within a unit cube.

In d dimensions, the problem with n spheres and radii r can be modelled as a nonlinear constrained optimisation problem, with the nonlinear program (NLP) written as:

$$\min_x(-r) \tag{1}$$

subject to:

$$\sqrt{\sum_{c=1}^{d}(X_{i,c} - X_{j,c})^2} \geq 2r, \qquad \text{for } 1 \leq i < j \leq n \tag{2}$$

$$X_{i,c} \geq r, \qquad \text{for } 1 \leq i \leq n, 1 \leq c \leq d \tag{3}$$

$$X_{i,c} \leq 1 - r, \qquad \text{for } 1 \leq i \leq n, 1 \leq c \leq d \tag{4}$$

where $X_{i,c}$ represents the coordinate in the c^{th} dimension ($c \in \{1, 2, ..., d\}$) of point i ($i \in \{1, 2, ..., n\}$). Without loss of generality, our unit hypercube consists of the space $[0, 1]^d = [0, 1] \times [0, 1] \times ... \times [0, 1]$. In other words, each dimension of our cube is restricted to the interval $[0, 1]$ and its extreme corners lie at $(0, 0, ..., 0)$ and $(1, 1, ..., 1)$.

- Equation (1) is derived by converting the problem of maximising r to the equivalent problem of minimising $-r$, allowing us to solve this problem using optimisation techniques learned from this subject.
- Constraint (2) is derived from the distance formula between any two points in d-dimensional Euclidean space. Because all pairs of points lie in a sphere with radius r, we require that these points are at least $2r$ apart to prevent any spheres from overlapping. Note that the sum is over the d dimensions and between each pair of points. We compare points $1 \leq i < j \leq n$ as the Euclidean distance is symmetrical, i.e. $\mathrm{d}(X_1, X_2) = \mathrm{d}(X_2, X_1)$, where $\mathrm{d}(\cdots)$ represents the Euclidean distance between two points.
- Constraint (3) requires that the centre of any sphere is at least r from the lower bound (0) of the cube in all dimensions. As the radii of each sphere is r, this ensures that no point is placed such that its boundary has a value less than 0 in any dimension.
- Constraint (4) is similar to (3), but for the upper bound (1) of the cube in all dimensions. As the radii of each sphere is r, this ensures that no point is placed such that its boundary has a value greater than 1 in any dimension.

This results in an NLP with:

- $nd + 1$ variables (each point has d dimensions, as well as a radius, r),
- $\binom{n}{2} = \frac{n(n-1)}{2}$ inequalities from (2), and $2nd$ inequalities from (3) and (4), resulting in a system of $\frac{n(n-1)}{2} + 2nd$ inequality constraints.

For the specific case in three dimensions, with n spheres each with radii r, the NLP (re-written with constraints ≤ 0) is:

$$\min_x(-r) \tag{1}$$

subject to:

$$2r - \sqrt{(X_{i,1} - X_{j,1})^2 + (X_{i,2} - X_{j,2})^2 + (X_{i,3} - X_{j,3})^2} \leq 0, \qquad \text{for } 1 \leq i < j \leq n \tag{2}$$

$$r - X_{i,1} \leq 0, \tag{3.1}$$

$$r - X_{i,2} \leq 0, \qquad \text{for } 1 \leq i \leq n \tag{3.2}$$

$$r - X_{i,3} \leq 0, \tag{3.3}$$

$$X_{i,1} - (1 - r) \leq 0, \tag{4.1}$$

$$X_{i,2} - (1 - r) \leq 0, \qquad \text{for } 1 \leq i \leq n \tag{4.2}$$

$$X_{i,3} - (1 - r) \leq 0, \tag{4.3}$$

3.1 Penalty methods

Like other constrained optimisation problems, we can "convert" this into an unconstrained problem using penalty methods. This allows this problem to be approached using numeric and algorithmic methods rather than just analytic methods. As with all penalty methods, constraints are incorporated into the objective function, with very large penalties be imposed when the constraints are not met. This coerces the algorithm into finding a solution which, whilst minimising the objective function, also gives a very good chance of meeting all the constraints required.

We will not explicitly evaluate the NLP using penalty methods in this report, but they will (implicitly) be used in section 5, when we explore other methods, such as Genetic Algorithms, which solve the NLP by utilising this formulation of the problem.

3.1.1 Log barrier penalty method

For a simple scenario of 3-dimensions and two spheres, each with radii r, we can write the new, unconstrained penalty function, with penalty term α:

$$P_\alpha(x) = f(x) - \frac{1}{\alpha}\big[\log(\sqrt{(X_{1,1}-X_{2,1})^2+(X_{1,2}-X_{2,2})^2+(X_{1,3}-X_{2,3})^2}-2r) + \sum_{i=1}^{2}\sum_{j=1}^{3}\log(X_{i,j}-r)+\sum_{i=1}^{2}\sum_{j=1}^{3}\log((1-r)-X_{i,j})\big]$$

Thus, with n spheres, noting that the sum of two logs is the log of the product, we get:

$$P_\alpha(x) = -r - \frac{1}{\alpha}\big[\log(\prod_{1\le i<j}^{n}\sqrt{(X_{i,1}-X_{j,1})^2+(X_{i,2}-X_{j,2})^2+(X_{i,3}-X_{j,3})^2}-2r) + \log(\prod_{i=1}^{n}(X_{i,1}-r))+\log(\prod_{i=1}^{n}(X_{i,2}-r))+\log(\prod_{i=1}^{n}(X_{i,3}-r)) + \log(\prod_{i=1}^{n}((1-r)-X_{i,1}))+\log(\prod_{i=1}^{n}((1-r)-X_{i,2}))+\log(\prod_{i=1}^{n}((1-r)-X_{i,3}))\big]$$

This function will be analysed as $\alpha \to \infty$ to evaluate a minimiser and the respective Karush-Kuhn-Tucker (KKT) multipliers as the constraints, $g_i(x)$, go to 0. The log barrier method has the advantage of having a twice continuously differentiable penalty function, allowing Newton's method to be applied for increased efficiency. Practically, however, this requires α to be extremely large to evaluate properly, which may cause computation and rounding errors. Therefore, an "exact method" such as the ℓ_1 penalty function, may be more practically appropriate, which still requires α to be sufficiently large, without being required to replicate an "$\alpha = \infty$" scenario. For n spheres:

$$P_\alpha(x) = -r + \alpha\big[\sum_{1\le i<j}^{n}\max\{2r-\sqrt{(X_{1,1}-X_{2,1})^2+(X_{1,2}-X_{2,2})^2+(X_{1,3}-X_{2,3})^2},0\} + \sum_{i=1}^{n}\sum_{j=1}^{3}\max\{r-X_{i,j},0\} + \sum_{i=1}^{n}\sum_{j=1}^{3}\max\{X_{i,j}-(1-r),0\}\big]$$

4 Optimisation Problem - Solving

Our primary method for solving the NLP was based around MATLAB's built-in *fmincon* function, around which we developed several algorithms. Each of these algorithms was tested for 1 to 30 spheres.

Results of the maximal radius found for each value of n spheres from $n = 1, ..., 30$, and the run-time of each algorithm can be found in the 'Appendix - Results' in section 8. Note that the 'best known' results we have benchmarked our performance against are from the 'Packomania' website [9].

4.1 Grid and Randomised initialisataion

Initially, we implemented two naive methods. Both methods consisted of a single use of *fmincon*, with a grid[1] and a randomised[2] initial condition respectively. Neither of these methods was particularly effective, and both performed extremely suboptimally for larger numbers of spheres.

[1] 'Grid' refers to a truncated simple cubic packing

[2] 'Randomised' refers to an initial condition with the x, y, z coordinates of each sphere taken from the uniform distribution $U(0, 1)$

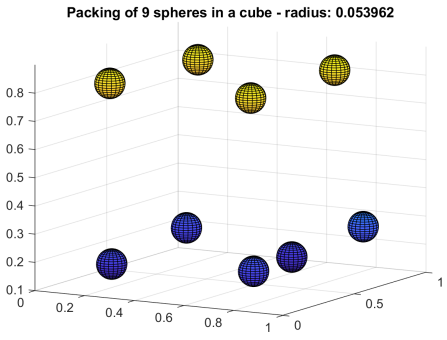

Figure 1: A clearly suboptimal packing of 9 spheres

4.2 SQP

The default algorithm for *fmincon* is an 'interior point' algorithm. This is a general algorithm suitable for a wide variety of optimisation problems. However, it carries the disadvantage that for a particular problem, the algorithm may not solve the problem as optimally as another, more specific algorithm would. Instead, we used a Sequential Quadratic Programming (SQP) algorithm, which is one of the algorithms supported by MATLAB within *fmincon*. SQP methods are appropriate for problems in which both the objective and constraint functions are twice continuously differentiable, a condition that our NLP satisfies. This algorithm was run from both grid and randomised initial conditions, as in the previous stage. Additionally, we also altered several of *fmincon*'s default options, which prevented the function from terminating prematurely.

In the case of the grid starting point, this algorithm does not alter the initial grid arrangement (although this is still an improvement from the interior point algorithm). Using a random starting point, however, yielded significantly more useful results. The average accuracy (over 30 spheres) using this method was 97.36%, and for some numbers of spheres the best known value was attained.

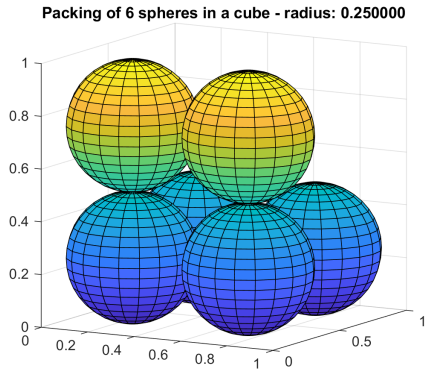

Figure 2: Packing obtained from a grid initial condition

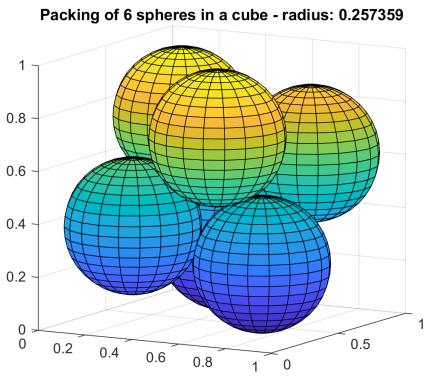

Figure 3: Packing obtained from a randomised initial condition

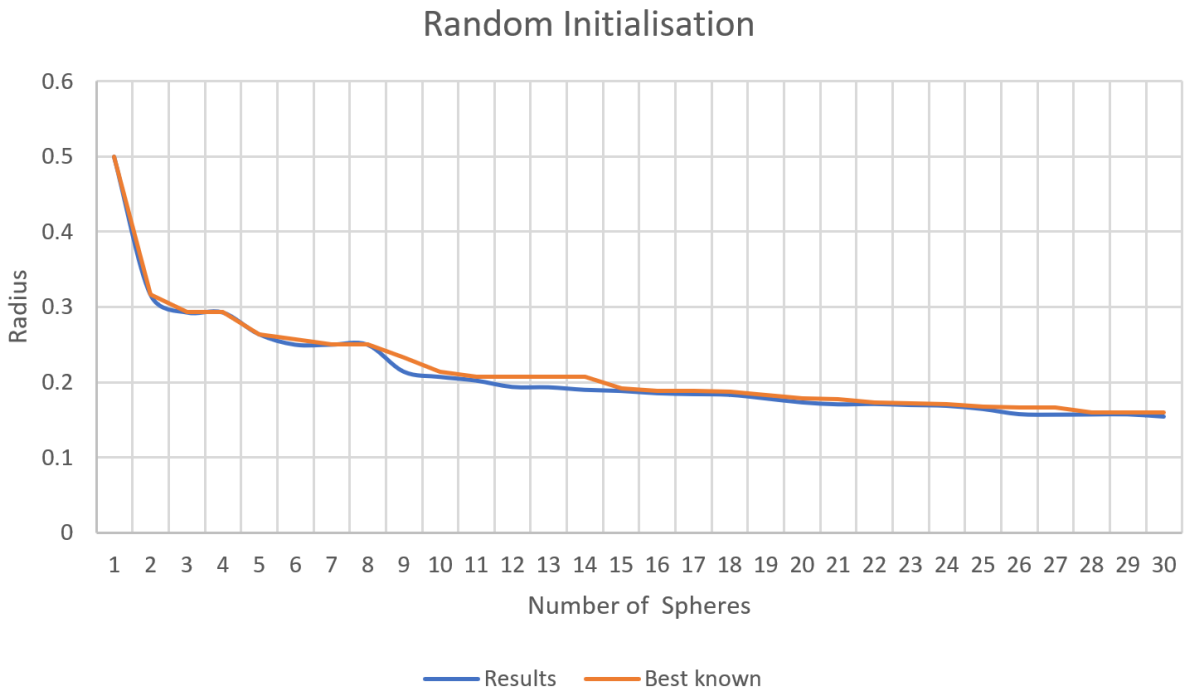

Figure 4: Comparison with best known radii

4.3 20 simulations

While the single run of *fmincon* on a randomised starting point gave promising results, the inherently probabilistic nature of the process means that it may not always give an optimal result, or indeed even the same result. Hence, to mitigate this randomness, we simply repeated the same process 20 times with a new random initial condition each time, taking the best result from these 20 simulations. This approach further improved our results to an average accuracy of 99.67%.

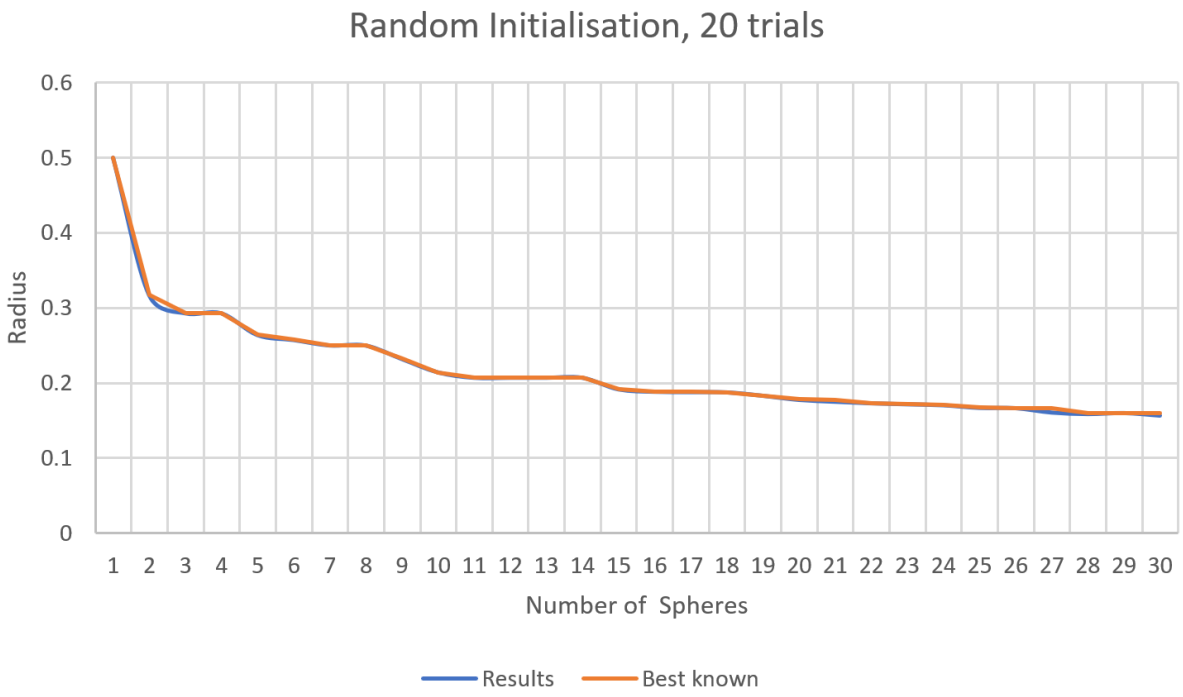

Figure 5: Comparison with best known radii

4.4 Parallelisation

Although increasing the number of simulations improved accuracy, it also increased the runtime of the process (approximately by a factor of 20, as one would expect). Clearly, this means that it will be difficult to scale up this approach as a solution for higher numbers of spheres. To combat this problem, we parallelised the algorithm, allowing multiple simulations to be run simultaneously (hardware permitting). This alteration improved our runtime, which decreased from approximately 440 seconds in the serial case to approximately 85 when run in parallel. The average accuracy remained similar, at 99.74%. We attributed this small increase to the inherently random nature of the algorithm, as it was still close to the value obtained in the serial case.

Furthermore, parallelisation allows our algorithm to scale with the number of CPU cores allocated to it, rather than scaling with CPU frequency alone. This is an important improvement, as even with modern technology, there is an effective threshold on CPU frequency (of around 5GHz), whereas it is far simpler to add additional cores/CPUs. Hence, an algorithm that is capable of scaling with additional cores is much more powerful. Similarly, parallelisation may also allow GPU computation (using CUDA) to be utilised, potentially offering even greater improvements in runtime. Unfortunately, we were unable to explore this avenue, due to the lack of a MATLAB- and CUDA-compatible GPU.

4.5 Perturbation

Although our algorithms thus far had attained the best known values for most values of n, there were several values of n (for example 21,22,27,28, amongst others) for which optimality was unattainable. In an attempt to increase the radius in these cases, we implemented perturbation.

When using an optimisation algorithm, it is possible for the algorithm to become 'stuck' at a local optimum, where all possible directions worsen the result, and thus be unable to reach a global optimum. Our goal is to allow the algorithm to break out of these local solutions, and thus be free to move onto a more optimal solution. By perturbing a locally optimal packing, we jolt the spheres out of this packing, while still remaining 'close' to the solution. Note that if we perturb the arrangement by too large a factor, then this is effectively no different from starting over with a new randomised initial condition.

4.5.1 Basic Perturbation

Our first method of perturbation was fairly straightforward. After running 20 random trials, as in the previous algorithm, the best packing from these trials was selected as the packing to be perturbed. Then, each coordinate of each sphere would be altered by a random amount, and the radius reset to zero. This random amount was generated by taking a value from the uniform distribution $U(-0.5, 0.5)$ (thus allowing for movement in both positive and negative directions), and then multiplying it by a given percentage of the original radius.

From here, *fmincon* was then reapplied (using the same settings as before) and allowed to find an optimum from this new starting point. This process was then repeated 20 times, and the best of these packings retained.

4.5.2 Stepped Perturbation

However, it is difficult to determine what percentage of the radius yields the best results, especially since it may differ between packings and numbers of spheres. Consequently, our second algorithm was developed in order to remove or at least reduce the need to determine this value.

Under this method, there are several 'steps' of perturbation, with each step performed in the same fashion as in the first type of perturbation. We specify a number of steps, along with an initial perturbation percentage, and carry out a number of perturbations and reoptimisations at this level of perturbation. The maximum perturbation value is then reduced, and more perturbations carried out. This continues until the specified number of steps has been performed. Carrying out the perturbations in a stepped manner means that we are less likely to 'over-perturb' a packing and skip over a more optimal solution.

4.5.3 Performance

Unfortunately, neither of these two methods of perturbation offered any noticeable improvement. For some numbers of spheres, such as 15, 25 or 27, there was a very small increase in radius, but in the majority of cases there was no change, even if the initial radius was less than the best known value. Furthermore, even in the values of n whose radii were improved, the changes were negligible, being of order 10^{-5} or even lower. The average accuracy remained at 99.74%, as with the 20 (parallel) random trials.

In conclusion, perturbation methods appeared to offer essentially no improvement, while still increasing the runtime of the algorithm when compared to the previous method of 20 random simulations.

4.6 35 Simulations

To compare the efficacy of perturbation against our more basic approach of random trials, we increased the number of trials to the point that the runtime was approximately equal to that of the stepped perturbation method, which turned out to be 35 trials. We then compared the accuracy of this method with that of the perturbations. Increasing the number of trials to 35 raised the average accuracy to 99.85%, a moderate increase when compared to the perturbation methods or only 20 trials.

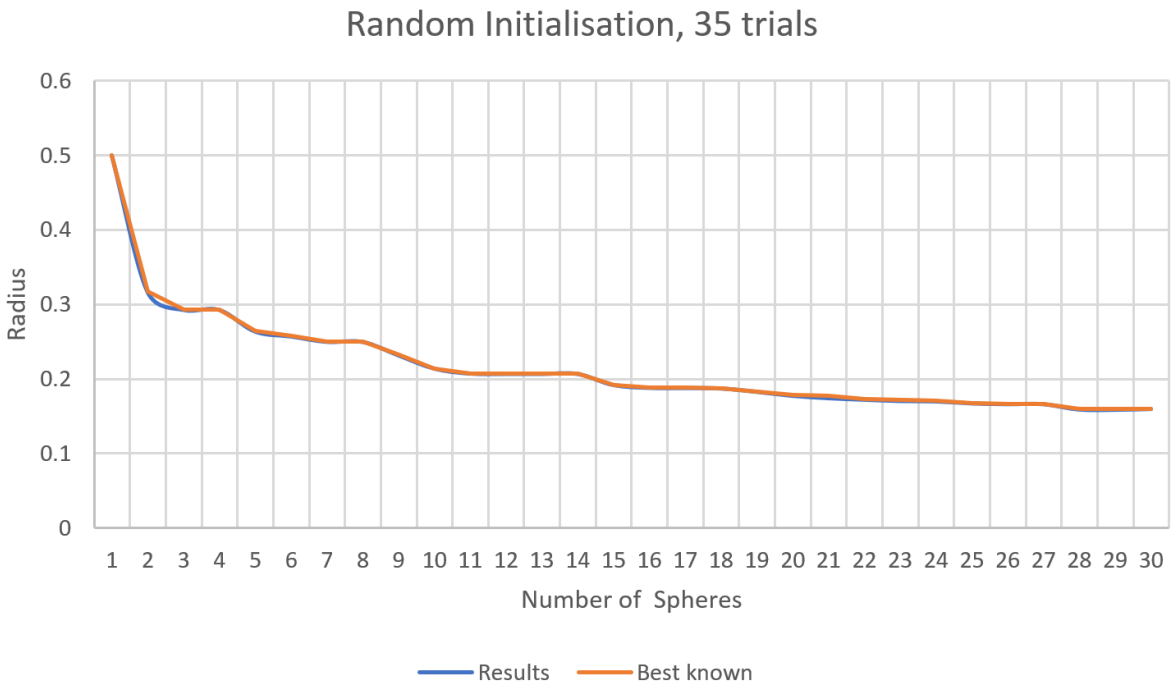

Figure 6: Comparison with best known radii

4.7 Scaled Simulations

When comparing the results from 20 trials, 35 trials and the best known values, we noted that for lower numbers of spheres, 20 or even one simulation was enough to reliably attain the maximum radius, while for larger numbers these extra trials generally improved the resultant radius. Indeed, even with 35 simulations, our algorithm became much less reliable at finding the optimum radius above 15 spheres. Thus, our final improvement was to have the number of simulations scale with the number of spheres, so that lower values of n did not have 'unnecessary' extra simulations carried out, while higher values had a sufficient number of trials to attain the best known radius.

We applied this algorithm for all numbers of spheres from 1 to 60, achieving an average accuracy of 99.71% and a total run time of approximately 4 hours 45 minutes.

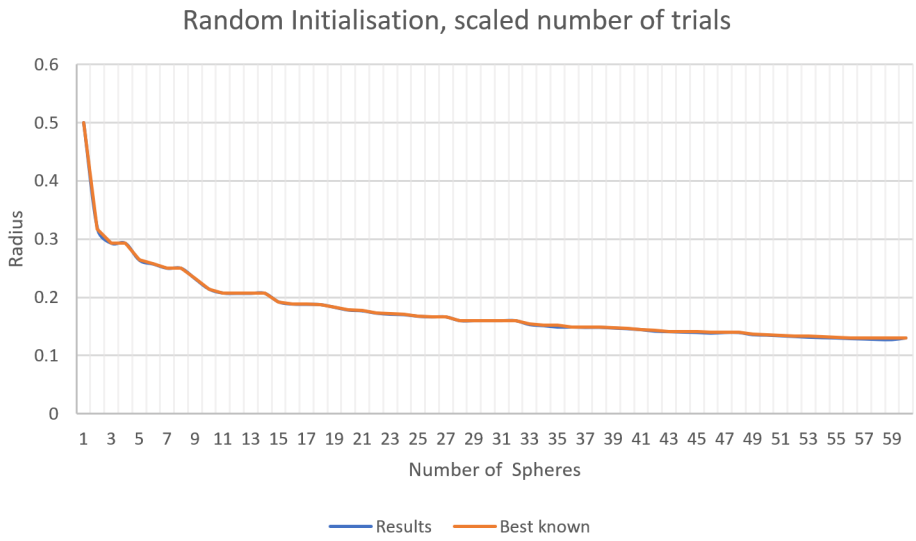

Figure 7: Comparison with best known radii

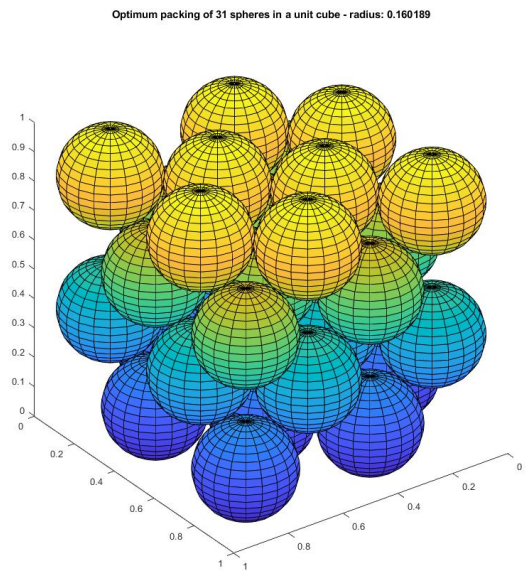

Figure 8: An optimal packing of 31 spheres

5 Other methods - Solving

Computational or metaheuristic methods can be adapted to this constrained optimisation problem using the penalty form of the NLP. As seen in section 3, this 'converts' our NLP into an unconstrained optimisation problem.

We explored the Genetic Algorithm (GA) and Particle Swarm Optimisation (PSO), both of which were featured in the literature as algorithms which have been successfully adapted to the sphere packing problem, and where generic MATLAB code was available.

5.1 Genetic Algorithm

5.1.1 Overview

Genetic Algorithms (GA) are based on the (Darwinian) principles of natural selection. It is a gradient-free method that does not impose any assumptions about the problem. Its basic steps involve:[10]

- Initialising a population with randomised solutions
- Evaluating the population based on some fitness (objective) function. In our case, this would include the radius as well as any penalties on solutions which do not meet constraints
- Using *selection, crossover* and *mutation* to iterate to the next 'generation'
- Iterating until convergence upon an optimal solution, which minimises the negative radius plus any penalties

The *selection, crossover* and *mutation* processes govern how the next generation is created from the current generation, playing a major role in determining the nature of the solutions carried through, as well as the level of randomisation applied to prevent being 'stuck' in a local optima. **Selection** involves taking the top $x\%$ of the population by fitness (the *elite*) and moving them straight onto the next generation. **Crossover** aims to replicate breeding and inheritance by taking two solutions from the current generation and 'combining them' (crossing them over) to create an offspring which will be progressed to the next generation. This is subject to two main decisions:

- How the two crossover candidates are selected - for example, selecting two candidates proportional to their ranking (high ranking for fitter solutions) with the idea that fitter individuals are more likely to create offspring, and

- The method for crossing over two selected candidates. An example is a single-point crossover - the offspring inherits parent A's elements up to a certain crossover point, and then parent B's afterward:

$$\begin{gathered}\text{Parent A: } [X^A_{1,1}, X^A_{1,2}, X^A_{1,3}, \ldots, X^A_{i-1,1}, X^A_{i-1,2}, X^A_{i-1,3}, \ldots, X^A_{n,1}, X^A_{n,2}, X^A_{n,3}, r] \\ + \\ \text{Parent B: } [X^B_{1,1}, X^B_{1,2}, X^B_{1,3}, \ldots, X^B_{i,1}, X^B_{i,2}, X^B_{i,3}, \ldots, X^B_{n,1}, X^B_{n,2}, X^B_{n,3}, r] \\ = \\ \text{Offspring: } [X^A_{1,1}, X^A_{1,2}, X^A_{1,3}, \ldots, X^A_{i-1,1}, X^A_{i-1,2}, X^A_{i-1,3}, X^B_{i,1}, X^B_{i,2}, X^B_{i,3}, \ldots, X^B_{n,1}, X^B_{n,2}, X^B_{n,3}, r]\end{gathered}$$

Mutation 'mutates' a current generation solution by perturbing, swapping, or inverting some of its elements, and passing this mutated solution onto the next generation. This is the mechanism which the GA uses to ensure it continues searching the solution space. An example of a simple swap mutation would be as follows:

$$\begin{gathered}\text{Parent: } [X_{1,1}, X_{1,2}, X_{1,3}, \ldots, X_{i-1,1}, X_{i-1,2}, X_{i-1,3}, \ldots, X_{n,1}, X_{n,2}, X_{n,3}, r] \\ \text{Offspring: } [X_{1,1}, X_{1,2}, X_{n,3}, \ldots, X_{i-1,1}, X_{i-1,2}, X_{i-1,3}, \ldots, X_{n,1}, X_{n,2}, X_{1,3}, r]\end{gathered}$$

Creating new generations continues until some convergence criteria is met, for example, a number of consecutive generations without the best element's fitness improving, the algorithm reaches a maximum generation limit, or improvement in the best element between generations falls below some threshold. The fittest element of the convergent generation would be taken to be the optimal solution from this algorithm.

When applied to the sphere packing problem, each element of the population represents the vector of sphere-centre coordinates and the respective radius for a particular configuration, as per section 4. The fitness function would be a penalty function as mentioned in section 3 - MATLAB's default for dealing with constrained problems is 'auglag' (Augmented Lagrangian Method), which is similar to the log-barrier penalty setup.[11] We ran this algorithm for 1 to 25 spheres.

5.1.2 Results

Our implmentation used the *ga* function from MATLAB's Global Optimisation Toolbox. It performed poorly compared to our methods in section 4 - an average accuracy of 60% and a significantly increased runtime. We identified two changes to the default parameters which made a difference to our results:

1. **Constraint method:** The 'penalty' parameter in the *ga* function uses a more complex, dynamic penalty function which is more akin to an ℓ_1-norm penalty method with a 'hard' barrier [12]. This showed an improvement over the default 'auglag'.

2. **Crossover method:** We used a 'tournament of 4' method to select each crossover candidate. This involves pitting 4 randomly selected elements of a population against in other in a tournament, where the strongest candidate is chosen as a crossover candidate; the other candidate is chosen identically. This made an improvement over the default method of 'stochastic uniform', where the chance of any element being selected for crossover is proportionate to each element's fitness function.

Using these settings improved our accuracy to 67%, still well down on our best results and of those in the literature. We noted that maintaining feasible solutions while 'mutating' or 'crossing over' random elements of our population was difficult, especially in a constrained setting. Randomly switching coordinates of one element, or crossing two elements over can easily create a new solution which is no longer feasible - a coordinate of a given sphere may now overlap with another, or lie outside of the cube completely. Moreover, a system must be in place to ensure the radius part of an element is not crossed over or mutated with any of the sphere centre position elements of the element. This requires coding a GA algorithm from scratch and proved difficult to replicate using generic software packages - something we could not achieve in the in the time given.

Some of the successful implementations of the GA for this problem have used bespoke fitness functions, crossover and mutation processes, and procedures for maintaining feasibile solutions.[13][14] Therefore, part of our poor GA performance was due to the increased expertise and care required to adapt a GA to this problem. It was also noted that the strength of the GA over traditional methods, such as combinatorial approaches, came in much higher dimensions of the packing problem. In our case, the added runtime did not pay off.

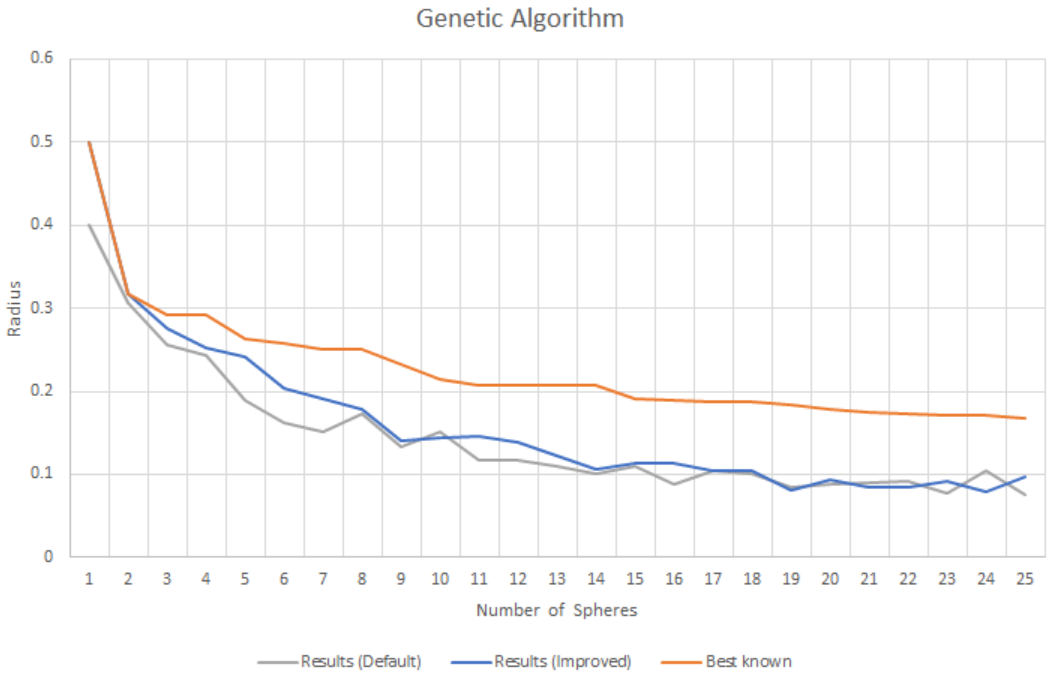

Figure 9: Comparison of Genetic Algorithm with best known radii

5.2 Particle Swarm Algorithm

5.2.1 Overview

Like the GA, the Particle Swarm Algorithm (PSO) is a computational, gradient-free method which relies upon evaluating the fitness of a population (swarm) of initial samples (particles), and updating the 'positions' of each based on a 'velocity'. In summary, at time t, each particle x_i has a:

- Fitness
- Position, $(x_i^{(t)})$
- Velocity or direction of movement, $(v_i^{(t)})$
- Memory of its own previous best position, as well as the swarm's previous best position

At each iteration, a particle's position is updated by:

$$x_i^{(t)} = x_i^{(t-1)} + v_i^{(t)}$$

In standard PSO implementations, velocity is a function of a particle's current position and velocity, and relative distance to its own and the swarm's (historical) best position, controlled by several 'tendency to best position' parameters. This is reiterated until the swarm converges under similar criteria to a GA, with the final result being the fittest particle of the convergent swarm. It was designed to mimic social, 'flocking' behaviour, with all particles 'swarming' toward the optimal solution. The setup of the PSO in the sphere packing problem is almost identical to the GA, bar the processes applied to each particle to move to the next iteration. Our results for the PSO are based on a user-created MATLAB code, which adapts the PSO method to a constrained optimisation problem.[15] We ran this algorithm for 1 to 25 spheres.

5.2.2 Results

Like the GA, our results from this method were significantly worse than those in section 4, with only 50% accuracy. For 2 and 3 spheres, we can see from the results in figure 10 that the PSO achieved a result better than the best. However, on these occasions, and on several others, we noted that our algorithm converged to a solution where some of the sphere coordinates were ¡ 0 or ¿ 1 - clear breaches of the constraints - despite specifying upper and lower bounds of

1 and 0 respectively, for each element of each particle. This means there were either issues with the code, or, like the GA, that it was having issues maintaining feasible solutions between iterations. Furthermore, as the radius coordinate is part of each particle, moving in a particular direction may not correspond with the properties of the updated sphere positions.

Successful adaptations of the PSO to similar problems have used velocities which are a function of the best few particles from a swarm, initialising the swarm through a tree search, as well as a process to 'repair' unfeasible solutions. [16, 17] Hence, we attribute our relatively poor results to an inability to customise the PSO sufficiently in the time provided to the level required to solve this problem with competitive results.

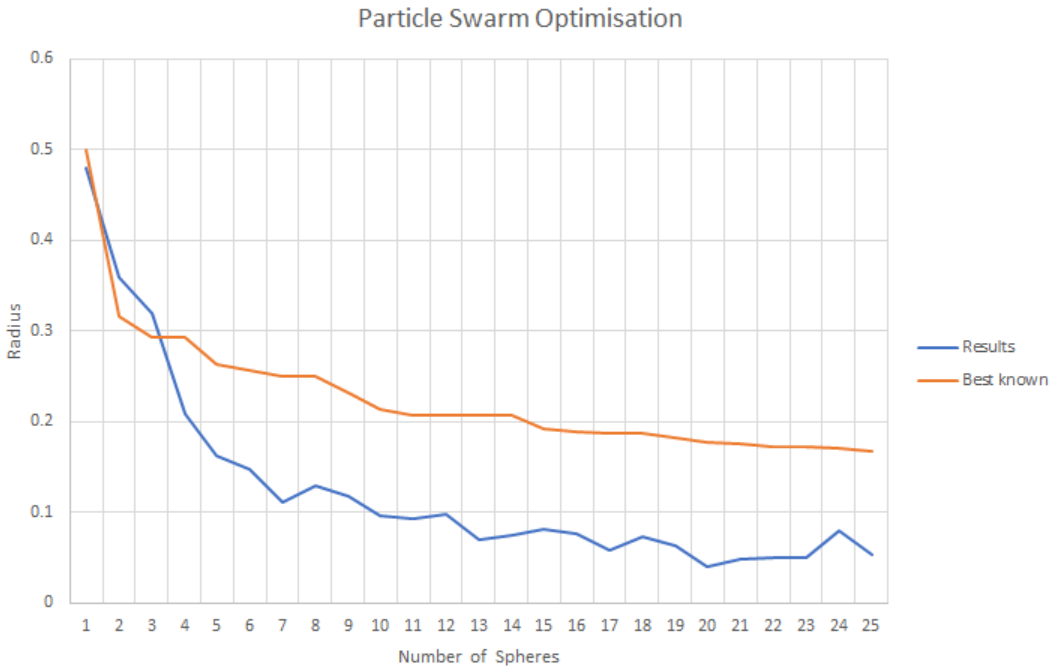

Figure 10: Comparison of Particle Swarm Algorithm with best known radii

6 Conclusion

The sphere packing problem has a long history and very important applications. In particular, the 2- and 3-dimensional cases have been applied in a range of areas, from logistics to surgery. Recent improvements in computational power have allowed this problem to be solved in higher dimensions, with the promise of further applications. It can be expressed as a constrained non-linear program, where centres of spheres are placed in a cube, subject to constraints which control overlaps and cube boundary crossings. As the problem is twice continuously differentiable, there exist established methods for solving this problem efficiently, allowing us to get close to achieving the best recorded results for up to 50 spheres. Other computational methods can be adapted to the problem once it is converted into an unconstrained problem with penalties. However, these methods usually require significant expertise and runtime to be able to provide results which are as competitive as traditional methods for the three-dimensional case involving homogeneous spheres packed within a cube.

7 References

[1] Peterson, I. (1998). Cracking Kepler's Sphere-Packing Problem. *Science News, 154(7)*, 103-103. Retrieved from http://www.jstor.org.ezp.lib.unimelb.edu.au/stable/4010790

[2] Wong, H., & Kwan, A., (2005). Packing density: a key concept for mix design of high performance concrete, *Proceedings of the Materials Science and Technology in Engineering Conference, HKIE Materials Division*, Hong Kong: 1–15.

[3] Martínez, J. M., & Martínez, L. (2003). Packing optimization for automated generation of complex system's initial configurations for molecular dynamics and docking. *Journal of Computational Chemistry, 24(7)*:819-825.

[4] Overbey, J., Kolve, A., & Hirtz, N., Optimizing Gamma Knife Radiosurgery Through Mathematical Morphology. Retrieved from https://pdfs.semanticscholar.org/107f/90aa49a598944f72457834b43fd0b7bf6bee.pdf

[5] Drezner, Z., & Erkut, E., (1995). Solving the continuous p-dispersion problem using non-linear programming. *Journal of the Operational Research Society, 46*,516–520.

[6] Pisinger, D., (1999). Exact solution of p-dispersion problems, DIKU, University of Copenhagen. Retrieved from https://pdfs.semanticscholar.org/1eb3/810077c0af9d46ed5ff2b0819d954c97dcae.pdf

[7] Error correcting codes. (n.d.).Retrieved April 29, 2018, from https://brilliant.org/wiki/error-correcting-codes/

[8] Betts, J., & Gablonsky, J., (2002). A Comparison of Interior Point and SQP, *Methods on Optimal Control Problems, Technical Document Series MCT-TECH-02-004, Mathematics and Engineering Analysis, The Boeing Company*, Retrieved from http://www.seabasedradar.com/assets/pdf/phantom/socs/benchmark/benchmark.pdf

[9] The best known packings of equal spheres in a cube (complete up to N = 1000). Retrieved May 20, 2018, from http://www.packomania.com

[10] How the Genetic Algorithm Works. Retrieved May 23, 2018, from https://au.mathworks.com/help/gads/how-the-genetic-algorithm-works.html

[11] Nonlinear Constraint Solver Algorithms. Retrieved May 22, 2018, from https://au.mathworks.com/help/gads/description-of-the-nonlinear-constraint-solver.html

[12] Kalyanmoy, D., (2000). An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanics and Engineering, 186(2-4)*, 311–338.

[13] Hopper, E., & Turton, B. (1998) Application of Genetic Algorithms to Packing Problems — A Review. In: Chawdhry, P.K., Roy, R., Pant, R.K. (eds) *Soft Computing in Engineering Design and Manufacturing.* Springer, London

[14] Cornforth, D. (2002) Evolution in the Orange Box — A New Approach to the Sphere-Packing Problem in CMAC-Based Neural Networks. In: McKay, B., Slaney, J. (eds) *AI 2002: Advances in Artificial Intelligence. AI 2002. Lecture Notes in Computer Science*, vol 2557. Springer, Berlin, Heidelberg

[15] Constrained Particle Swarm Optimization. Retrieved May 23, 2018, from https://au.mathworks.com/matlabcentral/fileexchange/25986-constrained-particle-swarm-optimization

[16] Hifi, M., Lazure, L., & Yousef, L., (2017). Solving packing identical spheres into a smallest sphere with a particle swarm optimization, *2017 4th International Conference on Control, Decision and Information Technologies (CoDIT), Control, Decision and Information Technologies (CoDIT)*

[17] Shin, Y.-B., & Kita, E., (2012). Application of particle swarm optimization to the item packing problem, *WIT Transactions on The Built Environment, 125*, 245-255,

8 Appendix - Results

All results generated on an AMD Ryzen 5 1600 processor, running MATLAB r2017b on Windows 10 Education Version 1803.

8.1 Results from Preliminary Algorithms

8.1.1 Absolute Results

Table 1: Radius

n	grid	rand	grid sqp	rand sqp	rand 20	Parallel	Perturb 1	Perturb 2	sqp 35	best
1	0.49999	0.49972	0.50000	0.50000	0.50000	0.50000	0.50000	0.50000	0.50000	0.50000
2	0.25000	0.31698	0.25000	0.31699	0.31699	0.31699	0.31699	0.31699	0.31699	0.31699
3	0.29289	0.23751	0.25000	0.29289	0.29289	0.29289	0.29289	0.29289	0.29289	0.29289
4	0.25000	0.01653	0.25000	0.29289	0.29289	0.29289	0.29289	0.29289	0.29289	0.29289
5	0.01450	0.01021	0.25000	0.26393	0.26393	0.26393	0.26393	0.26393	0.26393	0.26393
6	0.20655	0.00943	0.25000	0.25000	0.25736	0.25736	0.25736	0.25736	0.25736	0.25736
7	0.00780	0.00820	0.25000	0.25000	0.25011	0.25011	0.25011	0.25011	0.25011	0.25014
8	0.25000	0.00587	0.25000	0.25000	0.25000	0.25000	0.25000	0.25000	0.25000	0.25000
9	0.05699	0.00494	0.16667	0.21413	0.23205	0.23205	0.23205	0.23205	0.23205	0.23205
10	0.00452	0.00578	0.16667	0.20717	0.21429	0.21429	0.21429	0.21429	0.21429	0.21429
11	0.00382	0.00400	0.16667	0.20229	0.20711	0.20762	0.20762	0.20762	0.20762	0.20762
12	0.00452	0.00413	0.16667	0.19382	0.20711	0.20711	0.20711	0.20711	0.20711	0.20711
13	0.00354	0.00372	0.16667	0.19337	0.20711	0.20711	0.20711	0.20711	0.20711	0.20711
14	0.00321	0.00301	0.16667	0.19001	0.20711	0.20711	0.20711	0.20711	0.20711	0.20711
15	0.00259	0.00260	0.16667	0.18851	0.19170	0.19170	0.19170	0.19170	0.19231	0.19231
16	0.00190	0.00242	0.16667	0.18547	0.18848	0.18828	0.18828	0.18828	0.18850	0.18880
17	0.00166	0.00221	0.16667	0.18427	0.18798	0.18829	0.18829	0.18829	0.18824	0.18869
18	0.16136	0.00229	0.16667	0.18343	0.18768	0.18768	0.18768	0.18768	0.18768	0.18768
19	0.00181	0.00170	0.16667	0.17855	0.18318	0.18319	0.18319	0.18319	0.18319	0.18319
20	0.00171	0.00164	0.16667	0.17331	0.17768	0.17841	0.17841	0.17841	0.17784	0.17841
21	0.00122	0.00147	0.16667	0.17069	0.17522	0.17437	0.17437	0.17437	0.17464	0.17722
22	0.00116	0.00138	0.16667	0.17120	0.17314	0.17219	0.17219	0.17219	0.17263	0.17327
23	0.00111	0.00136	0.16667	0.16976	0.17182	0.17103	0.17103	0.17103	0.17093	0.17182
24	0.00160	0.00120	0.16667	0.16873	0.17054	0.17027	0.17027	0.17027	0.17037	0.17054
25	0.00081	0.00118	0.16667	0.16448	0.16701	0.16721	0.16723	0.16721	0.16773	0.16780
26	0.00068	0.00108	0.16667	0.15765	0.16667	0.16667	0.16667	0.16667	0.16667	0.16691
27	0.14294	0.00097	0.16667	0.15699	0.16084	0.16131	0.16131	0.16131	0.16667	0.16667
28	0.00073	0.00091	0.12500	0.15726	0.15883	0.15932	0.15932	0.15932	0.15945	0.16019
29	0.00074	0.00087	0.12500	0.15742	0.16019	0.16019	0.16019	0.16019	0.15918	0.16019
30	0.00065	0.00078	0.12500	0.15439	0.15678	0.16019	0.16019	0.16019	0.16019	0.16019

8.1.2 Relative Results

Table 2: Percentage of best known values

n	grid	rand	grid sqp	rand sqp	rand 20	Parallel	Perturb 1	Perturb 2	sqp 35
1	100.00%	99.94%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
2	78.87%	100.00%	78.87%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
3	100.00%	81.09%	85.36%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
4	85.35%	5.64%	85.36%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
5	5.49%	3.87%	94.72%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
6	80.26%	3.67%	97.14%	97.14%	100.00%	100.00%	100.00%	100.00%	100.00%
7	3.12%	3.28%	99.94%	99.94%	99.99%	99.99%	99.99%	99.99%	99.99%
8	100.00%	2.35%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
9	24.56%	2.13%	71.82%	92.28%	100.00%	100.00%	100.00%	100.00%	100.00%
10	2.11%	2.70%	77.78%	96.68%	100.00%	100.00%	100.00%	100.00%	100.00%
11	1.84%	1.93%	80.27%	97.43%	99.75%	100.00%	100.00%	100.00%	100.00%
12	2.18%	1.99%	80.47%	93.58%	100.00%	100.00%	100.00%	100.00%	100.00%
13	1.71%	1.80%	80.47%	93.37%	100.00%	100.00%	100.00%	100.00%	100.00%
14	1.55%	1.46%	80.47%	91.74%	100.00%	100.00%	100.00%	100.00%	100.00%
15	1.35%	1.35%	86.67%	98.02%	99.68%	99.68%	99.68%	99.68%	100.00%
16	1.01%	1.28%	88.28%	98.24%	99.83%	99.72%	99.72%	99.72%	99.84%
17	0.88%	1.17%	88.33%	97.66%	99.62%	99.79%	99.79%	99.79%	99.76%
18	85.97%	1.22%	88.80%	97.73%	100.00%	100.00%	100.00%	100.00%	100.00%
19	0.99%	0.93%	90.98%	97.47%	100.00%	100.00%	100.00%	100.00%	100.00%
20	0.96%	0.92%	93.42%	97.14%	99.59%	100.00%	100.00%	100.00%	99.68%
21	0.69%	0.83%	94.05%	96.32%	98.87%	98.39%	98.39%	98.39%	98.55%
22	0.67%	0.80%	96.19%	98.81%	99.92%	99.37%	99.37%	99.37%	99.63%
23	0.65%	0.79%	97.00%	98.80%	100.00%	99.54%	99.54%	99.54%	99.48%
24	0.94%	0.70%	97.73%	98.94%	100.00%	99.84%	99.84%	99.84%	99.90%
25	0.48%	0.70%	99.32%	98.02%	99.53%	99.65%	99.66%	99.65%	99.96%
26	0.41%	0.64%	99.85%	94.45%	99.86%	99.85%	99.85%	99.85%	99.85%
27	85.76%	0.58%	100.00%	94.19%	96.50%	96.78%	96.79%	96.78%	100.00%
28	0.45%	0.57%	78.03%	98.17%	99.15%	99.46%	99.46%	99.46%	99.54%
29	0.46%	0.55%	78.03%	98.27%	100.00%	100.00%	100.00%	100.00%	99.37%
30	0.41%	0.49%	78.03%	96.38%	97.87%	100.00%	100.00%	100.00%	100.00%
Average	25.64%	10.84%	88.91%	97.36%	99.67%	99.74%	99.74%	99.74%	99.85%

8.1.3 Runtime

Table 3: Runtime (seconds)

n	grid	rand	grid sqp	rand sqp	rand 20	Parallel	Perturb 1	Perturb 2	sqp 35
1	0.015568	0.883415	0.003593	0.009802	0.107696	0.155642	0.226111	0.386466	0.077765
2	0.026711	0.059098	0.003401	0.006303	0.1301	0.082351	0.142451	0.33148	0.111342
3	0.055615	0.074001	0.00549	0.010663	0.164264	0.118817	0.184496	0.383103	0.182326
4	0.041413	0.056536	0.006888	0.014428	0.260387	0.103018	0.189925	0.455088	0.21762
5	0.0837	0.156737	0.005533	0.023171	0.461883	0.131644	0.202903	0.396174	0.229117
6	0.220135	0.130633	0.006489	0.0153	0.705986	0.153272	0.227196	0.42528	0.412493
7	0.055676	0.092676	0.005737	0.030831	0.777701	0.187225	0.275446	0.551253	0.355817
8	0.098526	0.132928	0.007142	0.05129	0.973926	0.217236	0.303123	0.559276	0.339326
9	0.707199	0.175877	0.006063	0.04474	1.477166	0.332679	0.422557	0.680541	0.52612
10	0.296321	0.306521	0.009264	0.096984	2.194745	0.376739	0.4802	0.781721	0.671002
11	0.340601	0.297159	0.007553	0.101821	2.482488	0.518258	0.642848	0.977036	0.879765
12	0.237099	0.274457	0.008448	0.168527	3.280143	0.584664	0.725215	1.415835	1.196418
13	0.327118	0.449398	0.008276	0.163475	4.187893	0.713087	0.842992	1.265563	1.228153
14	0.750134	0.608445	0.010367	0.156508	5.532638	0.970546	1.108167	1.545827	1.850586
15	0.934254	0.436517	0.011006	0.309424	7.671443	1.205885	1.913552	3.494452	2.239305
16	1.235897	0.738624	0.016566	0.549154	8.45249	1.395762	1.74723	2.875249	2.289423
17	1.347516	0.59349	0.019318	0.606859	9.945808	1.950625	2.334008	3.307041	2.856112
18	0.639277	1.505035	0.022318	0.658788	13.14017	1.786072	2.025751	2.710135	3.216753
19	1.23742	0.823178	0.017778	0.65159	13.71634	2.178947	2.562249	3.921469	4.187264
20	2.830422	1.158872	0.030484	0.689184	14.35938	3.509281	3.776005	4.631732	5.133447
21	1.530748	1.486728	0.024789	0.696966	17.45225	5.758877	7.437383	8.870093	5.92002
22	1.814716	1.411554	0.027954	1.0865	21.3273	4.085926	4.692818	6.76501	6.889294
23	1.384986	2.617436	0.034772	2.085649	24.37232	4.448926	5.030734	6.833354	7.718257
24	2.336498	1.404037	0.03372	1.292674	35.00329	5.744689	6.527629	16.85117	11.06273
25	2.438462	2.109771	0.036959	2.301771	33.65035	6.064199	7.273592	10.98586	10.45301
26	3.63278	2.222526	0.039896	1.577639	34.24522	6.009504	6.578128	9.36958	9.615641
27	1.126429	2.386109	0.064658	1.69191	40.46089	6.555639	7.490956	10.58849	11.97759
28	1.615061	2.133165	0.072786	1.959598	40.64584	7.381869	8.057885	10.20922	13.30296
29	2.549478	2.170413	0.07617	3.075593	47.16913	9.298717	10.06385	12.21254	16.77599
30	3.056467	2.575899	0.081609	3.343607	59.26592	10.90935	11.62923	14.47198	17.10778
Total	32.96623	29.47123	0.705023	23.47075	443.6152	82.92945	95.11463	138.252	139.0234

8.2 Results from Final Algorithm

8.2.1 Breakdown

Table 4: 10-sphere cumulative results

Spheres	10	20	30	40	50	60
% Best	100.00%	99.98%	99.97%	99.89%	99.85%	99.71%
Time(s)	2.134097	42.21642	274.7482	1389.192	5376.46	17285.66

8.2.2 Complete Results

n	Radius	Best	% Best	Time(s)
1	0.5	0.5	100.00%	0.084955
2	0.316987	0.31699	100.00%	0.04872
3	0.292893	0.29289	100.00%	0.063642
4	0.292893	0.29289	100.00%	0.063781
5	0.263932	0.26393	100.00%	0.126567
6	0.257359	0.25736	100.00%	0.177608
7	0.250114	0.25014	99.99%	0.203631
8	0.25	0.25	100.00%	0.235746
9	0.232051	0.23205	100.00%	0.475162
10	0.214286	0.21429	100.00%	0.654286
11	0.207622	0.20762	100.00%	0.725761
12	0.207107	0.20711	100.00%	1.219502
13	0.207107	0.20711	100.00%	2.007784
14	0.207107	0.20711	100.00%	3.055163
15	0.192308	0.19231	100.00%	3.511175
16	0.188659	0.1888	99.93%	4.064785
17	0.188287	0.18869	99.79%	5.672222
18	0.187681	0.18768	100.00%	5.919521
19	0.183185	0.18319	100.00%	6.275828
20	0.178407	0.17841	100.00%	7.630577
21	0.177124	0.17722	99.95%	10.82517
22	0.173139	0.17327	99.92%	12.57844
23	0.171245	0.17182	99.67%	14.23468
24	0.170541	0.17054	100.00%	16.98484
25	0.167728	0.1678	99.96%	21.62259
26	0.166667	0.16691	99.85%	21.24235
27	0.166667	0.16667	100.00%	28.15318
28	0.160189	0.16019	100.00%	28.20343
29	0.160189	0.16019	100.00%	36.25911
30	0.160189	0.16019	100.00%	42.42797
31	0.160189	0.16019	100.00%	48.78151
32	0.160189	0.16019	100.00%	56.32399
33	0.1535	0.15461	99.28%	66.39057
34	0.151671	0.15212	99.71%	78.99141
35	0.149224	0.15167	98.39%	89.72969
36	0.149309	0.14938	99.95%	99.49989
37	0.14865	0.14906	99.73%	125.1161
38	0.148821	0.14904	99.85%	226.3082
39	0.147617	0.14762	100.00%	150.7034
40	0.146522	0.14706	99.63%	172.5989

n	Radius	Best	% Best	Time(s)
41	0.144812	0.14484	99.98%	206.2061
42	0.142208	0.14305	99.41%	225.3476
43	0.141386	0.14165	99.81%	245.3188
44	0.140574	0.14086	99.80%	270.368
45	0.140033	0.14063	99.58%	326.5905
46	0.138867	0.14025	99.01%	365.2402
47	0.139959	0.13996	100.00%	786.3186
48	0.139959	0.13996	100.00%	455.8377
49	0.136211	0.13643	99.84%	524.7204
50	0.135494	0.13595	99.66%	581.3202
51	0.13429	0.13502	99.46%	580.9131
52	0.1331	0.13382	99.46%	637.8073
53	0.131878	0.13323	98.99%	1586.321
54	0.131026	0.1322	99.11%	856.7583
55	0.130602	0.13122	99.53%	916.9671
56	0.129494	0.13066	99.11%	1030.259
57	0.128851	0.13062	98.65%	1199.101
58	0.127957	0.13062	97.96%	2008.375
59	0.127732	0.13062	97.79%	1400.927
60	0.130602	0.13062	99.99%	1691.769

8.3 Results from Other Algorithms

Table 5: Results from GA and PSO

n	GA (Default)	GA (Improved)	PSO	Best
1	0.40090	0.50000	0.48080	0.50000
2	0.30590	0.31700	0.35900	0.31699
3	0.25620	0.27620	0.32020	0.29289
4	0.24400	0.25150	0.20840	0.29289
5	0.18920	0.24140	0.16190	0.26393
6	0.16140	0.20440	0.14840	0.25736
7	0.15140	0.19160	0.11130	0.25014
8	0.17380	0.17900	0.12990	0.25000
9	0.13290	0.14020	0.11770	0.23205
10	0.15040	0.14490	0.09580	0.21429
11	0.11690	0.14560	0.09240	0.20762
12	0.11670	0.13840	0.09810	0.20711
13	0.10940	0.12240	0.07000	0.20711
14	0.10150	0.10600	0.07440	0.20711
15	0.10950	0.11300	0.08100	0.19231
16	0.08830	0.11400	0.07730	0.18880
17	0.10510	0.10450	0.05860	0.18869
18	0.10110	0.10510	0.07370	0.18768
19	0.08380	0.08170	0.06390	0.18319
20	0.08760	0.09440	0.04100	0.17841
21	0.08910	0.08370	0.04820	0.17722
22	0.09100	0.08480	0.05050	0.17327
23	0.07800	0.09110	0.05040	0.17182
24	0.10470	0.07890	0.07940	0.17054
25	0.07540	0.09650	0.05370	0.16780

9 Appendix - MATLAB Code

Listing 1: Wrapper code for running simulations

```
% Ends any existing parallel pool
delete(gcp('nocreate'))

%Creates new parallel pool according to the local profile of the current pc
parpool

%Initialise the result matrix, to avoid in-loop reallocation
results = zeros(60,4);

% Reads 'actual.m' into environment
run('actual.m')

% Find packings for 1 to 60 spheres
for i = 1:60

    % Perform a scaled number of trials for this number of spheres
    % We used a factor of 12 due to the number of threads available
    [s,r,b,t]=nSpherePack(i,12*ceil(i/4),0);

    % Store radius, runtime and number of spheres
    results(i,1)=i;
    results(i,2)=r;
    results(i,3)=t;

    % Compare to best known values
    results(i,4)=r/act(i);
    results(i,5)=act(i);

end
```

Listing 2: Function for running fmincon simulations

```
% m: number of spheres
% sims: number of simulations to run
% plot: whether or not to output a plot
% Determines a packing for m spheres using a given number of simulations
function [sols,max_rad,best_sol,time] = nSpherePack(m,sims,plot)
tic % start timer

n=m*3+1;

% Define linear constraints and bounds for the NLP
A = [];
b = [];
Aeq = [];
beq = [];
lb = zeros(1,n);
ub = ones(1,n);
ub(n) = 0.5;

% Define nonlinear constraints and the objective function
nonlcon = @nonlcon2;
fun = @objecfun_n_sph;

```

```
% Change default tolerances and algorithm
options = optimoptions('fmincon','Algorithm','sqp', ...
    'MaxFunEvals',100000,'TolX',1.e-6,'TolFun',1.e-6,'MaxIter',10000);

% Preallocate the matrix of solutions
sols=zeros(sims,n);

% Run simulations in parallel
parfor i=1:sims

    % Randomise initial condition
    x0=rand(1,n);

    % Generate and store local optimum for current initial condition
    optimRes = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options);
    sols(i,:)=optimRes;

end
time = toc; % end timer

% Determine the most optimal solution
max_rad=max(sols(:,n));
for i = 1:sims
    if sols(i,n) == max_rad
        best_sol = i;
        break
    end
end

% Plots sphere packing if needed
if plot == 1
    sphere_plot(sols(best_sol,:));
end
end
```

Listing 3: Objective function

```
function f = objecfun_n_sph(x)
f = -(x(length(x)));
```

Listing 4: Non-linear constraints

```
function [c,ceq] = nonlcon2(x)
% Last element of x has radius, each sphere has three dimensions
n=(length(x)-1)/3;

% Last element of x will hold the radius value
r=x(length(x));

% Number of non-linear constraints to calculate pairwise distance betweeen
% spheres given by tri(n-1)
nonlin = tri(n-1);

% Generates array of all pairs of spheres, i and j, which must have a
% distance constraint calculated
p = pairs(n);

% row counter to keep track of which row in c to write a constraint to
```

```
row = 1;

% pairwise distance constraints
for i =1:nonlin
    c(row)=4*(r^2)-(x(3*(p(i,1)-1)+1) - x(3*(p(i,2)-1)+1))^2 ...
                  -(x(3*(p(i,1)-1)+2) - x(3*(p(i,2)-1)+2))^2 ...
                  -(x(3*(p(i,1)-1)+3) - x(3*(p(i,2)-1)+3))^2;
    row = row +1;
end

% Cube boundary constraints
for i =1:3*n
    c(row)=r-x(i);
    c(row+3*n)=x(i)+r-1;
    row=row+1;
end
c=c';
ceq=[];
```

Listing 5: Generate all pairwise sphere matchings for nonlcon2.m

```
% Generates all pairwise matchings of spheres
function pair_mat = pairs(n)
pair_mat = [];
row=1;
for i = 1:n
    for j = i+1:n
        pair_mat(row,1)=i;
        pair_mat(row,2)=j;
        row = row +1;
    end
end
```

Listing 6: Generate number of nonlinear constraints required

```
% Generates the nth triangle number
function num = tri(n)
sum=0;
for i=1:n
    sum = sum +i;
end
num=sum;
```

Listing 7: Plot spheres based on outputs of nSpherePack.m.m

```
% For plotting sphere packings
function t = sphere_plot(coords)
figure

n=length(coords);
daspect([1 1 1])
view(30,10)
title(sprintf('Packing of %d spheres in a unit cube - radius: %0.6f', (n-1)/3,coords(n)))
hold on
for i = 1:(n-1)/3
    [x,y,z]=sphere;
    s = [coords(3*i-2),coords(3*i-1),coords(3*i),coords(n)];
    t=surf(x*s(4)+s(1),y*s(4)+s(2),z*s(4)+s(3));
```

```
    hold on
end


end
```

Listing 8: Initialise spheres in grid pattern

```
% Arranges spheres in a basic grid/cubic pattern
function coords=grid_init(n)

    % Approximate radius to 1/ceiling(nearest cuberoot)
    len = ceil(nthroot(n,3));
    r= 0.5/len;
    o=[r,r,r];
    coords=zeros(1,3*n+1);
    num=0;

    % Place sphere coordinates
    for k = 1:len
        start=[o(1),o(2),o(3)+(k-1)*2*r];
        for j = 1:len
            new = [start(1),start(2)+(j-1)*2*r,start(3)];
            for i = 1:len
                if num >=n
                    break
                end
                coords(3*num+1) = new(1)+(i-1)*2*r;
                coords(3*num+2) = new(2);
                coords(3*num+3) = new(3);
                num = num +1;
            end
        end
    end
    coords(3*n+1)=r;
end
```

Listing 9: Wrapper code for running simulations under perturbation method

```
% Gets results for 20 simulations, perturbation and stepped perturbation

% Preallocate result matrices
results_p=zeros(30,5);
results_ps=zeros(30,5);
results_r=zeros(30,5);

% Settings for fmincon
options = optimoptions('fmincon','Algorithm','sqp', ...
    'MaxFunEvals',100000,'TolX',1.e-6,'TolFun',1.e-6,'MaxIter',10000);


for i=1:30
    tic % start timer

    n=i*3+1;
    sols=zeros(35,n);

    % Constraints and objective function
```

```
    fun = @objecfun_n_sph;
    A = [];
    b = [];
    Aeq = [];
    beq = [];
    lb = zeros(1,n);
    ub = ones(1,n);
    ub(n) = 0.5;
    nonlcon = @nonlcon2;

    % Run 20 random trials
    parfor j = 1:20
        x0=rand(1,n);
        x1 = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options);
        sols(j,:)=x1;
    end

    % Determine best result
    max_rad = max(sols(:,n));
    for k=1:20
        if sols(k,n) == max_rad
            best_sol = k;
            break
        end
    end

    % Store best result and time taken
    best=sols(best_sol,:);
    t_r=toc;

    % Use basic perturbation method, measure time taken
    tic
    x_opt_p = perturb(best,0.05,20);
    t_p=toc+t_r;

    % Use stepped perturbation method, measure time taken
    tic
    x_opt_ps = perturb_steps(best,0.1,20,4);
    t_ps=toc+t_r;

    % Store 20 random trial results
    results_r(i,1)=best(n);
    results_r(i,2)=t_r;
    results_r(i,3)=i;
    results_r(i,4)=best(n)/act(i);
    results_r(i,5)=act(i);

    % Store basic perturbations results
    results_p(i,1)=x_opt_p(n);
    results_p(i,2)=t_p;
    results_p(i,3)=i;
    results_p(i,4)=x_opt_p(n)/act(i);
    results_p(i,5)=act(i);

    % Store stepped perturbation results
    results_ps(i,1)=x_opt_ps(n);
    results_ps(i,2)=t_ps;
```

```
    results_ps(i,3)=i;
    results_ps(i,4)=x_opt_ps(n)/act(i);
    results_ps(i,5)=act(i);
end
```

Listing 10: Basic perturbation method

```
% Carries out a given number of perturbations on a given packing
function x_opt = perturb(x0,mvmt,reps)

num_sph=(length(x0)-1)/3;
n=length(x0);

% Set up bounds and constraints
A = [];
b = [];
Aeq = [];
beq = [];
lb = zeros(1,n);
ub = ones(1,n);
ub(n) = 0.5;
nonlcon = @nonlcon2;
fun = @objecfun_n_sph;

% Change fmincon options, as in main program
options = optimoptions('fmincon','Algorithm','sqp', ...
    'MaxFunEvals',50000,'TolX',1.e-10,'TolFun',1.e-10,'MaxIter',10000);

% Get initial optimum packing and radius
r=x0(n);
x_opt=x0;

% Preallocate matrix for solutions
p_sols = zeros(reps,n);

% Do perturbations in parallel, store the results
parfor i =1:reps
    x1=x0+(-0.5+rand(1,n))*mvmt*r;
    x_new = fmincon(fun,x1,A,b,Aeq,beq,lb,ub,nonlcon,options);
    p_sols(i,:)=x_new;
end

% Determine best results from all pertubations
best_rad = max(p_sols(:,n));
for i = 1:reps
    if p_sols(i,n)==best_rad
        x_opt = p_sols(i,:);
        break
    end
end
end
```

Listing 11: Stepped perturbation method

```
% Carries out a given number of perturbations at each of a given
% number of steps
function x_opt = perturb_steps(x0,mvmt,reps,steps)

```

```
% Determines the amount to perturb by at each step
mvmt_steps = mvmt/steps:mvmt/steps:mvmt;

% Set up objective function and constraints
fun = @objecfun_n_sph;
num_sph=(length(x0)-1)/3;
n=length(x0);
x0=x0;
A = [];
b = [];
Aeq = [];
beq = [];
lb = zeros(1,n);
ub = ones(1,n);
ub(n) = 0.5;
nonlcon = @nonlcon2;

% Store inital configuration and radius
r=x0(n);
x_opt=x0;

% Options for fmincon
options = optimoptions('fmincon','Algorithm','sqp', ...
    'MaxFunEvals',50000,'TolX',1.e-10,'TolFun',1.e-10,'MaxIter',10000);

% Preallocate matrix for solutions
ps_sols = zeros(reps*steps,n);

% Do perturbations at each step, store the results
step = 0;
count = 1;
for s = mvmt_steps

    count = count + reps*step;
    parfor i =count:count+reps-1
        x1=x0+(-0.5+rand(1,n))*s*r;
        x_new = fmincon(fun,x1,A,b,Aeq,beq,lb,ub,nonlcon,options);
        ps_sols(i,:)= x_new;
    end
    step=step+1;
end

% Find best result
best_rad = max(ps_sols(:,n));
for i = 1:reps
    if ps_sols(i,n)==best_rad
        x_opt = ps_sols(i,:);
        break
    end
end
end
```

Listing 12: Wrapper code for GA, PSO

```
% Ends any existing parallel pool
delete(gcp('nocreate'))

%Creates new parallel pool according to the local profile of the current pc
```

```
parpool

%Initialise the result matrix, to avoid in loop reallocation
results_ga = zeros(25,4);
results_pso = zeros(25,4);

% Reads 'actual.m' into environment
run('actual.m')

for i = 1:25

    if i == 11
        parpool
    end

    %%% Genetic Algo
    [s,r,b,t]=nSpherePack_ga(i,1,0);
    results_ga(i,1)=r;
    results_ga(i,2)=t;
    results_ga(i,3)=i;

    % Compare to best known values
    results_ga(i,4)=r/act(i);
    results_ga(i,5)=act(i);

    %%% Particle swarm - often breaches a bound, so check for this
    [s_pso,r_pso,b_pso,t_pso]=nSpherePack_pso(i,1,0);
    results_pso(i,1)=r_pso;
    results_pso(i,2)=t_pso;
    results_pso(i,3)=i;

    % Compare to best known values
    results_pso(i,4)=r_pso/act(i);
    results_pso(i,5)=act(i);
    % check for bound breaches
    results_pso(i,6) = sum(s_pso > 1) + sum(s_pso < 0);

end
```

Listing 13: Function for running GA

```
%%% Genetic algorithm
% https://au.mathworks.com/help/gads/examples/constrained-minimization-using-the-genetic-algorithm.
    html
% m = number of spheres
% sims: number of simulations to run
% plot: whether or not to output a plot
% Determines a packing for m spheres using a given number of simulations
% with GA
function [best,max_rad,best_loc,time] = nSpherePack_ga(m,sims,plot)

fprintf('nSpherePack_ga - %d spheres, %d sims...\n',m,sims)

tic % start timer

n=m*3+1;

% Define linear constraints and bounds for the NLP
```

```
A = [];
b = [];
Aeq = [];
beq = [];
lb = zeros(1,n);
ub = ones(1,n);
ub(n) = 0.5;

% Define nonlinear constraints and the objective function
nonlcon = @nonlcon2;
fun = @objecfun_n_sph;

if m > 10
    % Then run parallel
    options = optimoptions('ga','NonlinearConstraintAlgorithm','penalty',...
    'MutationFcn',{@mutationadaptfeasible},...
    'SelectionFcn',{@selectiontournament,4},...
    'MaxGenerations',200*n,...
    'PopulationSize',1000,...
    'InitialPopulationRange',[0;1],...
    'ConstraintTolerance', 1.e-6,'UseParallel', true);

    [optimRes,fval,exitFlag,Output] = ga(fun,n,[],[],[],[],lb,ub, nonlcon, options);
    fprintf('The number of generations was : %d\n', Output.generations);
    fprintf('The number of function evaluations was : %d\n', Output.funccount);
    fprintf('The best radius found was : %g\n', -fval);

else
    % No parallel
    options = optimoptions('ga','NonlinearConstraintAlgorithm','penalty',...
        'MutationFcn',{@mutationadaptfeasible},...
        'SelectionFcn',{@selectiontournament,4},...
        'MaxGenerations',200*n,...
        'PopulationSize',1000,...
        'InitialPopulationRange',[0;1],...
        'ConstraintTolerance', 1.e-6);

    [optimRes,fval,exitFlag,Output] = ga(fun,n,[],[],[],[],lb,ub, nonlcon, options);
    fprintf('The number of generations was : %d\n', Output.generations);
    fprintf('The number of function evaluations was : %d\n', Output.funccount);
    fprintf('The best radius found was : %g\n', -fval);
end

max_rad = optimRes(n);
best_loc = 1;
best = optimRes;

time = toc; % end timer

% Plots sphere packing if needed
if plot == 1
    sphere_plot(best);
end
end
```

Listing 14: Function for running PSO

```
%%% Constrained Particle swarm algo
```

```
% https://au.mathworks.com/matlabcentral/fileexchange/25986-constrained-particle-swarm-optimization
% m = number of spheres
% sims: number of simulations to run
% plot: whether or not to output a plot
% Determines a packing for m spheres using a given number of simulations
% with PSO
function [best,max_rad,best_loc,time] = nSpherePack_pso(m,sims,plot)

fprintf('nSpherePack_pso - %d spheres, %d sims...\n',m,sims)

tic % start timer

n=m*3+1;

% Define linear constraints and bounds for the NLP
A = [];
b = [];
Aeq = [];
beq = [];
lb = zeros(1,n);
ub = ones(1,n);
ub(n) = 0.5;

% Define nonlinear constraints and the objective function
nonlcon = @nonlcon2;
fun = @objecfun_n_sph;

    options = psooptimset('pso',...
        'Generations',500*n,...
        'PopulationSize', 100*n,...
        'ConstraintTolerance', 1.e-6);

    [optimRes,fval,exitFlag] = pso(fun,n,[],[],[],[],lb,ub,nonlcon, options);

max_rad = optimRes(n);
best_loc = 1;
best = optimRes;


time = toc; % end timer

% Plots sphere packing if needed
if plot == 1
    sphere_plot(best);
end
end
```

Listing 15: Stored table of best results

```
% Best known radii up to 80 spheres
act =    [0.50000 0.31699 0.29289 0.29289 0.26393 0.25736 0.25014 0.25000...
          0.23205 0.21429 0.20762 0.20711 0.20711 0.20711 0.19231 0.18880...
          0.18869 0.18768 0.18319 0.17841 0.17722 0.17327 0.17182 0.17054...
          0.16780 0.16691 0.16667 0.16019 0.16019 0.16019 0.16019 0.16019...
          0.15461 0.15212 0.15167 0.14938 0.14906 0.14904 0.14762 0.14706...
          0.14484 0.14305 0.14165 0.14086 0.14063 0.14025 0.13996 0.13996...
          0.13643 0.13595 0.13502 0.13382 0.13323 0.13220 0.13122 0.13066...
          0.13062 0.13062 0.13062 0.13062 0.13062 0.13062 0.13062 0.12674...
```

```
10          0.12618 0.12574 0.12448 0.12344 0.12314 0.12304 0.12303 0.12303...
11          0.12179 0.12134 0.12132 0.12018 0.11906 0.11855 0.11801 0.11779];
```