SoftwareX 18 (2022) 101051

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

3DRSP: Matlab-based random sphere packing code in three dimensions

Travis J. Black, Alexei F. Cheviakov *

Department of Mathematics and Statistics, University of Saskatchewan, Saskatoon, SK, Canada S7N 5E6

ARTICLE INFO

Article history:
Received 29 October 2021
Received in revised form 15 March 2022
Accepted 15 March 2022

Keywords:
Random sphere packing
Three-dimensional domain
Probability distribution
Matlab

ABSTRACT

A `Matlab`-based computational procedure is proposed to fill a given volume with spheres whose radii are randomly picked from any specified probability distribution supported by `Matlab`. The general program sequence and examples of filling a unit cube, a parallelepiped, and a concave domain between two hemispherical surfaces, with spheres whose radii are drawn from the Weibull and Gamma distributions, are presented. A sample application to the numerical modeling of bond formation between particles heated by a laser beam in powder bed 3D printing process is considered.

© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Code metadata

Current code version	1.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00213
Permanent link to Reproducible Capsule	
Legal Code License	Public domain
Code versioning system used	none
Software code languages, tools, and services used	Matlab 2021a
Compilation requirements, operating environments & dependencies	
If available Link to developer documentation/manual	
Support email for questions	shevyakov@math.usask.ca

1. Introduction

The proposed `Matlab`-based code fills a given three-dimensional domain with spheres of radii following any prescribed random distribution of sphere available in `Matlab`. The motivation behind the creation of this code was to build a Discrete Element Method (DEM) model of the powder bed 3D printing process. The initial step in such a model was the determination of the initial location of a packing of individual particles within the powder bed domain, and which particles were in contact. This posed a problem to which no simple solution was readily available. Indeed, finding an optimal packing of unequal spheres is a challenging task [1]. Previous DEM models (e.g., Refs. [2,3]) relied on the particle dynamics approach to determine location with gravitational and inter-particle forces; the interaction and final positions of the spheres was determined using Newtonian mechanics. This required particle–particle and particle–boundary collision detection, as well as calculation of contact forces [2]. For a large number of particles, such calculations can be highly computationally expensive. Since our only concern was the final placement on the particles, the current code was designed to bypass the dynamics of powder settling and find a way to determine the rest location of the particles based on geometrical considerations.

Random sphere packing has broad applications, including DEM modeling, granular dynamics, radiosurgery for treating brain tumors [1], optimal packing problems, etc. Other sphere packing methods have similar aims [4,5], however they did not meet our needs. Vast literature is dedicated to a related but different problem of random packing of equally-sized spheres (see, e.g., Refs. [6,7] and references therein).

The goal for the program presented below was not to find the *optimal* packing, but rather a packing of unequal spheres which closely models a realistic packing of metal powder particles in

* Corresponding author.
E-mail addresses: tjb235@usask.ca (Travis J. Black), shevyakov@math.usask.ca (Alexei F. Cheviakov).

https://doi.org/10.1016/j.softx.2022.101051
2352-7110/© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

the powder bed printing process. In the simplest setting where the total domain to be filled can be represented as a union of parallelepiped-shaped bricks, initially, a brick is filled with the distribution of spheres, and then is used as a building block to achieve the desired volume. The symmetry of the brick is exploited to keep track of particle contacts.

Two related methods were developed, and are available to the user within the current Matlab-based package. Each consists of a part responsible for filling a single brick and a part that builds the volume.

For Method 1, the approach to filling the unit brick, an arbitrary parallelepiped-shaped volume, the main sphere fitting function, a typical program sequence and three run examples are discussed in Section 2. The first method randomly fills the edges of the unit brick, then the faces, and then the volume, which results in a unit brick filled with a non-symmetric spherical distribution. The total domain may coincide with a single brick, or be made of several bricks in x, y, and z directions; in the latter case, the total domain is constructed by reflections of the unit brick about its faces, providing boundary sphere contacts.

For the second method, details and examples are provided in Section 3. In Method 2, unlike Method 1, for the unit brick, only one edge is filled in each direction, and four parallel copies of each are made. Similarly, only three faces in each plane are filled with random spheres, and are copied onto the opposite ones. This yields a unit brick that has identical opposite faces (but a non-symmetric volume filling), and hence a direct copy–paste of unit bricks can be used to fill a larger total domain. Another difference of Method 2 from Method 1 is that instead of being fully inside of the unit brick and touching the brick faces, in Method 2, centers of the spheres on each face are located on the brick faces themselves.

Example 3 (Section 4) illustrates an application of the geometric approach to create a spherical filling of a more complex-shaped domain: a parallelepiped with the subtraction of two hemispheres centered in the middles of two opposite faces in the x-direction.

A physical example containing a simple model considering discrete laser-induced heat-based bonding in powder bed 3D printing process is considered in Section 5, following Ref. [8]. The physical principles and constants are described in Section 5.1, and a result of a simulated print of a small square are presented in Section 5.2.

In examples used in the current work, Weibull and Gamma probability distribution of spherical radii [9,10] were employed. The Weibull and Gamma distribution parameters were chosen to correspond to powder bed additive manufacturing involving steel spheres. The presented software supports all probability distributions provided in Matlab.

The paper is concluded with a summary discussion in Section 6.

2. Filling a domain with spheres: Method 1

The first method, as well as the second method described later, can be used to fill any parallelepiped-shaped domain $\mathcal{V}$ with a given random distribution of spheres.

The domain $\mathcal{V}$ can be filled either in a completely random manner, or for a quicker computation, it can be subdivided into smaller standard parallelepipeds ("unit bricks"). In the latter case, a single brick would be filled with spheres randomly, and bricks can be copied and joined, as explained below, any prescribed number of times in x, y, and z directions, to create a filling of $\mathcal{V}$. In the former case, the full domain $\mathcal{V}$ is treated as a single unit brick.

During the spherical filling, positions and radii of random spheres are recorded, as well as pairwise connections between touching spheres, and sets of sphere indices corresponding to spheres lying on each face of $\mathcal{V}$.

2.1. Method 1: filling the unit brick

The initial step in the first method of filling up a unit brick with a given distribution of spheres consists in placing a sphere of average radius in each corner of the brick, then filling the edges between each adjacent corners with contacting spheres with sizes drawn randomly from the same distribution, and then filling faces the same way. After all faces are finished, the remaining volume is filled. When a new sphere is placed, neighboring spheres in contact with the new one are recorded. This way, when the volume is filled, there is a list containing all pairs of spheres that are in contact. As new spheres are placed, a list of other spheres that are in contact with the given one is kept, based on a constant dimensionless parameter ε that specifies acceptable separation/overlap of two particles to be considered in contact. Particles that are considered "close" (controlled by another constant dimensionless parameter δ) are stored as possible "parents". When a new sphere needs to be placed, to determine its location, the program runs through the list of possible "parents", as explained below, and the new sphere is placed to be in contact with possible parent particles (while we work in 3D, the idea is shown in 2D in Fig. 1). This is achieved by solving a system of equations

$$\|\mathbf{x}_n - \mathbf{x}_1\| = R_n + R_1, \quad \|\mathbf{x}_n - \mathbf{x}_2\| = R_n + R_2, \quad \|\mathbf{x}_n - \mathbf{x}_3\| = R_n + R_3,$$

where $\mathbf{x}_i \in \mathbb{R}^3$, $i = 1, 2, 3$ is the triplet parent spheres, $\mathbf{x}_n$ is the unknown position of the center of the new sphere, and R_i denote the corresponding radius.

Once the new sphere is placed in its putative position, a check is run to see if it overlaps with any other spheres. If it does, then the current putative location is discarded, and the sphere is matched with the next set of possible parents. Once the sphere is placed and does not intersect with any other spheres, its location and contacts are stored, and the next sphere's radius is randomly drawn from the given distribution. If a given new sphere does not fit with any of the parents, it is discarded, and a new sphere radius is randomly drawn.

We note that in Method 1, unlike the following Method 2, each edge and face of the unit brick is covered with a different random set of spheres.

2.2. Method 1: filling a parallelepiped-shaped volume

When a unit brick is filled, it may be used "as is" to represent the full domain $\mathcal{V}$ filled with spheres in a non-symmetric random manner, or as a building block to build larger volumes in a relatively small amount of computing time, by exploiting the symmetry of the unit brick. Method 1 of volume filling is employed to generate a parallelepiped of size $m \times n \times p$, where m, n and p are the numbers of unit bricks in x, y and z direction respectively.

In Method 1, information about the spheres that are in contact with the unit brick's faces is used to fill the desired volume by reflecting the unit brick symmetrically with respect to its face planes, so that spheres on the faces would be in direct contact with spheres in a symmetric copy of the unit brick. This process is repeated until desired length is met, and then is repeated in the perpendicular directions. Thus instead of a time-consuming process of space filling with spheres, the coordinates of spheres in additional bricks are computed simply through symmetry transformations of coordinates of spheres in the original unit brick, and radii and contact information are copied directly.

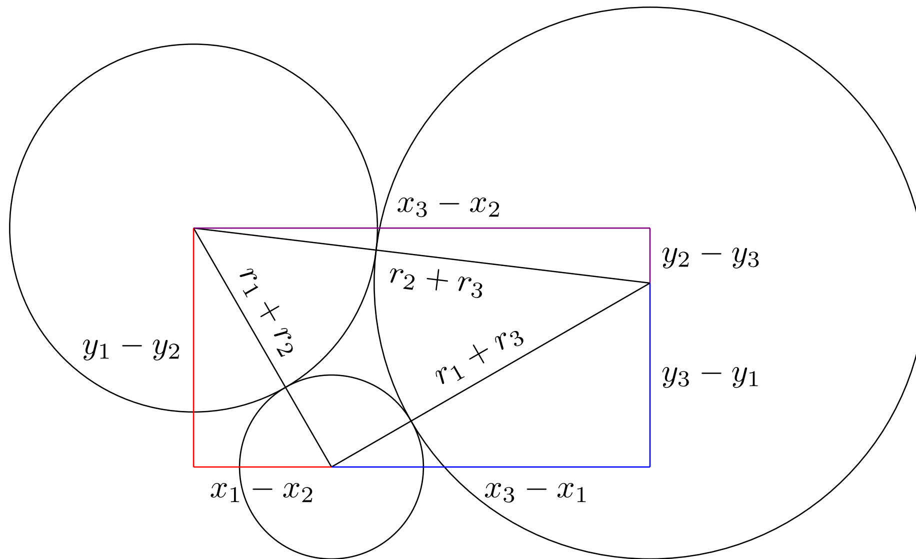

Fig. 1. New sphere in contact with parent spheres: a 2D cartoon.

2.3. Method 1: the main sphere fitting function

The `Matlab` function `Method1GenerateSpheres.m` implements Method 1 of unit brick generation and volume filling as described above. The input and output parameters are given in the order of appearance.

Input parameters:

- `ProbabilityDistr`: a `Matlab` probability distribution of the radii of spherical particles. This object can be created using the `Matlab makedist` function. For a given distribution, we call $\bar{r}$ the average sphere radius.
- `FaceGoal`: fraction of the unit brick face area covered by spheres in contact with it.
- `BodyGoal`: fraction of the unit brick volume filled by spheres.
- `SphereContactParameter`: The contact parameter ε, within $[0, 1]$. If two spherical particles are within `SphereContactParameter` $\times\, \bar{r}$ of each other, they are considered to be in contact.
- `ParentParameter`: the "parent parameter" δ, within $[0, 1]$. If two spherical particles are within `ParentParameter` $\times\, \bar{r}$ of each other, they are considered to be potential "parents" to further particles.
- `BrickSideLengths`: an array of three values corresponding to absolute lengths (in physical units) of the unit brick sides along x, y, and z.
- `BrickNumbers`: an array of three integer values specifying numbers of copies of the unit brick in x, y, and z directions required to build the total volume $\mathcal{V}$.

The function `Method1GenerateSpheres.m` uses the prescribed probability distribution to fill with spheres the parallelepiped $\mathcal{V}$ having physical dimensions in x, y and z directions given by

$$L_i = \texttt{BrickSideLengths(1)} \times \texttt{BrickNumbers(i)}, \quad i = 1, 2, 3. \tag{2.1}$$

Output parameters:

- `FinalNSpheres`: The total final number of spheres in the total domain $\mathcal{V}$.
- `UnitBrickNSpheres`: The number of spheres in each unit brick.
- `Positions`: a matrix
$$\begin{pmatrix} x_1 & \cdots & x_n & \cdots \\ y_1 & \cdots & y_n & \cdots \\ z_1 & \cdots & z_n & \cdots \end{pmatrix}$$
of dimension $3 \times$ `FinalNSpheres`; the first row stores the x-coordinates, second row stores the y-coordinates, and the third row the z-coordinates of all spheres in $\mathcal{V}$. Thus the nth column of the `Positions` matrix gives the coordinates of the nth sphere.
- `Radii`: a $1 \times$ `FinalNSpheres` matrix that stores the radii of all spheres in the whole domain $\mathcal{V}$. The nth entry is the radius of the nth sphere.
- `Contacts`: keeps track of which particles are in contact. This matrix consists of two columns; a pair of entries in the same row is the pair of indices of two spheres that are in contact.
- `ListXmin`, `ListYmin`, `ListZmin`, `ListXmax`, `ListYmax`, `ListZmax`: single-column matrices, each storing all indices of spherical particles in the `Positions` matrix that are in contact with the respective boundaries of the total domain $\mathcal{V}$ corresponding to minimal x, minimal y, minimal z, maximal x, maximal y, and maximal z.

2.4. Method 1: a typical program sequence and run examples

As a run example for the first method of volume filling, we choose the Weibull distribution [9,10] for the sphere radii, given by the probability density function (PDF)

$$f(r; \lambda, k) = \frac{k}{\lambda}\left(\frac{r}{\lambda}\right)^{k-1} e^{-(r/\lambda)^k}, \tag{2.2}$$

where $r \geq 0$ is the dimensional random variable describing the sphere radius, $\lambda > 0$ is the scale parameter measured in the same length units as the random variable r, and $k > 0$ is the dimensionless shape parameter. The distribution (2.2) has the mean value

$$\bar{r} = \lambda\, \Gamma\left(1 + \frac{1}{k}\right), \tag{2.3}$$

where Γ is the gamma function. For the current example, we choose random sphere parameters corresponding to powder bed 3D printing process with 316L stainless steel powder [9],

$$\lambda = 15.7\ \mu\text{m}, \qquad k = 3.55, \qquad \bar{r} \simeq 14.14\ \mu\text{m}. \tag{2.4}$$

We note that in the literature, in particular, that devoted to additive manufacturing, diameters of spherical particles are often used instead of radii. For example, the diameter-based value $\lambda = 31.4\ \mu$m is used in Ref. [9] (see Ref. [9] Table 1, steel sample S2).

The Matlab script `Example1A_Method1_Generate_and_Plot.m` listed in Appendix A below specifies the probability distribution (2.2), and defines the main parameters for the run, including the characteristics of the domain to be filled with spheres, and the variables controlling the sphere sizes (2.4) and the surface and volume fill ratios. The description of some commands and the run parameters used in the script are also listed in Appendix A. The script calls the main volume filling function `Method1GenerateSpheres.m`, saves the data, and plots the resulting graphs.

Example 1A. In the first example run for Method 1, the following input parameters were used.

- `ProbabilityDistr`: Weibull (2.2), (2.4).
- `FaceGoal`: 0.8.
- `BodyGoal`: 0.55.
- `SphereContactParameter`: 0.2.
- `ParentParameter`: 0.5.
- `BrickSideLengths: [1; 1; 1]*std_length`, where `std_length` $= 15\,\bar{r}$.
- `BrickNumbers: [2; 2; 1]`.

We note that a reference value for the face goal parameter can be computed as the ratio of the sphere projection area to the area of a square with side length $2r$. Similarly, the body goal is estimated as the ratio of sphere to circumscribed brick volume ratio, which yields

$$\begin{aligned} &\texttt{FaceGoal} \sim (\pi r^2)/(2r)^2 = \pi/4 \simeq 0.78, \\ &\texttt{BodyGoal} \sim (4\pi r^3/3)/(2r)^3 \simeq 0.52. \end{aligned} \tag{2.5}$$

Values of `FaceGoal` and `BodyGoal` optimal for a specific application can be determined experimentally.

The sphere generating script `Example1A_Method1_Generate_Plot.m` is also used to produce the plots and save figure files for the current example. Fig. 2(a, c, e) shows the unit brick, its internal structure, and the histogram of actual particle sizes compared to the probability density of the given distribution (2.2) for Example 1A.

Example 1B. Here we use the same setup as in Example 1A, with a larger unit brick side length:

`std_length` $= 30\,\bar{r}$,

and consequently, four times as many spheres per unit cube. The unit cube and the sphere size histogram for this example are shown in Fig. 2(b, d, f). In particular, the actual sphere size distribution histogram in Fig. 2(f) is closer to the given Weibull distribution than that for Example 1A (Fig. 2(e)); this is due to an increased freedom of fitting random-sized spheres into a unit cube that is larger (relative to $\bar{r}$) than that in Example 1A.

Fig. 3 show the construction of the total volume $\mathcal{V}$ made of $2 \times 2 \times 1$ unit cubes, and the connectivity graph joining pairs of particles that are in contact, located within the horizontal slab $z = (0.6 \pm 0.3)\times$ `std_length`.

Table 1
Sample desktop computation times T numbers of spheres N in a unit cube for Examples 1A and 1B, for the total domain $\mathcal{V}$ built of $2 \times 2 \times 1$ unit cubes of different side lengths, for two different values of the sphere contact parameter.

Cube side length	Sphere contact parameter	
	0.1	0.2
$15\,\bar{r}$ (Example 1A)	$T = 6$ min, $N = 450$	$T = 2$ min, $N = 412$
$30\,\bar{r}$ (Example 1B)	$T = 187$ min, $N = 3075$	$T = 40$ min, $N = 2867$

Examples 1A and 1B: computation times. Computations were performed on Matlab 2021a, using a Dell workstation with two Xeon processors, 32 logical processors, and 128 GB memory. The computation times listed in Table 1 below depend on the dimensions size of the unit brick and the numbers of unit bricks `BrickNumbers` along the axes to form the total domain $\mathcal{V}$. A strong dependence of the computation times on `SphereContactParameter` ε is observed. (All computations were also tested on an Intel i7-based laptop with one physical and four logical processors and 16 GB memory, resulting on average in 1.5 to three times longer computations).

We also note that plotting can take a relatively long time, similar to the computation time of sphere filling, due to a large number of spherical particles, each represented by a graphical object with multiple faces in the Matlab `sphere` plotting routine.

Example 1C: non-cubical unit bricks. In the current example, we use a function similar to Example 1's `Example1A_SpherePackingMethod1_Generate.m` to call the main sphere fitting function `Method1GenerateSpheres.m` with different parameters:

- `ProbabilityDistr`: Weibull distribution (2.2), (2.4) of sphere radii,
- `FaceGoal`: 0.8,
- `BodyGoal`: 0.55,
- `SphereContactParameter`: 0.2,
- `ParentParameter`: 0.5,
- `BrickSideLengths: [1.2; 1.7; 1]*std_length`, where `std_length` $= 15\,\bar{r}$,
- `BrickNumbers: [4; 4; 2]`.

As a result, the total domain $\mathcal{V}$ of size $4 \times 4 \times 2 = 32$ unit bricks is filled with spheres. Each unit brick is non-cubical, with size lengths specified in `BrickSideLengths`.

Fig. 4 shows the resulting unit brick and the spheres touching the sides corresponding to the minimal and the maximal x-value, as well as the total build of the domain $\mathcal{V}$.

The total computation of the Method 1 sphere filling of the total domain $\mathcal{V}$ using 32 unit bricks specified above, on the same hardware/software configuration took about 10 min. For comparison, if the same volume $\mathcal{V}$ is constructed from four larger bricks instead, that is,

- `BrickSideLengths: [1.2; 1.7; 1]*std_length`, where `std_length` $= 30\,\bar{r}$,
- `BrickNumbers: [2; 2; 1]`,

the computation time is increased to approximately 160 min.

3. Filling a domain with spheres: Method 2

Similarly to the first method, the second method is used to fill a parallelepiped-shaped domain $\mathcal{V}$ with one or more unit

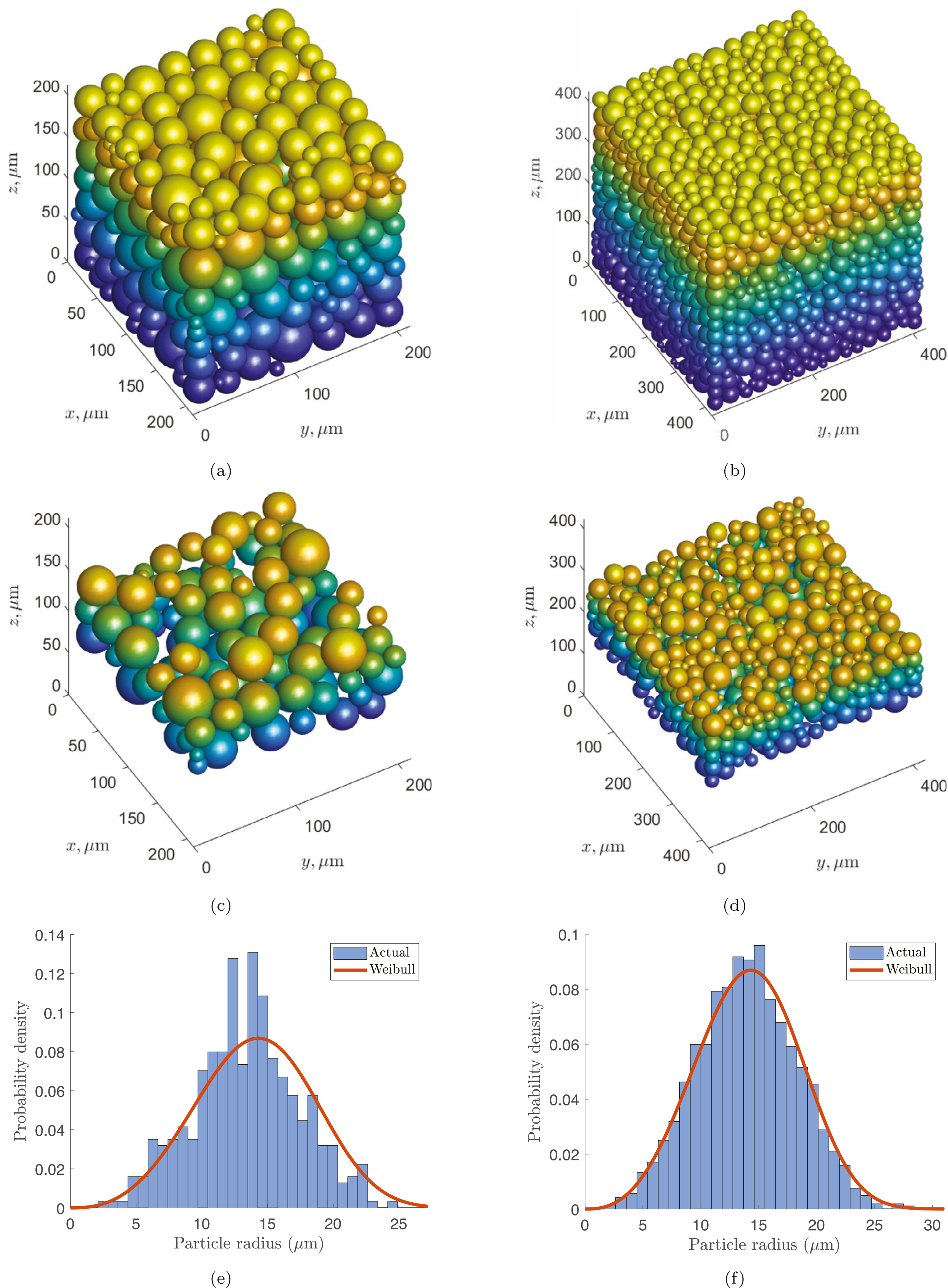

Fig. 2. (a) Example 1A: a unit cube with side length $15\,\bar{r}$ filled with spheres using Method 1 with and Weibull distribution (2.2), (2.4). (c) Spheres in middle 1/3 of the cube, showing the internal structure of the filled cube. (e) Actual sphere radius distribution in the obtained sample, compared to the given Weibull distribution (2.2), (2.4). (b,d,f) Same plots Example 1B: a unit cube with side length $30\,\bar{r}$.

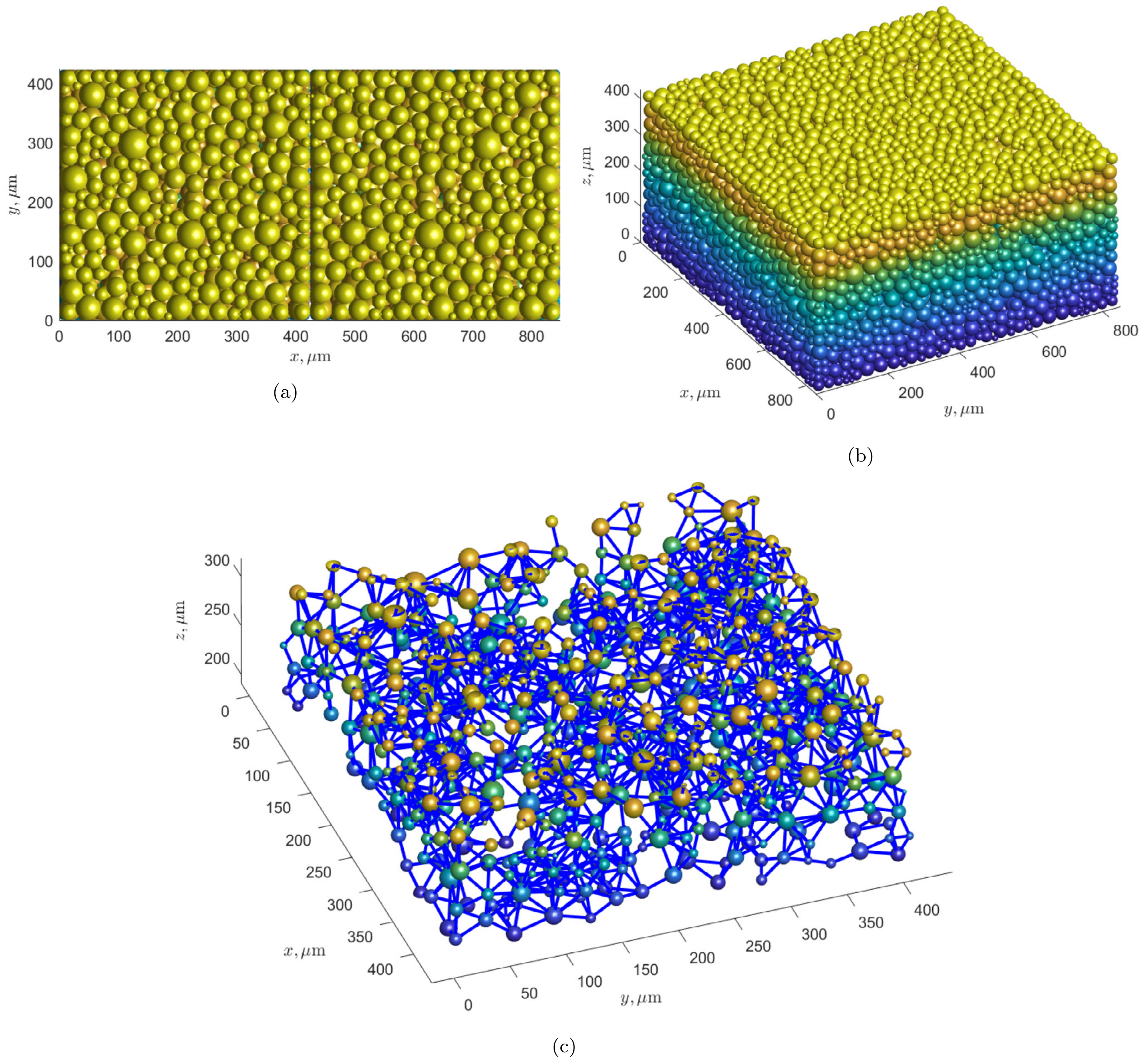

Fig. 3. (a) Example 1B: two unit cubes joined to form a 2 × 1 row — top view. (b) Two rows (four unit cubes) joined to form the final domain $\mathcal{V}$. (c) The connectivity graph for a horizontal slab subdomain of the unit cube of Example 1B (the spheres are scaled down to 40% of their actual sizes to visualize connections).

bricks packed with random spheres whose radii follow a given probability distribution.

One of the main differences in the second method from the first one is the fact that instead of being fully inside of the unit brick and touching the brick faces (Method 1), in Method 2, centers of the spheres on each face are located on the faces themselves. This results in somewhat different-looking unit bricks. Another difference lies in the symmetry of edges and faces. When filling the unit brick, Method 2 starts by placing equal spheres into all corners; then, unlike the first method, only one edge is then filled in each (x, y, and z) direction, and four parallel copies of each edge are made, to fill all 12 edges of the unit brick. Similarly, only three faces (in xy, yz, and xz planes) are filled with random spheres, and these facies are copied onto the opposite ones. Finally, the unit brick volume is filled with random spheres. As a result of this procedure, a unit brick with symmetric faces but a non-symmetric volume filling is obtained. In order to construct a sphere filling for the total domain $\mathcal{V}$ when it is made of several unit bricks, the unit brick is directly copied as many times as required in x, y, and z directions; the face symmetry provides a seamless fit when unit bricks are joined together.

3.1. Method 2: the main sphere fitting function

Parallel to Method 1, Method 2 is implemented in `Matlab`, in the function named `Method2GenerateSpheres.m`, which has the same parameters as the Method 1 function (see Section 2.3). The function uses any prescribed probability distribution, fills the unit brick, and then the domain constructed from unit bricks, as specified. All input and output parameters in the Method 2 sphere

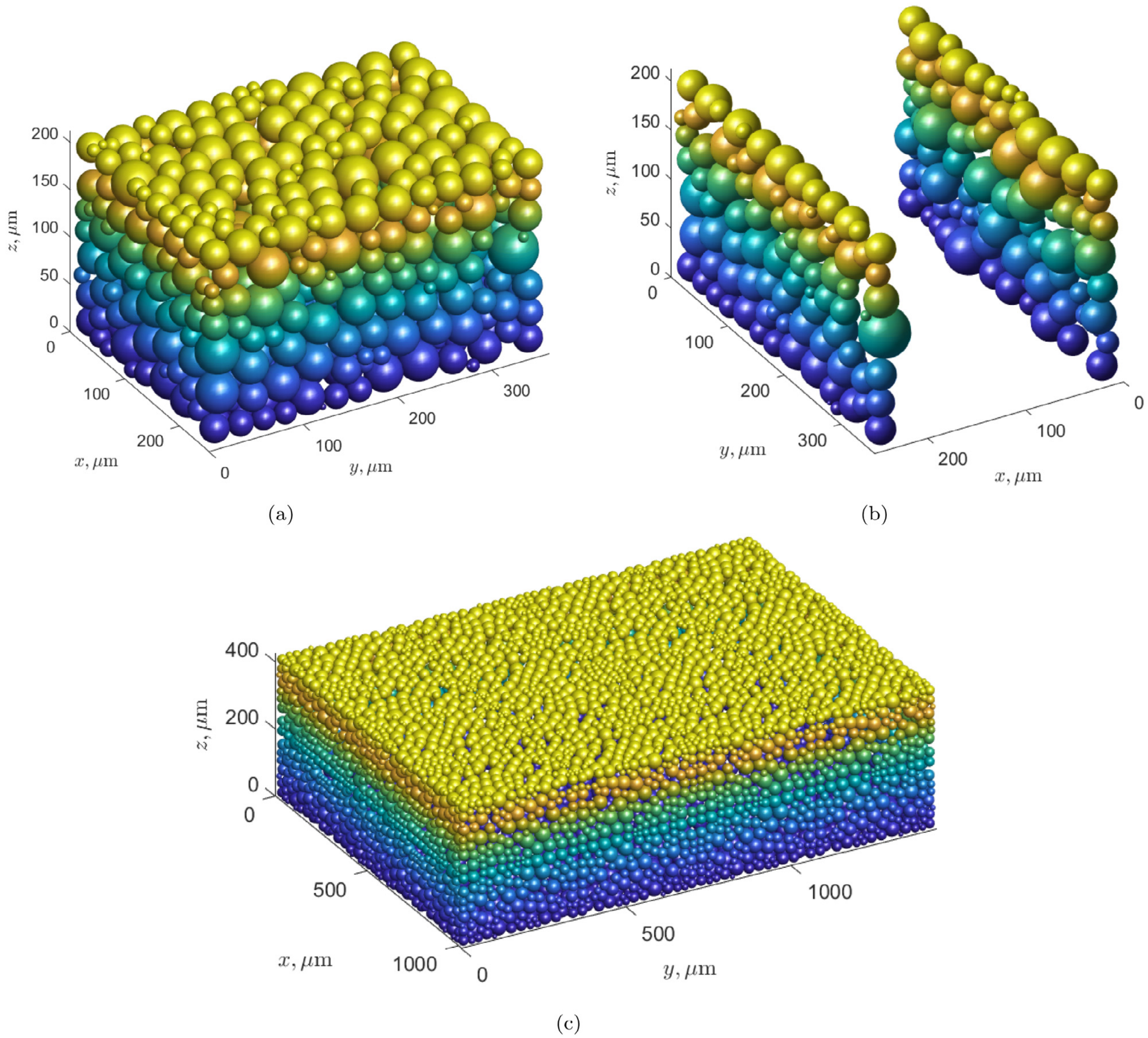

Fig. 4. (a) Example 1C: the unit brick with side lengths $15\,\bar{r} \times [1.2, 1.7, 1]$ filled with spheres using Method 1 with and Weibull distribution (2.2), (2.4). (b) Spheres tangent to the brick sides corresponding to the minimal and the maximal x. (c) the total domain $\mathcal{V}$: $4 \times 4 \times 2$ unit bricks.

fitting function also coincide with those for Method 1, making the two methods fully interchangeable, yet leading to different fillings.

3.2. Method 2: a typical program sequence and a run example

Example 2. In the current example for Method 2, we chose a different probability distribution for sphere sizes: the Gamma distribution given by the probability density function (PDF)

$$f(r; k, \theta) = \frac{1}{\Gamma(k)} \frac{x^{k-1}}{\theta^k} e^{-x/\theta}, \tag{3.1}$$

where $r \geq 0$ is the dimensional random variable describing the sphere radius, $\theta > 0$ is the scale parameter measured in the same units as the random variable r, and $k > 0$ is the dimensionless shape parameter. The distribution (3.1) has the mean value

$$\bar{r} = k\theta. \tag{3.2}$$

For the current example, we choose random sphere parameters similar to Example 1 above, so that the average radius approximately matches that of (2.4)

$$\theta = 7\ \mu\text{m}, \qquad k = 2, \qquad \bar{r} = 14\ \mu\text{m}. \tag{3.3}$$

The script `Example2_Method2_Generate_and_Plot.m` (Appendix B) was used to call the sphere filling function `Method2GenerateSpheres.m` with the input parameters listed below.

- `ProbabilityDistr`: Gamma (3.1), (3.3).
- `FaceGoal`: 1.0.
- `BodyGoal`: 0.9.
- `SphereContactParameter`: 0.1.
- `ParentParameter`: 0.5.
- `BrickSideLengths: [1; 1; 1]*std_length`, where `std_length` $= 30\,\bar{r}$ (see (3.3)).
- `BrickNumbers: [2; 2; 1]`.

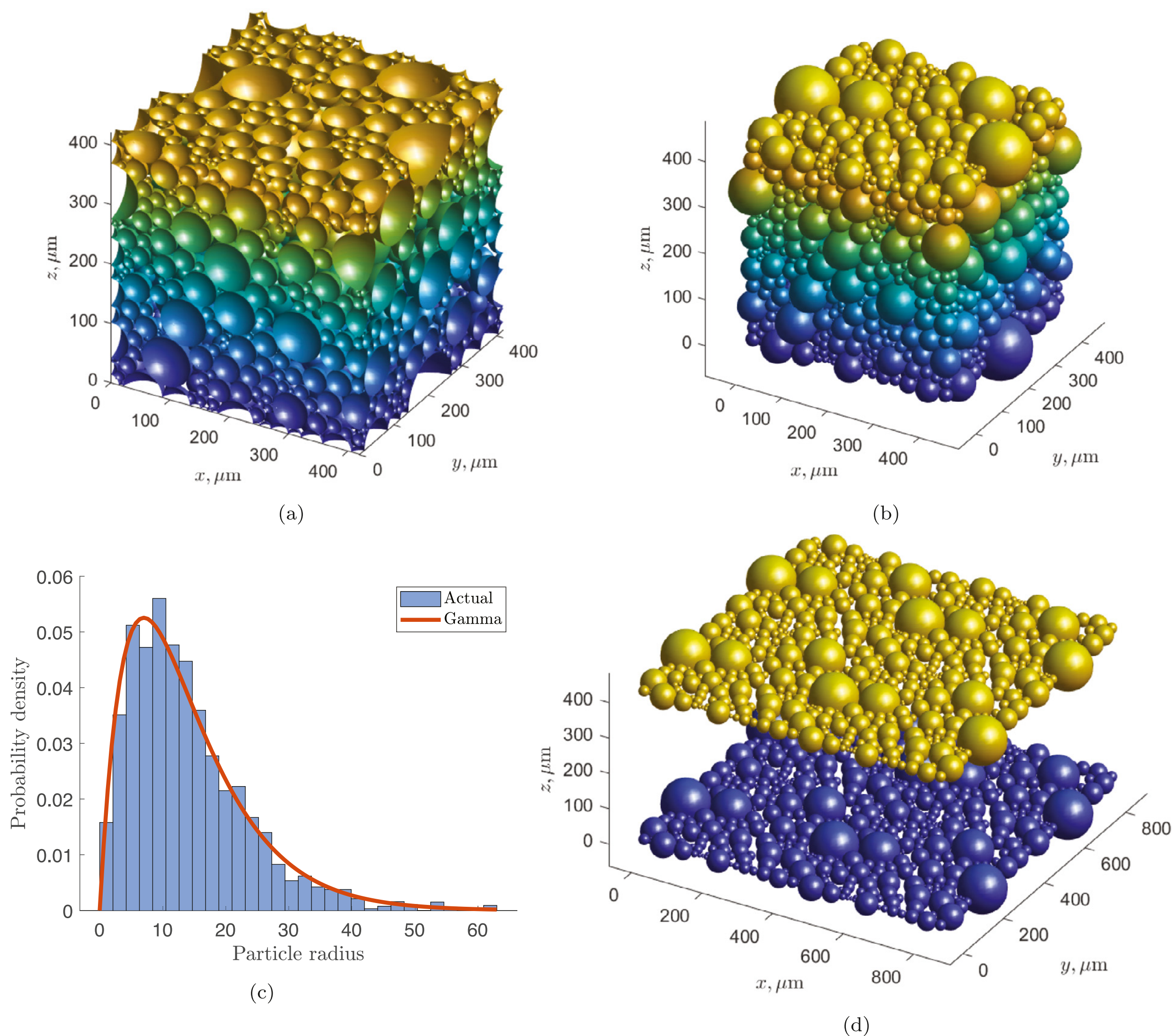

Fig. 5. (a) Example 2: a unit brick (cube) with side length $30\bar{r}$ filled with spheres using Method 2 with Gamma distribution (3.1), (3.3) (unit cube clipped to its size, showing the centers of the spheres on the faces). (b) Same as (a) where the full unit cube is shown in a larger domain. (c) The probability function of the theoretical Gamma distribution (3.1), (3.3) vs. the histogram of actual particle sizes in the unit cube. (d) The top and bottom of the total volume $\mathcal{V}$ made of $2 \times 2 \times 1$ unit bricks contain eight identical copies of a horizontal face of the unit cube.

Here the total domain $\mathcal{V}$ is constructed of $2 \times 2 \times 1$ unit bricks (which are cubes in this example). We note the higher values for `FaceGoal` and `BodyGoal` than used in Method 1 examples. This is natural because for Method 2, centers of boundary spheres are located on the faces, which results in higher relative area and volume occupied by the spheres in the unit brick. Optimal values for face and body goals can be chosen by the user experimentally, depending on a particular application.

Fig. 5 shows the unit cube with spheres centered on faces (compare with Fig. 2(a)) and the z-boundaries of the total volume $\mathcal{V}$, consisting of the same repeating copies of the unit cube face in the $x = y = 0$ plane.

Table 2 contains run times for computations analogous to those performed for Method 1 and listed in Table 1. Interestingly, for Method 2, unlike Method 1 (cf. Table 1), the computation times for the smaller sphere contact parameter 0.1 are not significantly different, (in fact, are smaller than) the computation times for the contact parameter value 0.2 (which remains true when the Weibull distribution of Example 1 is used in Method 2 instead of the Gamma distribution).

We have also performed runs of Method 2 with the same Weibull distribution (2.2), (2.4) as in Method 1 (resulting graphs are not shown). In this case, using Method 2, the computation times finish faster than those done with Method 1, and the

Table 2
Sample desktop computation times T numbers of spheres N in a unit cube for Example 2 with Gamma distribution (3.1), (3.3) , for the total domain $\mathcal{V}$ built of $2 \times 2 \times 1$ unit bricks of different side lengths, for two different values of the sphere contact parameter.

Cube side length	Sphere contact parameter	
	0.1	0.2
$15\,\bar{r}$	$T = 7$ s, $N = 160$	$T = 11$ s, $N = 178$
$30\,\bar{r}$	$T = 14$ min, $N = 1757$	$T = 15$ min, $N = 1771$

Table 3
Sample desktop computation times T numbers of spheres N in a unit cube for Example 2 with Weibull distribution (2.2), (2.4), for the total domain $\mathcal{V}$ built of $2 \times 2 \times 1$ unit cubes of different side lengths, for two different values of the sphere contact parameter (cf. Table 1 for Method 1).

Cube side length	Sphere contact parameter	
	0.1	0.2
$15\,\bar{r}$	$T = 4$ min, $N = 591$	$T = 4$ min, $N = 594$
$30\,\bar{r}$	$T = 32$ min, $N = 3332$	$T = 38$ min, $N = 3606$

computations also result in bigger numbers of spheres in the unit cube (compare Table 1 for Method 1 with Table 3 for Method 2).

4. Example 3: a more complex-shaped domain

The next example builds on the same framework as Methods 1 and 2 of sphere filling, extending to a non-parallelepiped-shaped domain. In this example, the domain $\mathcal{V}$ is similar to a single unit brick in Methods 1 and 2, with the subtraction of two hemispheres centered in the middles of two opposite faces corresponding to $x = 0$ and the maximal x. The radii of these spheres must be less or equal to 1/2 of the smallest of the brick dimensions.

A special sphere generating function `Example3GenerateSpheres.m` has been created for this example. The input and output parameters for this function are outlined below. The parameter set for `Example3GenerateSpheres.m` is smaller than that for the sphere generating functions in Methods 1 and 2, but the meaning remains the same (see Section 2.3).

Input parameters:

- `ProbabilityDistr`: a `Matlab` probability distribution of the radii of spherical particles. The average sphere radius is denoted by $\bar{r}$.
- `BrickSideLengths`: an array of three values corresponding to absolute lengths (in physical units) of the unit brick sides along x, y, and z.
- `HemisphereRadii`: an array containing two values corresponding to the radii (in physical units) of the two hemispheres centered in the middles of two opposite faces corresponding to $x = 0$ and the maximal x, defining the computation domain.
- `FaceGoal`: fraction of the domain boundary face area covered by projections of spheres in contact with it.
- `BodyGoal`: fraction of the domain volume filled by spheres.
- `SphereContactParameter`: the contact parameter, within $[0, 1]$. If two spherical particles are within `SphereContactParameter` $\times\ \bar{r}$ of each other, they are considered to be in contact.

Output parameters:

- `NSpheres`: The total number of random spheres placed into the domain $\mathcal{V}$.
- `Positions`: a matrix of coordinates of the random spheres (see Section 2.3).
- `Radii`: a vector storing the radii of all random spheres in the total domain $\mathcal{V}$ (see Section 2.3).
- `Contacts`: A matrix containing sphere pairs that are in contact (see Section 2.3).

Example 3A. In the first sample run, we use the Weibull distribution (2.2), (2.4) to fill a domain $\mathcal{V}$ based on a cube-shape with side length

$$\texttt{std_length} = 30\,\bar{r} \tag{4.1}$$

(for Weibull distribution, $\bar{r}$ is given by (2.3)), with the subtraction of two hemispheres of equal radii $0.5 \times$ `std_length`. A script `Example3A_Cube_Hemisph_Generate_and_Plot.m` (Appendix C) calls the sphere placing routine `Example3GenerateSpheres.m` with the following parameters.

- `ProbabilityDistr`: Weibull (2.2), (2.4).
- `BrickSideLengths: [1; 1; 1]*std_length`.
- `HemisphereRadii: [0.5; 0.5]*std_length`.
- `FaceGoal`: 0.4.
- `BodyGoal`: 0.4.
- `SphereContactParameter`: 0.1.

This example took 49 min to complete in the workstation configuration (138 min in the laptop configuration). The resulting spherical arrangement is shown in Fig. 6(a).

Example 3B. In the second sample run, the same Weibull distribution (2.2), (2.4) and the typical domain size (4.1) are used to fill a domain based on a non-cubical brick, with subtraction of two hemispheres of non-equal radii. A script based on `Example3A_Brick_Hemisph_Generate_and_Plot.m` of Example 3A calls the sphere placing routine with different domain and computation parameters, as follows.

- `ProbabilityDistr`: Weibull (2.2), (2.4).
- `BrickSideLengths: [1.3; 1; 1]*std_length`.
- `HemisphereRadii: [0.2; 0.4]*std_length`.
- `FaceGoal`: 0.4.
- `BodyGoal`: 0.4.
- `SphereContactParameter`: 0.2.

The resulting spherical filling and related graphs is shown in Fig. 6(b,c,d). This computation took 41 min to complete in the workstation configuration (144 min in the laptop configuration). The resulting sphere size distribution histogram (Fig. 6(d)) shows a good agreement with the given probability density function.

5. A physical example: bonding modeling in powder bed 3D printing process

We now use the sphere packing Method 2 to model heat-based bonding in the additive manufacturing process that uses a metal powder bed and a guided laser beam to heat the spherical particles and thus form bonds between them, as described in Ref. [8]. Particle size distribution for 316L stainless steel powder is approximated by the Weibull distribution (2.2), (2.4) [9,10].

5.1. The heat source and powder bed models

For the heat source model, physical assumptions are as follows.

- The heat flux from laser into the ith spherical particle is given by

  $$q_{i_{\rm laser}} = Q\,\frac{{r_i}^3}{{r_\ell}^3}\,,$$

  where Q is the total power of the laser, r_i is the radius of the particle, and r_ℓ is the radius of the laser beam.
- The heat flux from convection is

  $$q_{i_{\rm conv}} = k_b(T_R - T_i)\,,$$

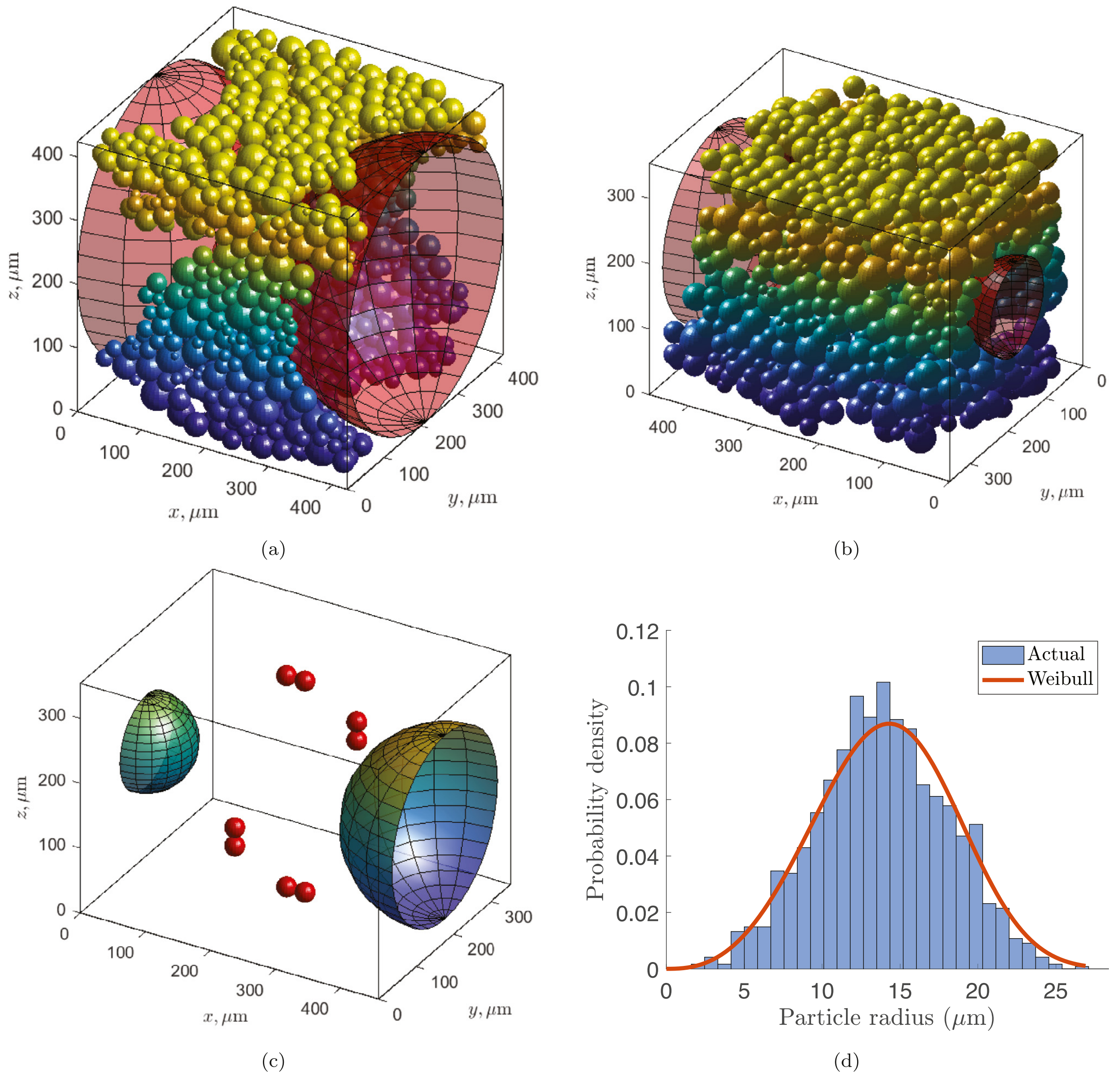

Fig. 6. (a) Example 3A: a unit cube with side length (4.1) minus two equal hemispheres, filled with random spheres and Weibull distribution (2.2), (2.4). (b) Example 3B: a similar computation with a non-cubic domain minus two non-equal hemispheres. (c) The computational domain and first parent sphere parents for Example 3B. (d) Actual sphere radius distribution in Example 3B, compared to the given Weibull distribution.

where k_b is the heat transfer coefficient, and T_R is the temperature of the surrounding air.

- The heat flux between two particles is given by

$$q_{ij} = k_t\left(T_j - T_i\right),$$

where k_t is the thermal conductivity, and T_i, T_j denote the temperatures of particles i and j. The total heat flux into the ith particle thus becomes

$$q_i = q_{i_{\text{laser}}} + q_{i_{\text{conv}}} + \sum_{j=1}^{N} q_{ij}.$$

- Using the discrete time stepping t_1, $t_2 = t_1 + \Delta t$, ..., the temperature update for the ith particle is expressed by

$$T_i^{t+\Delta t} = T_i^t + \frac{q_i^t}{m_i C_p}\Delta t,$$

where T_i^t is the particle's temperature at the previous time step, $T_i^{t+\Delta t}$ is the temperature at the following time step, q_i^t is the total initial energy flux, m_i is the mass of the ith spherical particle, and C_p denotes the material specific heat.
- If particles i and j are in contact, and both above the sintering temperature T_s, a bond is formed between them.

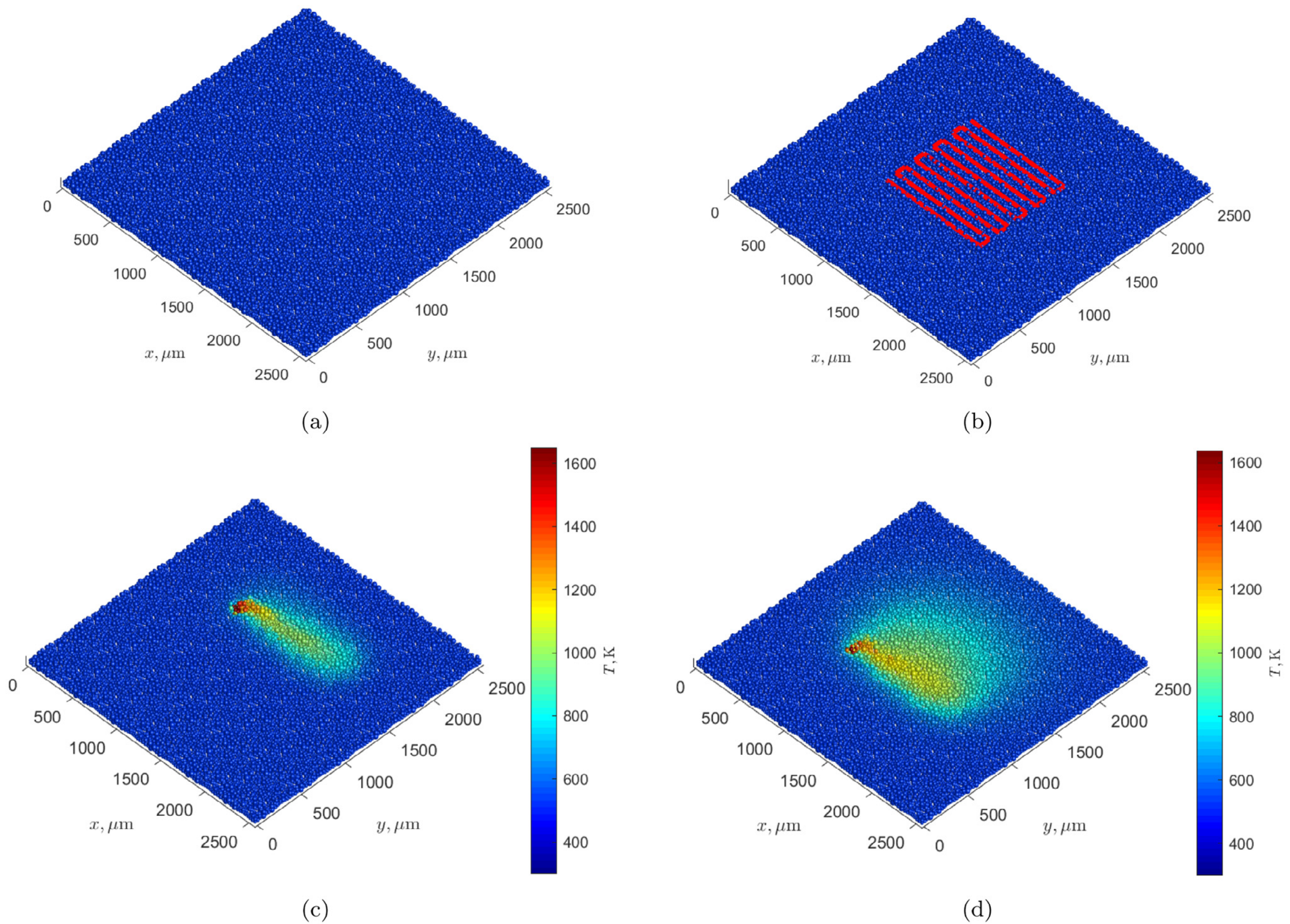

Fig. 7. A simulated print of a square (Section 5). (a) The printing domain (size in μm). (b) The laser beam path. (c) Second pass of the laser (temperature scale in Kelvins). (d) Final pass of the laser.

The following sample values for the simulated print using the parameters below were adapted from Ref. [9].

- Total power of the laser: $Q = 100$ W.
- Thermal conductivity of air: $\tau_a = 0.262$ W/(m · K).
- Thermal conductivity of steel: $\tau_s = 15$ W/(m · K).
- Heat transfer coefficient at the air boundary: $k_b = \pi \tau_a r_{\text{avg}}/2 = 2.9090 \times 10^{-6}$ W/K.
- Heat transfer coefficient of steel: $k_t = \pi \tau_s r_{\text{avg}}/2 = 3.3309 \times 10^{-4}$ W/K.
- Specific heat capacity of steel $C_p = 0.5$ J/(g · K).
- Steel density $\rho = 8 \times 10^{-12}$ g/$(\mu\text{m})^3$.
- Mass of a spherical particle $m_i = (4\pi/3)\,\rho\, r_i^3$.
- Laser radius $r_\ell = 50\,\mu$m.
- Ambient air temperature: $T_R = 300$ K.
- Sintering temperature: $T_s = 1000$ K.

5.2. A simulated print of a square

A sample powder bed was generated using Method 2 (Section 3) with parameters

- `ProbabilityDistr`: Weibull (2.2), (2.4),
- `FaceGoal`: 1.0,
- `BodyGoal`: 0.9,
- `SphereContactParameter`: 0.1,
- `ParentParameter`: 0.5,
- `BrickSideLengths`: `[1; 1; 0.12] * std_length`, where `std_length` $= 30\,\bar{r}$,
- `BrickNumbers`: `[6; 6; 1]`.

The computation took around five minutes, yielding the domain of size $\sim$ 2545 × 2545 × 50.89 μm, with the average particle radius 14.14 μm (Fig. 7(a)). A laser path shown in Fig. 7(b) was chosen to simulate the print of a small square in the middle of the domain by building thermally induced connections between the particles. Figs. 7(c,d) show the temperature maps on the second and the final laser pass. Fig. 8 depicts particles that were bonded in the result of the simulated print process and the top view of the bond graph.

The current software is useful for models and applications that employ a discrete approach where interactions between discrete particles, which can be modeled by spheres, become important. For such applications, a three-dimensional domain of interest needs to be filled with a large number of contacting non-overlapping spheres of sizes randomly drawn from a prescribed probability distribution. The current code employs geometrical principles to place initial spheres and then insert new spheres so that they are in contact with previous ones, in a random fashion. User-specified parameters include the domain dimensions,

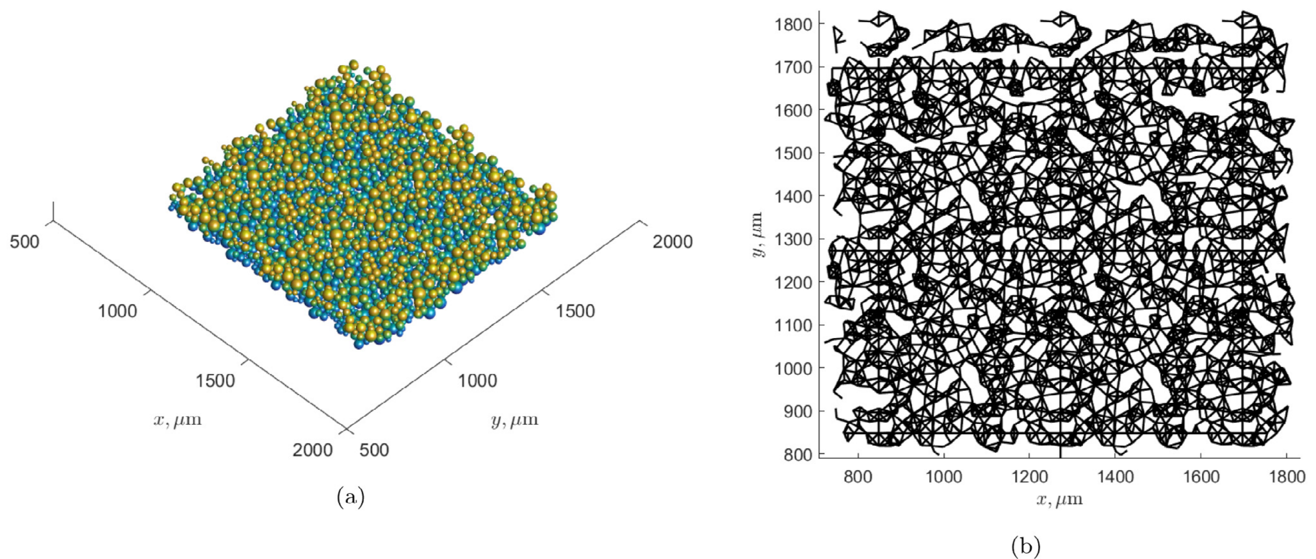

Fig. 8. The simulated print of a square (Section 5). (a) Particles bonded during the simulated print (b) Bonds formed between particles.

the desired probability distribution, and the desired surface and volume fractions of the spheres on the surface and in the bulk of the domain. The output includes sphere center locations, sphere radii, and the connectivity graph defined by pairs of spheres in contact with each other. The resulting sphere filling can be easily visualized using standard `Matlab` plotting capabilities. Examples are included.

6. Discussion and impact

The current manuscript describes a `Matlab` implementation of a geometry-based algorithm for filling a three-dimensional domain with spheres whose radii are randomly distributed according to a given probability distribution. Two methods, Method 1 and Method 2 (Section 2) fill a parallelepiped-shaped domain with spheres, using one or more copies of a standard "unit brick" formed with user-prescribed sizes. The two methods are implemented as freely interchangeable `Matlab` functions with identical input and output parameters. The methods differ in the placement principle and symmetry of spheres on the faces of the unit domain. Examples of using various sphere radius distributions and different domain shapes are included. The output of the presented code includes sphere center locations, sphere radii, and the connectivity graph defined by pairs of spheres in contact with each other. The resulting sphere filling can be easily visualized using standard `Matlab` plotting capabilities. An example in Section 4 illustrates the technique for spherical filling of a more complex domain shape involving hemispherical boundaries.

The presented software will be useful in models and applications that employ a discrete approach where interactions between a large number of randomly sized discrete particles, modeled by spheres, become important. Applications include the model of heat-induced connections between discrete metal particles in additive manufacturing (powder bed 3D printing) that served as the main motivation for this work (see Section 5). Other application areas that can directly benefit from the use of this software are various models of filtration, percolation and diffusion in granular media that arise, for example, in geological sciences, life sciences, filtration-related industrial applications, and waste management.

The code presented in this work can be efficiently adapted to generate spherical fillings of domains of different sizes and shapes, using any common particle size probability distribution from those available in `Matlab`.

The versatility of the presented algorithms in terms of domain shapes and sizes and particle size probability distributions make these algorithms rather different from those that generate random or non-random close packings of spheres (or other objects) of the *same size* (see, e.g., [6,7]). A common measure of optimality of the latter designs is the volume fraction ϕ occupied by the repeated object. It is well known that random close packings of identical spheres can attain $\phi \sim 0.64$, whereas the face-centered cubic lattice corresponds to $\phi \sim 0.74$ ([6] and references therein). The code presented in the current work, however, aims essentially at problems where spherical radii are unequal, following a given nonsingular (usually continuous) probability distribution that models a specific physical situation. The provided code therefore can be used in conjunction with a shooting method to estimate close-to-maximal volume fraction occupied by spherical particles in any given setup without the need to perform actual experiments. It can also be used to approximately solve the inverse problem of estimating the particle size distribution in situations where direct measurements are not feasible.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors are grateful to the anonymous referees for valuable suggestions and references, and to NSERC of Canada for support through the Discovery grant RGPIN-2019-05570 and a USRA fellowship.

Appendix A. Data generation and plotting script for sphere packing Method 1, Example 1 A

- Main file: `Example1A_Method1_Generate_and_Plot.m`
- Output:
  - Fig. 2(b,d,f).
  - Figure similar to Fig. 3(a,b) for Example 1B.
  - `Matlab` data file `Example1A_Method1_Results.mat`
- Required additional files (in the same folder):
  - `Method1GenerateSpheres.m`: the main Method 1 sphere filling routine
  - Auxiliary routine files for `Method1GenerateSpheres.m`:
    - `M1Position2Xmax.m`
    - `M1Position2Xmin.m`
    - `M1Position2Ymax.m`
    - `M1Position2Ymin.m`
    - `M1Position2Zmax.m`
    - `M1Position2Zmin.m`
    - `M1Position3.m`
    - `M1Search2D.m`
    - `M1Search3D.m`

The description of some commands and run parameters used in the script is as follows.

- `rng(0)`: Set `Matlab` pseudorandom seed to zero (default).
- `Weibull_scale`, `Weibull_shape`: the distribution parameters λ, k in (2.4).
- `ProbabilityDistribution`: the `Matlab` distribution variable for the PDF (2.2), (2.4).
- `cube_side_length`: side length of a unit cube; here 15 times the average sphere radius $\bar{r}$.
- `FaceGoal`, `BodyGoal`: minimal percentages of area and volume fill ratios of cube sides. Here 0.8 and 0.55.
- `BrickSideLengths = [1;1;1]* std_length` where `std_length` $= 15\,\bar{r}$.
- `BrickNumbers = [2;2;1]`: numbers of unit bricks in x, y, z directions, making up the total domain $\mathcal{V}$.
- `SphereContactParameter`, `ParentParameter`: the contact and the parent parameter (see Section 2.3). Here 0.2 and 0.5.

```
%% Matlab script for Example 1A: Method 1, using Weibull distribution
% Produces the sphere filling of the domain V and plots of Figure 2 (a, c, e) and figures similar to Fig.3 (a,b)
% Required files:
% * Method1GenerateSpheres.m -- main sphere filling routine
% * Auxiliary routine files for Method1GenerateSpheres.m :
%     - M1Position2Xmax.m
%     - M1Position2Xmin.m
%     - M1Position2Ymax.m
%     - M1Position2Ymin.m
%     - M1Position2Zmax.m
%     - M1Position2Zmin.m
%     - M1Position3.m
%     - M1Search2D.m
%     - M1Search3D.m

clear all; close all; clc;

%% Sphere Packing parameters: Weibull distribution

% -- use Weibull distribution implemented in Matlab
% with parameters from Steuben, Iliopoulos, Michopoulos (2016)

rng(0); %random seed zero: Matlab default
Weibull_scale = 31.4/2; % scale parameter lambda for the radius in Weibull distribution; in micrometers. The scale parameter
    for *diameter* is 31.4.
Weibull_shape = 3.55; % shape parameter k in Weibull distribution; dimensionless

ProbabilityDistribution = makedist('Weibull','a',Weibull_scale ,'b', Weibull_shape);

average_radius = mean(ProbabilityDistribution);
FaceGoal = 0.8;  % desired sphere/face area fraction; recommended for Method 1: 0.8
BodyGoal = 0.55; % desired sphere/volume fraction; recommended for Method 1: 0.55

%Unit brick: here unit cube
std_length = 15*average_radius;
BrickSideLengths = [1;1;1]* std_length;
```

```
%numbers of cubes in x, y, z directions to fill the total domain V
BrickNumbers = [2;2;1];

SphereContactParameter = 0.2; %This is the contact parameter, within [0,1]. If particles are within SphereContactParameter*
     average_radius of each other, they are considered to be in contact
ParentParameter = 0.5; %This is the parent parameter, within [0,1]. If particles are within ParentParameter*average_radius of
     each other, they are considered to be potential parents

%% Run the sphere generation Method 1 algorithm and save results.

tic;
[FinalNSpheres, UnitBrickNSpheres, Positions, Radii, Contacts, ListXmin, ListYmin, ListZmin, ListXmax, ListYmax, ListZmax] =
     ...
  Method1GenerateSpheres( ...
    ProbabilityDistribution, ...
    FaceGoal, BodyGoal, ...
    SphereContactParameter, ParentParameter, ...
    BrickSideLengths, BrickNumbers );

sphere_filling_total_time = toc

%Save computations
save('Example1A_Method1_Results.mat')

%% Now some plots: reproduce Fig.

[x_sph, y_sph, z_sph] = sphere;

%% Unit Cube Plot
figure(11)
hold on
axis equal
percent = 0;
List = 1:UnitBrickNSpheres; %particles you want to display
light              % create a light
lighting gouraud   % preferred method for lighting curved surfaces
for count = 1:size(List,2)
    i = List(count);
    surf(Radii(i)*x_sph + Positions(1,i), Radii(i)*y_sph + Positions(2,i), Radii(i)*z_sph + Positions(3,i),'EdgeColor','none',
         'FaceLighting','gouraud')

    if mod(count,1000)==0
        percent = 100*(count/UnitBrickNSpheres); %Keeping track of progress
        disp(['Plotting Unit Cube ',num2str(percent), '% complete'])
    end

end
xlim([0, BrickSideLengths(1)]);
ylim([0, BrickSideLengths(1)]);
zlim([0, BrickSideLengths(1)]);
xlabel('$x, {\rm \mu m}$', 'Interpreter', 'latex');
ylabel('$y, {\rm \mu m}$', 'Interpreter', 'latex');
zlabel('$z, {\rm \mu m}$', 'Interpreter', 'latex');
view(60, 45);
hold off;

% Save figure - uncomment as needed
% savefig('Example1A_Method1_UnitCube.fig');
% print('Example1A_Method1_UnitCube','-dpdf');

%% Unit Cube Middle Third
figure(13)
hold on
axis equal
List = find((Positions(3,1:UnitBrickNSpheres) > (1/3)*BrickSideLengths(1)) & (Positions(3,1:UnitBrickNSpheres) < (2/3)*
     BrickSideLengths(1))); %Selecting all spheres in middle third of cube
light              % create a light
lighting gouraud   % preferred method for lighting curved surfaces
for count = 1:size(List,2)
    i = List(count);
    surf(Radii(i)*x_sph + Positions(1,i), Radii(i)*y_sph + Positions(2,i), Radii(i)*z_sph + Positions(3,i),'EdgeColor','none',
         'FaceLighting','gouraud')

    if mod(count,1000)==0
        percent = 100*(count/size(List,2)); %Keeping track of progress
        disp(['Plotting middle third of Unit Cube ',num2str(percent), '% complete'])
    end
end
xlim([0, BrickSideLengths(1)]);
ylim([0, BrickSideLengths(1)]);
```

```
zlim([0, BrickSideLengths(1)]);
xlabel('$x, {\rm \mu m}$', 'Interpreter', 'latex');
ylabel('$y, {\rm \mu m}$', 'Interpreter', 'latex');
zlabel('$z, {\rm \mu m}$', 'Interpreter', 'latex');
view(60, 45);
hold off;

% Save figure - uncomment as needed
% savefig('Example1A_Method1_UnitCube_Middle.fig');
% print('Example1A_Method1_UnitCube_Middle','-dpdf');

%% Plot theoretical vs. actual distribution
FF=figure(14);
hold on
%plot histogram of obtained radii
histogram(Radii,30,'Normalization','pdf');

%plot Weibull PDF
Weibull_scale = 31.4/2; % scale parameter lambda in Weibull distribution; in micrometers
Weibull_shape = 3.55; % shape parameter k in Weibull distribution; dimensionless
ProbabilityDistribution = makedist('Weibull','a',Weibull_scale ,'b', Weibull_shape);
X = linspace(0,max(Radii)*1.1);
Y = pdf(ProbabilityDistribution,X);
plot(X,Y, 'LineWidth', 3);
set(gca, 'FontSize', 14)
xlabel('Particle radius ($\mu$m)','Interpreter','latex')
ylabel('Probability density','Interpreter','latex')
legend('Actual', 'Weibull', 'Interpreter','latex', 'location', 'northeast')
xlim([0, max(Radii)*1.1]);
hold off

% Save figure - uncomment as needed
% savefig('Example1A_Method1_UnitCube_DistributionCurves.fig');
% print('Example1A_Method1_UnitCube_DistributionCurves','-dpdf');

%% Two Cubes joined to make a 2x1 brick
figure(21)
hold on
axis equal

%1:Number will be the spheres in the first unit cube, N is the total number
%of spheres, all cubes combined.
List = 1:2*UnitBrickNSpheres;%
light               % create a light
lighting gouraud    % preferred method for lighting curved surfaces
for count = 1:size(List,2)
    i = List(count);
    surf(Radii(i)*x_sph + Positions(1,i), Radii(i)*y_sph + Positions(2,i), Radii(i)*z_sph + Positions(3,i),'EdgeColor','none',
        'FaceLighting','gouraud')

    if mod(count,1000)==0
        percent = 100*(count/(2*UnitBrickNSpheres)); %Keeping track of progress
        disp(['Plotting 2 Cubes Joined ',num2str(percent), '% complete'])
    end
end
xlim([0, 2*BrickSideLengths(1)])
ylim([0, BrickSideLengths(1)])
zlim([0, BrickSideLengths(1)])
xlabel('$x, {\rm \mu m}$', 'Interpreter', 'latex');
ylabel('$y, {\rm \mu m}$', 'Interpreter', 'latex');
zlabel('$z, {\rm \mu m}$', 'Interpreter', 'latex');

%Overhead View
view(0,90);
hold off;

% Save figure - uncomment as needed
% savefig('Example1A_Method1_Overhead.fig');
% print('Example1A_Method1_TwoCubes_Overhead','-dpdf');


%% Total Build (in this example, it is 2x2 unit cubes)
figure(23)
hold on
axis equal

List = 1:FinalNSpheres;
light               % create a light
lighting gouraud    % preferred method for lighting curved surfaces
for count = 1:size(List,2)
```

```
    i = List(count);
    surf(Radii(i)*x_sph + Positions(1,i), Radii(i)*y_sph + Positions(2,i), Radii(i)*z_sph + Positions(3,i),'EdgeColor','none',
        'FaceLighting','gouraud')

    if mod(count,1000)==0
        percent = 100*(count/FinalNSpheres); %Keeping track of progress
        disp(['Plotting total build ',num2str(percent), '% complete'])
    end
end
xlim([0, 2*BrickSideLengths(1)])
ylim([0, 2*BrickSideLengths(1)])
zlim([0, BrickSideLengths(1)])
xlabel('$x, {\rm \mu m}$', 'Interpreter', 'latex');
ylabel('$y, {\rm \mu m}$', 'Interpreter', 'latex');
zlabel('$z, {\rm \mu m}$', 'Interpreter', 'latex');
view(60, 30);
hold off;

% Save figure - uncomment as needed
% savefig('Example1A_Method1_TotalBuild.fig');
% print('Example1A_Method1_TotalBuild','-dpdf');
```

Appendix B. Data generation and plotting script for sphere packing Method 2, Example 2

- Main file: `Example2_Method2_Generate_and_Plot.m`
- Output:
  - Fig. 5.
  - Matlab data file `Example2_Method2_Results.mat`
- Required additional files (in the same folder):
  - `Method2GenerateSpheres.m`: the main Method 2 sphere filling routine
  - Auxiliary routine files for `Method2GenerateSpheres.m`:
    - `M2Position2Xmin.m`
    - `M2Position2Ymin.m`
    - `M2Position2Zmin.m`
    - `M2Position3.m`
    - `M2Search2D.m`
    - `M2Search3D.m`

```
%% Matlab script for Example 2: Method 2, using the Gamma distribution
% Produces the sphere filling of the domain and plots of Figure 5
% Required files:
% * Method2GenerateSpheres.m -- main sphere filling routine
% * Auxiliary routine files for Method1GenerateSpheres.m :
%     - M2Position2Xmin.m
%     - M2Position2Ymin.m
%     - M2Position2Zmin.m
%     - M2Position3.m
%     - M2Search2D.m
%     - M2Search3D.m

clear all; close all; clc;

%% Sphere Packing parameters: Gamma distribution

rng(0); %random seed zero: Matlab default

Gamma_scale = 7; % scale parameter theta in the Gamma distribution; dimensional (micrometers)
Gamma_shape = 2;  % shape (k) k in Gamma distribution; dimensionless

%first parameter a: shape; 2nd parameter b: scale
ProbabilityDistribution = makedist('Gamma','a', Gamma_shape ,'b', Gamma_scale);
average_radius = mean(ProbabilityDistribution) %equal to Gamma_shape*Gamma_scale

FaceGoal = 1.0;
BodyGoal = 0.9;

%Unit brick: here unit cube
std_length = 30*average_radius;
```

```
BrickSideLengths = [1;1;1] * std_length;

%numbers of cubes in x, y, z directions
BrickNumbers = [2;2;1];

SphereContactParameter = 0.1;
ParentParameter = 0.5;

%% Run the sphere generation Method 2 algorithm and save results

tic

[FinalNSpheres, UnitBrickNSpheres, Positions, Radii, Contacts, ListXmin, ListYmin, ListZmin, ListXmax, ListYmax, ListZmax] =
    ...
  Method2GenerateSpheres( ...
    ProbabilityDistribution, ...
    FaceGoal, BodyGoal, ...
    SphereContactParameter, ParentParameter, ...
    BrickSideLengths, BrickNumbers );

sphere_filling_total_time = toc

save('Example2_Method2_Gamma_Results.mat')

%% Method 2, plot desired vs. actual distribution: Fig.5c

FF=figure(14);
hold on;

%plot histogram of obtained radii
histogram(Radii,30,'Normalization','pdf');

%plot Weibull PDF
X = linspace(0,max(Radii));
Y = pdf(ProbabilityDistribution,X);
plot(X,Y, 'LineWidth', 3);
set(gca, 'FontSize', 14)
xlabel('Particle radius','Interpreter','latex')
ylabel('Probability density','Interpreter','latex')
legend('Actual', 'Gamma', 'Interpreter','latex', 'location', 'northeast')
hold off

% savefig('Example2_Method2_Gamma_DistributionCurves.fig');
% print('Example2_Method2_Gamma_DistributionCurves','-dpdf');

%% Example 2, Method 2, Unit Brick Plot - open (Fig. 5a)
max_radius=max(Radii);
minimal_sphere_x=-max_radius;
maximal_sphere_x=max(Positions(1,:))+max_radius;
minimal_sphere_y=-max_radius;
maximal_sphere_y=max(Positions(2,:))+max_radius;
minimal_sphere_z=-max_radius;
maximal_sphere_z=max(Positions(3,:))+max_radius;

figure(11)
hold on
axis equal
[x_sph, y_sph, z_sph] = sphere;
percent = 0;
List = 1:UnitBrickNSpheres; %particles to display
light              % create a light
lighting gouraud   % preferred method for lighting curved surfaces
for count = 1:size(List,2)
    i = List(count);
    surf(Radii(i)*x_sph + Positions(1,i), Radii(i)*y_sph + Positions(2,i), Radii(i)*z_sph + Positions(3,i),'EdgeColor','none',
        'FaceLighting','gouraud')

    if mod(count,1000)==0
        percent = 100*(count/UnitBrickNSpheres); %Keeping track of progress
        disp(['Plotting Open Unit Brick ',num2str(percent), '% complete'])
    end

end
xlim([0, BrickSideLengths(1)]);
ylim([0, BrickSideLengths(2)]);
zlim([0, BrickSideLengths(3)]);
xlabel('$x, {\rm \mu m}$', 'Interpreter', 'latex');
ylabel('$y, {\rm \mu m}$', 'Interpreter', 'latex');
zlabel('$z, {\rm \mu m}$', 'Interpreter', 'latex');
view(30, 30);
```

```
hold off;

% save as needed
% savefig('Example2_Method2_Gamma_UnitBrickOpen.fig');
% print('Example2_Method2_Gamma_UnitBrickOpen','-dpdf');

%% Example 2, Method 2, Unit Brick Plot - closed (Fig. 5b)
figure(12)
hold on
axis equal
[x_sph, y_sph, z_sph] = sphere;
percent = 0;
List = 1:UnitBrickNSpheres; %particles to display
light                % create a light
lighting gouraud     % preferred method for lighting curved surfaces
for count = 1:size(List,2)
    i = List(count);
    surf(Radii(i)*x_sph + Positions(1,i), Radii(i)*y_sph + Positions(2,i), Radii(i)*z_sph + Positions(3,i),'EdgeColor','none',
        'FaceLighting','gouraud')

    if mod(count,1000)==0
        percent = 100*(count/UnitBrickNSpheres); %Keeping track of progress
        disp(['Plotting Closed Unit Brick ',num2str(percent), '% complete'])
    end

end

max_sphere_radius=1.05*max(Radii);
xlim([-max_sphere_radius, BrickSideLengths(1) + max_sphere_radius]);
ylim([-max_sphere_radius, BrickSideLengths(2) + max_sphere_radius]);
zlim([-max_sphere_radius, BrickSideLengths(3) + max_sphere_radius]);
xlabel('$x, {\rm \mu m}$', 'Interpreter', 'latex');
ylabel('$y, {\rm \mu m}$', 'Interpreter', 'latex');
zlabel('$z, {\rm \mu m}$', 'Interpreter', 'latex');
view(30, 30);
hold off;

% save as needed
% savefig('Example2_Method2_Gamma_UnitBrickClosed.fig');
% print('Example2_Method2_Gamma_UnitBrickClosed','-dpdf');

%% Example 2, Method 2, Total domain - top and bottom faces  (Fig. 5d)

figure(31)
hold on
axis equal
List = ListZmin';
List2 = ListZmax';
light                % create a light
lighting gouraud     % preferred method for lighting curved surfaces
for count = 1:size(List,2)
    i = List(count);
    surf(Radii(i)*x_sph + Positions(1,i), Radii(i)*y_sph + Positions(2,i), Radii(i)*z_sph + Positions(3,i),'EdgeColor','none',
        'FaceLighting','gouraud')
end
xlim([minimal_sphere_x, maximal_sphere_x]);
ylim([minimal_sphere_y, maximal_sphere_y]);
zlim([minimal_sphere_z, maximal_sphere_z]);
xlabel('$x, {\rm \mu m}$', 'Interpreter', 'latex');
ylabel('$y, {\rm \mu m}$', 'Interpreter', 'latex');
zlabel('$z, {\rm \mu m}$', 'Interpreter', 'latex');

for count = 1:size(List2,2)
    i = List2(count);
    surf(Radii(i)*x_sph + Positions(1,i), Radii(i)*y_sph + Positions(2,i), Radii(i)*z_sph + Positions(3,i),'EdgeColor','none',
        'FaceLighting','gouraud')
end
xlim([minimal_sphere_x, maximal_sphere_x]);
ylim([minimal_sphere_y, maximal_sphere_y]);
zlim([minimal_sphere_z, maximal_sphere_z]);
xlabel('$x, {\rm \mu m}$', 'Interpreter', 'latex');
ylabel('$y, {\rm \mu m}$', 'Interpreter', 'latex');
zlabel('$z, {\rm \mu m}$', 'Interpreter', 'latex');
view([30, 30]);
hold off;

% save as needed
% savefig('Example2_Method2_Gamma_UnitCube_SidesZ.fig');
% print('Example2_Method2_Gamma_UnitCube_SidesZ','-dpdf');
```

Appendix C. Data generation and plotting script, Example 3

- Main file: `Example3A_Cube_Hemisph_Generate_and_Plot.m`
- Input and output: see Section 4, Example 3A for details.
- Required additional files (in the same folder):
  - `Example3GenerateSpheres.m`: the main sphere filling routine
  - Auxiliary routine files for `Example3GenerateSpheres.m`:
    * `EG3Position2Ymin.m`
    * `EG3Position2Ymax.m`
    * `EG3Position2Zmin.m`
    * `EG3Position2Zmax.m`
    * `EG3Position3.m`

```
%% Matlab script for Example 3: a brick (here cube) with partially spherical noundary

clear all; close all; clc;

% -- use Weibull distribution implemented in Matlab
% with parameters from Steuben, Iliopoulos, Michopoulos (2016)

rng(0); %random seed zero: Matlab default

Weibull_scale = 31.4/2; % scale parameter lambda for the radius in Weibull distribution; in micrometers. The scale parameter
    for *diameter* is 31.4.
Weibull_shape = 3.55; % shape parameter k in Weibull distribution; dimensionless

ProbabilityDistribution = makedist('Weibull','a',Weibull_scale ,'b', Weibull_shape);

average_radius = mean(ProbabilityDistribution);

FaceGoal = 0.4;
BodyGoal = 0.4;

% Unit brick: here a unit cube
std_length = 30*average_radius;
BrickSideLengths = [1;1;1] * std_length;

% Radii of hemispheres defining the romain, located on x_min and x_max faces
% Recommended: (x-length of the domain)/2 or smaller
HemisphereRadii = [0.5; 0.5] * std_length;

SphereContactParameter = 0.1;

tic;

[NSpheres, Positions, Radii, Contacts] = ...
  Example3GenerateSpheres( ...
    ProbabilityDistribution, ...
    BrickSideLengths, ...
    HemisphereRadii, ...
    FaceGoal, BodyGoal, ...
    SphereContactParameter ...
 );

sphere_filling_total_time = toc

%% Save all data
save('Example3A_Cube_Hemisph_data.mat')

%% Plot the given Weibull disribution vs. the actual distribution
figure(1)
hold on
X = linspace(0,max(Radii));
Y = wblpdf(X,Weibull_scale,Weibull_shape);
histogram(Radii,30,'Normalization','pdf')
plot(X,Y, 'LineWidth', 3);
set(gca, 'FontSize', 16)
xlabel('Diameter of particle $\mu m$','Interpreter','latex')
ylabel('Frequency of Diameter','Interpreter','latex')
legend('Actual Distribution', 'Desired Dirstibution')
set(gcf,'color','w');
```

```
% save figure: uncomment if needed
% print('Radii_Distribution_SphericalBoundary','-dpdf');
% savefig('Radii_Distribution_SphericalBoundary.fig');

%% Particle Placement Plot
figure(2)
hold on
axis equal
[x, y, z] = sphere;
List = 1:NSpheres;
light
lighting gouraud
for count = 1:size(List,2)
    i = List(count);
    surf(Radii(i)*x + Positions(1,i), Radii(i)*y + Positions(2,i), Radii(i)*z + Positions(3,i),'EdgeColor','none','
        FaceLighting','gouraud')
end
xlabel('$x, {\rm \mu m}$', 'Interpreter', 'latex');
ylabel('$y, {\rm \mu m}$', 'Interpreter', 'latex');
zlabel('$z, {\rm \mu m}$', 'Interpreter', 'latex');
set(gcf,'color','w');
view([45, 45])

% save figure: uncomment if needed
% print('ParticlePlacement_SphericalBoundary','-dpdf');
% savefig('ParticlePlacement_SphericalBoundary.fig');


%% First parents plot
x_max = BrickSideLengths(1); y_max = BrickSideLengths(2); z_max = BrickSideLengths(3);
x_min = 0; y_min = 0; z_min = 0;
x_max_input = BrickSideLengths(1); y_max_input = BrickSideLengths(2); z_max_input = BrickSideLengths(3);

figure(3)
hold on
axis equal
[x, y, z] = sphere;
List = 1:8; %These are the first parents

surf(HemisphereRadii(1)*x, HemisphereRadii(1)*y + y_max_input/2, HemisphereRadii(1)*z + z_max_input/2,'FaceAlpha',0.8); %First
    Spherical Boundary
surf(HemisphereRadii(2)*x + x_max_input, HemisphereRadii(2)*y + y_max_input/2, HemisphereRadii(2)*z + z_max_input/2,'FaceAlpha
    ',0.8); %Second Spherical Boundary
light
lighting gouraud
for count = 1:size(List,2)
    i = List(count);
    surf(Radii(i)*x + Positions(1,i), Radii(i)*y + Positions(2,i), Radii(i)*z + Positions(3,i),'EdgeColor','none','Facecolor',
        'r')
end
xlim([0 x_max])
ylim([0 y_max])
zlim([0 z_max])
plot3([x_min, x_max],[y_min, y_min],[z_min, z_min],'k')
plot3([x_min, x_max],[y_max, y_max],[z_min, z_min],'k')
plot3([x_min, x_min],[y_min, y_max],[z_min, z_min],'k')
plot3([x_max, x_max],[y_min, y_max],[z_min, z_min],'k')
plot3([x_min,x_min],[y_min,y_min],[z_min,z_max],'k')
plot3([x_max,x_max],[y_min,y_min],[z_min,z_max],'k')
plot3([x_max,x_max],[y_max,y_max],[z_min,z_max],'k')
plot3([x_min,x_min],[y_max,y_max],[z_min,z_max],'k')
plot3([x_min, x_max],[y_min, y_min],[z_max, z_max],'k')
plot3([x_min, x_max],[y_max, y_max],[z_max, z_max],'k')
plot3([x_min, x_min],[y_min, y_max],[z_max, z_max],'k')
plot3([x_max, x_max],[y_min, y_max],[z_max, z_max],'k')
xlabel('$x, {\rm \mu m}$', 'Interpreter', 'latex');
ylabel('$y, {\rm \mu m}$', 'Interpreter', 'latex');
zlabel('$z, {\rm \mu m}$', 'Interpreter', 'latex');
view([30, 30])

% save figure: uncomment if needed
% print('FirstParents_SphericalBoundary','-dpdf');
% savefig('FirstParents_SphericalBoundary.fig');

%% Plot the sphere filling and the domain boundaries
figure(4)
hold on
axis equal
[x, y, z] = sphere;
```

```
List = 1:NSpheres;
light
lighting gouraud
for count = 1:size(List,2)
    i = List(count);
    surf(Radii(i)*x + Positions(1,i), Radii(i)*y + Positions(2,i), Radii(i)*z + Positions(3,i),'EdgeColor','none')
end
surf(HemisphereRadii(1)*x, HemisphereRadii(1)*y + y_max_input/2, HemisphereRadii(1)*z + z_max_input/2,'FaceAlpha',0.5,'
     Facecolor','r'); %First Spherical Boundary
surf(HemisphereRadii(2)*x + x_max_input, HemisphereRadii(2)*y + y_max_input/2, HemisphereRadii(2)*z + z_max_input/2,'FaceAlpha
     ',0.5,'Facecolor','r'); %Second Spherical Boundary
xlim([0 x_max])
ylim([0 y_max])
zlim([0 z_max])
plot3([x_min, x_max],[y_min, y_min],[z_min, z_min],'k')
plot3([x_min, x_max],[y_max, y_max],[z_min, z_min],'k')
plot3([x_min, x_min],[y_min, y_max],[z_min, z_min],'k')
plot3([x_max, x_max],[y_min, y_max],[z_min, z_min],'k')
plot3([x_min,x_min],[y_min,y_min],[z_min,z_max],'k')
plot3([x_max,x_max],[y_min,y_min],[z_min,z_max],'k')
plot3([x_max,x_max],[y_max,y_max],[z_min,z_max],'k')
plot3([x_min,x_min],[y_max,y_max],[z_min,z_max],'k')
plot3([x_min, x_max],[y_min, y_min],[z_max, z_max],'k')
plot3([x_min, x_max],[y_max, y_max],[z_max, z_max],'k')
plot3([x_min, x_min],[y_min, y_max],[z_max, z_max],'k')
plot3([x_max, x_max],[y_min, y_max],[z_max, z_max],'k')
xlabel('$x, {\rm \mu m}$', 'Interpreter', 'latex');
ylabel('$y, {\rm \mu m}$', 'Interpreter', 'latex');
zlabel('$z, {\rm \mu m}$', 'Interpreter', 'latex');
view([30, 30])

% save figure: uncomment if needed
% print('ParticlesAndSphericalBoundaries_SphericalBoundary','-dpdf');
% savefig('ParticlesAndSphericalBoundaries_SphericalBoundary.fig');
```

Appendix D. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.softx.2022.101051.
It includes Matlab routines and scripts that correspond to the presented computational examples.

References

[1] Wang J. Packing of unequal spheres and automated radiosurgical treatment planning. J Comb Opt 1999;3(4):453–63.
[2] Michopoulos JG, Iliopoulos AP, Steuben JC, Birnbaum AJ, Lambrakos SG. On the multiphysics modeling challenges for metal additive manufacturing processes. Addit Manuf 2018;22:784–99.
[3] Xin H, Sun W, Fish J. Discrete element simulations of powder-bed sintering-based additive manufacturing. Int J Mech Sci 2018;149:373–92. http://dx.doi.org/10.1016/j.ijmecsci.2017.11.028.
[4] Hifi M, Yousef L. A local search-based method for sphere packing problems. European J Oper Res 2019;274(2):482–500. http://dx.doi.org/10.1016/j.ejor.2018.10.016.
[5] Stoyan Y, Scheithauer G, Yaskov G. Packing unequal spheres into various containers. Cybernetics and Systems 2016;52(3):419–26.
[6] Torquato S, Truskett TM, Debenedetti PG. Is random close packing of spheres well defined? Phys Rev Lett 2000;84(10):2064.
[7] Williams S, Philipse A. Random packings of spheres and spherocylinders simulated by mechanical contraction. Phys Rev E 2003;67(5):051301.
[8] Spierings AB, Levy G. Comparison of density of stainless steel 316l parts produced with selective laser melting using different powder grades. In: Proceedings of the annual international solid freeform fabrication symposium. Austin, TX; 2009, p. 342–53.
[9] Steuben JC, Iliopoulos AP, Michopoulos JG. Discrete element modeling of particle-based additive manufacturing processes. Comput Methods Appl Mech Engrg 2016;305:537–61.
[10] Spierings AB, Herres N, Levy G. Influence of the particle size distribution on surface quality and mechanical properties in AM steel parts. Rapid Prototyp J 2011.