Engineering with Computers (2021) 37:951–973
https://doi.org/10.1007/s00366-019-00866-2

ORIGINAL ARTICLE

Percentage porosity computation of three-dimensional non-convex porous geometries using the direct Monte Carlo simulation

Mauricio Campillo[1] · Pablo Pérez[1] · Jorge Daher[1] · Luis Pérez[1]

Received: 7 May 2019 / Accepted: 16 September 2019 / Published online: 25 September 2019

Abstract
The pursuit of more representative numerical models for open-cell metallic foams requires the computation of volume and percentage porosity of geometries containing randomly distributed interconnected pores, which is one of the main characteristics that determines its mechanical properties. From a mathematical standpoint, the analytical definition of foam geometries forms a three-dimensional non-convex set. It is known that the volume computation of *n*-dimensional polytopes and sets is a P-hard problem. A common way to approach this problem is using the Monte Carlo techniques; however, efforts are oriented toward the treatment of convex polytopes and polyhedrons. In this article, the Direct Monte Carlo Simulation (DMCS) is used to compute the percentage porosity of three-dimensional non-convex sets. A single-thread Python code was implemented, and tests were run to estimate the percentage porosity of three-dimensional open-cell porous geometries. Measurements of percentage porosity and runtime requirements over cubical and cylindrical geometries containing from 100 to 4000 overlapping spherical pores showed high accuracy and consistency in non-convex three-dimensional sets, while the proposed algorithm achieved a significant reduction in computing time with respect to the currently available method. In the same manner, results from the proposed algorithm were compared with a similar software available, showing a gain in both performance time and accuracy.

Keywords Direct Monte Carlo simulation · Percentage porosity · Non-convex sets · Metallic foams · Python

1 Introduction

Metallic foams have gained importance in recent years which can be verified by considerable number of investigations in the literature on this subject. They are composed of a metal-based structure with internal cavities called pores, similar to a conventional Metal Matrix Composite (MMC) but containing a void secondary phase. Percentage porosity is measured as the volume fraction of the void phase to the overall volume, which is the complementary percentage of the relative density of the solid phase. Porosity can be introduced in metal foams using a wide variety of fabrication methods, including processes using metals in solid, liquid, and gaseous states [1]. The main characteristic that determines the mechanical properties of metallic foams is the percentage porosity. This characteristic is mainly influenced by the random distribution of pores and the manufacturing process which establishes the internal structure of the foams. When pores exhibit a structure where a membrane bounds each one independently (e.g., honeycombs), it is said to be a closed-cell structure, while if interconnection exists between pores (e.g., sponges), it is said to be an open-cell structure [2].

The Finite-Element Analysis (FEA) has been a powerful and feasible numerical technique to model the mechanical behavior of both open-cell and closed-cell foams in the aid to identify plausible fabrication routes, depending on its desired microstructure and macroscopic mechanical properties. However, several challengers remain to be overcome to achieve better accuracy in the representation of its three-dimensional internal microstructure and macroscopic mechanical behavior, as its properties are highly dependent of the quality of the Computer-Aided Design (CAD) geometrical models [3]. FEA of heterogenous and porous media is commonly carried out by a Representative Volume Element (RVE) analysis [4], where the mechanical properties

✉ Mauricio Campillo
mauricio.campillo@postgrado.usm.cl

1 Department of Mechanical Engineering, Universidad Técnica Federico Santa María, Av. España 1680, Casilla 110-V, Valparaíso, Chile

of the medium are obtained by solving a Boundary-Value Problem (BVP) over a small and rather simple domain from which the equivalent mechanical properties are transferred to a more complex one assumed as a continuum [5–7].

One approach to improve the representativeness of the numerical results obtained in FEA-based models is by the inclusion of randomly distributed spherical pores, mimicking the natural distribution that can be observed in the microstructure of open-cell metallic foams obtained using the Space Holders Phase (SHP) technique, either by conventional powder metallurgy (PM) [8] or by infiltration of liquid metal [9]. These models must represent adequately the topological parameters of the metallic foams (e.g., pore distribution and sizes, interconnection, porosity percentage, etc.). To generate these metallic foam models, one of the most used mechanisms is using a CAD software package and executing a series of sequential command operations contained in a script [10–12], examples of these models can be seen in Fig. 1.

However, despite good results for the mechanical behavior of foams are reported in literature, the script commands' processing time is considerably excessive in relation to the FEA total time, which poses the necessity of developing an alternative to overcome this shortcoming. Figure 2 shows an example of the average time required to import a geometry into FEA preprocessor ANSYS v18 Design Modeler module and a quadratic relation with the number of cutting operations required to include pores.

In three dimensions, RVEs of porous media with open interconnected pores can be defined by the intersection of the interior of an external body (e.g., a cube or a cylinder) and the exterior of the union of spheres, which are allowed to intersect the exterior surface of the external body, to produce surface porosity seen in Fig. 1. Due to the overlapping pores and the manner in which the set is defined, both the geometry of the RVE and the union of the spheres of the void phase are non-convex sets. Hereafter, the concept of non-convex set may be referred to either of the prior.

In this article, an algorithm based on the Direct Monte Carlo Simulation (DMCS), implemented in Python language, is developed to estimate the percentage porosity of CAD generated metallic foam geometries. A grid cell-based linked list strategy and a Latin Hypercube Sampling (LHS) approach are used to achieve better computational efficiency and convergence [13]. A well-defined data set composed of 40 individual and distinct non-convex sets (i.e., 3D foam geometries) was used to evaluate the proposed method. This implementation has shown a time reduction between 80 and 90% with respect to the execution of the complete extruding and cutting operations by a CAD software package, as well as better accuracy and runtimes when compared to McVol [14], which is a Monte Carlo-based software intended to compute molecular volume of proteins, developed by Till et al.

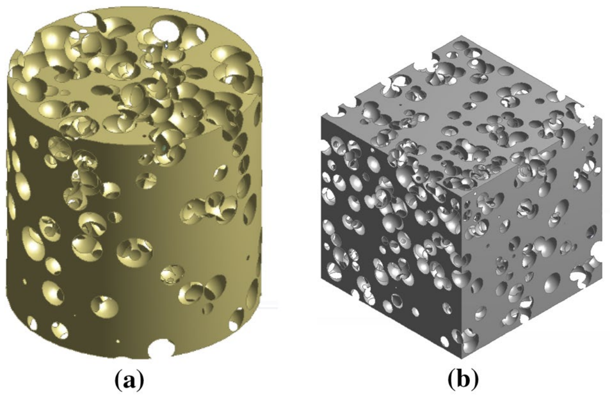


Fig. 1 3D foam geometries of solids containing spherical overlapping pores of 350–450 µm in diameter: **a** porous cylinder of diameter 2.25 mm and height 1.8 mm with 100 pores and 75.03% porosity and **b** porous cubes of size 3.6 mm with 1320 pores and 70.08% porosity

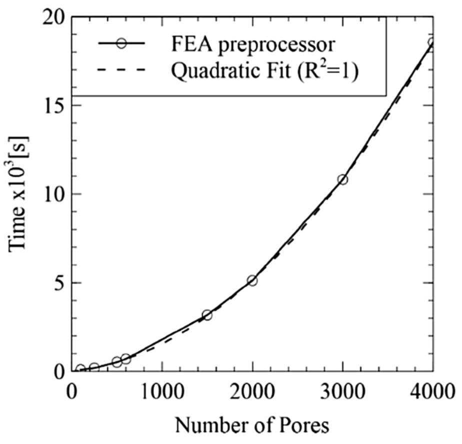


Fig. 2 Average loading time measured for cube-based foam geometries of size 3.6 mm into FEA Preprocessor Ansys v18 Design Modeler containing different number of spherical overlapping pores of 350–450 µm in diameter via JavaScript scripts execution

2 Methodology

2.1 CAD geometries' generation

The process to generate the CAD geometries of the foams is based in a basic three-step procedure, which is schematically shown in Fig. 3, for a two-dimensional case. First, a Discrete-Element Method (DEM) [15] simulation is carried out, from which the instantaneous element positions are used as the center position distribution of the pores (Fig. 3a). Then, using the defined pores center distribution, each pore diameter is redefined to meet the required criteria to fit the intender microstructure, such as pore interconnectivity and size distribution (Fig. 3b). Finally, a CAD geometry is completely defined as a script, which after being imported to an FEA Preprocessor or CAD package is seen as the remaining solid phase of the intersection between the interior of the external body and the exterior of the union of spheres, as shown in Fig. 3c.

Two sets of 3D CAD geometries containing randomly distributed spherical pores have been generated using DEM, where one set corresponds to cylinder-based geometries, while the second set is cube-based, similar to those shown in Fig. 1. These two general shapes have been selected due to their relevance on both numerical and experimental studies of mechanical behavior of metallic foams. While experimental tests use cylindrical specimens for compressive testing [16, 17], numerical simulations are oriented toward the characterization of mechanical behavior based on a reduced RVE in association to a multiscale scheme [18, 19], and hence, the cubic-shaped geometry is more suited for orthogonal load testing [20–22].

Originally presented to model the behavior of granular media, the DEM is a powerful numerical tool to model the random distribution of pores generated in foam fabrication methods such as SHP due to the solid mixing process [17] or in liquid metal infiltration process [23, 24]. The DEM represents the medium as a collection of material particles exhibiting independent rigid body motion behavior where the total external force acting over each particle is determined by the interaction with neighboring particles in which each pair interaction force is governed by a low-range contact force law [15]. In this work, a Hertz-type law with dampening has been considered for this purpose, where the total force has a normal component and a tangential component contributing to the linear and angular momentum Newton's conservation law correspondently. This type of contact law is found to be usual on the simulation of granular media [25–27].

Geometries based on cylinders of 6.75 mm in diameter (D) and 5.4 mm in height (H), and cubes of size (A) 6.75 mm containing 100, 500, 1000, 2000, and 4000 pores with diameters (d) ranging from 350 to 450 μm have been created using DEM open software LIGGGHTS [28]. An H/D ratio of 0.8 has been chosen to avoid flexural deflection and inelastic buckling, in accordance with geometric recommendations for short specimens in ASTM E9-09 standard [29]. A ratio of characteristic length to maximum pore size of 15 (i.e., D/d for cylinders and A/d for cubes) was chosen, to allow the inclusion of a wide range of number of pores throughout the tests. The maximum number of pores was set to 4000, as loading times into the FEA preprocessor becomes excessive.

Cylinder-based geometries have been set to have its revolution axis coaxial with the Z-axis, while the base is contained in the X–Y plane protruding through the $Z > 0$ subspace. On the other hand, cube-shaped geometries were oriented to have three of its faces aligned with the X–Y, Y–Z, and Z–X planes, while its remaining three faces contained in the $X, Y, Z > 0$ subspace.

DEM special domain used considered 1 time the maximum pore size (450 μm) for each side, being the radius, lower and upper height limits for cylinders and for each lower and upper direction for cubes, to promote the presence of pores on the surface of the resulting geometry and particle size was set to 0.25 times the maximum pore size (112 μm) to achieve interconnection between pores on the generated CAD geometries.

For both geometry types, the conditions for particles insertion were initial velocity (0.5 mm/s on each direction), high Young's modulus (5.0e8 MPa), and coefficient of restitution (0.95) and near zero gravity with the intention of generating high level of interaction between particles and maintaining high total kinetic energy in the system during simulation, according to Pérez et al. [9].

Results from DEM simulations were post-processed using a purposely developed Python script to generate ANSYS v18 Design Modeler input JavaScript scripts, using the corresponding center of the particles and assigning a uniformly distributed random value for each pore diameter between 350 and 450 μm, similar to Ref. [9]. The aforementioned script follows the following pseudo-code structure.

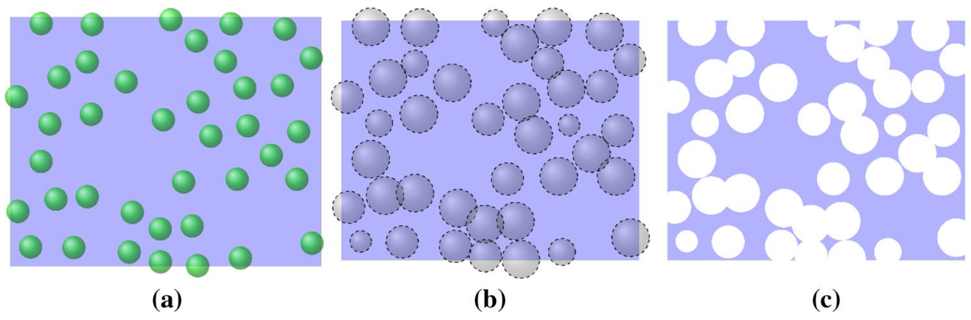


Fig. 3 Two-dimensional representation of the RVE generation of porous media with open-cell porosity: **a** Pores' center distribution obtained by DEM; **b** Uniform random definition of pores diameter; and **c** resulting foam geometry

```
#Define Type of Geometry
(Geo.Type) = {CUBE, CYLINDER}
#Define Characteristic Lengths
Switch Case (Geo.Type):
    CUBE: (Geo.charlength) = [(A)]
    CYLINDER: (Geo.charlength) = [(H), (D)]
#Define Pore Minimum and Maximum Diameters
(pore_diam) = [(min_diam), (max_diam)]
#Retrieve List of Centers
input_file.open()
for each line in (input_file):
    Append info to (X_vector) and (Y_vector) and (Z_vector)
input_file.close()
#Build JavaScript file
output_file.open()
#Set Global Dimensions
    Switch Case (Geo.Type):
        CUBE:  output_file.write((A))
        CYLINDER: output_file.write((H), (D))
#Write Instructions for Basic Extrusion
Switch Case (Geo.Type):
    CUBE:
        output_file.write(Build Square of Edge (A))
        output_file.write(Extrude Square (A) Units)
    CYLINDER:
        output_file.write(Build Circle of Diameter (D))
        output_file.write(Extrude Circle (H) Units)
#Write instructions for Pores as Cutting Operations
for i in range(list_of_pores):
    #Define Pore Center and Diameter
    (diameter) = random.uniform((pore_diam.min),(pore_diam.max))
    (center)=(X_vector[i],Y_vector[i],Z_vector[i])
    #Write Instructions
    output_file.write(Create Sphere Cut with (center), (diameter))
output_file.close()
```

2.2 The Monte Carlo simulation

Volume computation of polytopes is a problem which has been studied for at least 3 decades [30–32], and since then, author has been using different approaches to provide polynomial solutions to this problem, at least for low-dimensional cases. One of the most common approaches is based on the principle of divide-and-conquer, and the polyhedron is divided into smaller and simpler instances where the volume can be computed after by addition. Algorithms such as volume decomposition [33] and triangulation [34] are examples of this approach. Another commonly used approach is based on the approximation of volume by means of the Monte Carlo (MC) simulation. Here, various algorithm implementations have been developed for the volume estimation of convex n-dimensional polytopes for which validation is restricted to low-dimension geometries with known volume, whether it is analytically or not [35, 36]. These algorithms are typically based under a ‘hit-and-run’ technique and random walks used for a uniform sampling throughout the geometry surface of the n-dimensional volume, providing a polynomial time solution for the treatment of convex polytopes [37]. It is worth noticing that a general non-convex set can be approximated by a non-convex polytope and the approximating non-convex polytope can be rewritten as a union of convex and simpler polytopes. Hence, in principle, the volume of non-convex sets can be computed using algorithms and techniques intended for the treatment of convex polytopes, but the convex representation of non-convex polytopes can be extremely expensive and algorithms efficiency can be severely affected by the number of instances needed to represent the original set [38].

Although spheres from which pores are generated correspond to convex three-dimensional bodies, the union of two or more pores, which is called a macro-pore, is not guaranteed to hold this property [39]. In the case of the CAD geometries treated in this work, the existing overlap between spheres in relation to the sphere diameters is mainly unknown, which supposes that the generated volume is highly likely to form a non-convex set. Due to this non-convexity, the application of the previously described algorithms is restricted and a different technique, such as simple quadrature, may be applicable [40]. This problem has been tackled in fields such as biochemistry, where the volume of proteins, represented as a collection of overlapping spheres, can be computed using algorithms based on volume decomposition [41] or the MC simulation [14].

2.2.1 Direct Monte Carlo simulation (DMCS)

In this work, the MC simulation is used as a mechanism to gather information about an analytic 3D CAD geometry. In this approach, a set of N independent Bernoulli random variables, $\boldsymbol{X}_j^N$, are evaluated, each one according to a probability distribution function such as:

$$f(\boldsymbol{X}) = \begin{cases} 1 & X \in \text{solid} \\ 0 & \text{Elsewise} \end{cases}, \tag{1}$$

where each $\boldsymbol{X}_j^N$ corresponds to an independent three-dimensional vector in the subspaces enclosed by the exterior surface of the geometry. As all the information relative to the pores is known a priori (being its centers and radii), an alternative which poses an equivalent but less expensive to solve problem is the one on estimating the void volume inside the geometry and then to compute the solid portion as the difference between the solid volume and the estimated void volume. In this manner, the probability distribution function proposed in Eq. 1, can be rewritten as:

$$f(\boldsymbol{X}) = \begin{cases} 1 & X \in \text{a void} \\ 0 & X \notin \text{any void} \end{cases}. \tag{2}$$

With this, for each Bernoulli random variable, if the vector $\boldsymbol{X}_j^N$ happens to reside inside a pore, it considers a 'hit', while if it does not reside inside any void, it is considered a 'miss'. For any given number N of Bernoulli random variables, it is known that they follow a Binomial distribution, where its estimation, λ_N, of the expected value, λ, is determined as the product between the probability of success (or 'hit') and the total number of independent variables, N. As the expected value of any given distribution is known from the evaluation of every variable, the probability of 'hit' on any Binomial distribution, p_N, may be determined as a function of the number of independent variables as:

$$p_N = \frac{\lambda_N}{N}. \tag{3}$$

With this, as the number of independent Bernoulli variables increases, the estimator, λ_N, tends to the real expected value, λ, of the Binomial distribution [42] and, as a consequence, the real probability of success, p, is found. This is:

$$\lim_{N\to\infty} p_N = \lim_{N\to\infty} \frac{\lambda_N}{N} = p. \tag{4}$$

In terms of the DMCS, each Bernoulli random variable evaluation of the binary probability function in Eq. 2 is named a *membership oracle call*, since there is an algorithm established a priori to determine whether the returned for an arbitrary input value is 0 or 1. Later, the wanted percentage porosity of a CAD geometry foam is associated with the probability of success of the related Binomial distribution followed by the random points submitted to the membership oracle.

2.2.2 Random point generation

A Latin Hypercube Sampling (LHS) [34] strategy was used to generate the sampling points to be submitted to the membership oracle. To implement the LHS, each dimension of the domain is stratified in m equiprobable strata. Later, each stratum is randomly sampled one time. For cubes, each dimension has a span from 0 to A; therefore, each stratum has equal length $l = A/m$. Whether for cylinders, to achieve the equiprobability condition in cylindrical coordinates, it is required that each stratum has a different length in the radial direction (R), so the volume of each stratum can remain constant, and hence, the jth radial stratum will be of length l_j^R:

$$l_j^R = \sqrt{\frac{(j-1)}{k}}\left(\frac{D}{2}\right). \tag{5}$$

Later, each stratum, $\boldsymbol{S}_{i,j,k}$, is defined as:

$$\boldsymbol{S}_{i,j,k}^{CUBE} = \begin{bmatrix} il, (i+1)l \\ jl, (j+1)l \\ kl, (l+1)l \end{bmatrix} | \{i,j,k\} \in [0, 1, \dots, (m-1)], \tag{6}$$

whether for cylinders, each stratum is defined by:

$$\boldsymbol{S}_{i,j,k}^{CYL} = \begin{bmatrix} \sqrt{i}l^R, \sqrt{i+1}l^R \\ jl^\theta, (j+1)l^\theta \\ kl^z, (k+1)l^z \end{bmatrix} | \{i,j,k\} \in [0, 1, \dots, (m-1)], \tag{7}$$

where the lengths of the strata in the other directions $l_\theta = 2\pi/m$ and $l_z = H/m$ are constant. Although, for arbitrary parallelepipeds, individual lengths, $l_i = A_i/m$ shall be used for the ith dimension in Eq. 6, the proposed implementation does not feature this option.

Finally, for each stratum, $\boldsymbol{S}_n^N = \boldsymbol{S}_{i,j,k}$, a uniform random point ($\boldsymbol{X}_n^N$) is generated where:

$$\boldsymbol{X}_n^N = \boldsymbol{X}_{i,j,k} = \left\{ \boldsymbol{X} \in \boldsymbol{S}_{i,j,k} \subset \mathbb{R}^3 \forall i,j,k \in \{0, 1, \dots, (m-1)\} \right\}, \tag{8}$$

where $n = \left[0, \left(m^3 - 1\right)\right]$ or in terms of the total number of Bernoulli random variables, $n = [0, (N-1)]$.

2.2.3 Porosity estimation

The expected value, λ_k, for an arbitrary number, k, of random measurements of the random variable X distributed

Binomial with parameters n and p is known to tend to the expected value of the Binomial distribution, μ, this is:

$$\lambda\left(\bar{X}\right) = \sum_{i=1}^{k} \frac{1}{k}\mu_i = n\left(\frac{\mu}{n}\right) = \mu.|k \rightarrow \infty\,. \tag{9}$$

The prior is based on the premise that the random variable X, which represents the number of successes in any random sampling of size n, is distributed Normal with mean $\mu = np$ and variance $\sigma^2 = np(1-p)$. This is:

$$X = N\left(\mu, \sigma^2\right) = N(np, np(1-p)). \tag{10}$$

A standardization of the later Normal distribution results in a new random variable Z also distributed Normal defined by:

$$Z = \frac{X_i - n_i p}{n_i p(1-p)} = \frac{\frac{X_i}{n_i} - p}{p(1-p)}. \tag{11}$$

The above random variable Z, measured by X_i/n_i associated with the previously defined Binomial distribution, is then distributed Normal with mean value p and variance $p(1-p)$. As mean value of a collection of k random samples $\bar{Z}$ is known to be

$$E\left(\bar{Z}\right) = \mu(z) = \frac{1}{k}\sum_{i}^{k} Z_i = 0|k \rightarrow \infty, \tag{12}$$

it is necessary that the summation term in Eq. 10 satisfies:

$$\frac{1}{k}\sum_{i=1}^{k}\left(\frac{X_i}{n_i} - p\right) = 0|k \rightarrow \infty. \tag{13}$$

From Eq. 11, given that

$$\frac{1}{k}\sum_{i=1}^{k} p = \frac{1}{k}(kp) = p, |k \rightarrow \infty\,, \tag{14}$$

it follows consequently that

$$p = \frac{1}{k}\sum_{i=1}^{k}\frac{X_i}{n_i} = \frac{1}{k}\sum_{i=1}^{k} p_i = \bar{p}|k \rightarrow \infty. \tag{15}$$

Hence, for a random sample of measurements of the probability of success, $p_i = X_i/n_i$, of the Binomial distribution done by arbitrary samples over the distribution, it is guaranteed that its mean value, $\bar{p}$, represents an estimator of the unknown probability of success, p [43].

2.2.4 Convergence

As it is known that the sample mean, $\bar{p}$, in general will not coincide with the random distribution mean, p, it is a common practice to evaluate the accuracy of the results from MC simulations based on the distribution of the independently computed sample mean. Therefore, rather than computing the exact value of the distribution, p, a confidence interval is established to limit the error on the estimator of the mean, $\bar{p}$ [44, 45]. For normally distributed variables with unknown standard deviation and small sample sizes, n, it is well known that a confident interval for the mean, with a confidence level of $(1-\alpha)$, can be established based on the estimator of both the mean, $\bar{p}_n$, and the standard deviation, S_n, or rather the standard error, $S_n/\sqrt{n}$, by:

$$\Pr\left[\bar{p}_n - t_{n-1}\left(1-\frac{\alpha}{2}\right)\frac{S_n}{\sqrt{n}} \le p \le \bar{p}_n + t_{n-1}\left(1-\frac{\alpha}{2}\right)\frac{S_n}{\sqrt{n}}\right] = 1-\alpha, \tag{16}$$

where t_{n-1} is the Student's distribution of $(n-1)$ degrees of freedom. Furthermore, when the sample size is large enough (e.g., $n \ge 30$), the Student's distribution can be approximated by the Standard's Normal distribution (Z). Hence, Eq. 16 can be restated as:

$$\Pr\left[\bar{p} - Z\left(1-\frac{\alpha}{2}\right)\frac{S_n}{\sqrt{n}} \le p \le \bar{p} + Z\left(1-\frac{\alpha}{2}\right)\frac{S_n}{\sqrt{n}}\right] = 1-\alpha. \tag{17}$$

When the standard normal percentile $Z(1-\alpha/2) = 3$, Eq. 17 takes the form of the well-known three-sigma rule, which established a confidence interval with a confidence level of 99.97%. Hence, for any sample size, an uncertainty level, ε, can be established, although its confidence level will increase as the sample size increases. With this, a stopping rule for the sampling algorithm can be established based on the sample size and its standard deviation as:

$$a\frac{S_n}{\sqrt{n}} \le \varepsilon \Rightarrow \frac{S_n}{\sqrt{n}} \le \frac{\varepsilon}{a}, \tag{18}$$

where the parameter a is the number of standard deviations allowed in the uncertainty and the corresponding confident level is related to the confidence level of either the student or the standard normal distribution percentile as:

$$a = \begin{cases} t_{n-1}\left(1-\frac{\alpha}{2}\right) & n < 30 \\ Z\left(1-\frac{\alpha}{2}\right) & n \ge 30. \end{cases} \tag{19}$$

In this article, the three-sigma rule and a minimum sample size of 4 MC experiments have been adopted for the stopping rule of the MC simulations, which establishes a minimum confidence level of 94%. This is:

$$\Pr\left[\bar{p}_n - 3\frac{S_n}{\sqrt{n}} \le p \le \bar{p}_n + 3\frac{S_n}{\sqrt{n}}\right] \ge 0.939|n \ge 4. \tag{20}$$

2.2.5 Performance

For each MC experiment, N calls to the membership oracle are needed. In addition, each call to the oracle is resolved in a maximum of m independent operations, given by the evaluation of the distance between the random sample point and the center of the pores contained in all the neighbor cells, defined by the linked lists; this means that, in the worst-case scenario, the membership oracle will evaluate the pores contained in a maximum of 27 cells (i.e., the central cell and all the adjacent cells that shares one face or vertex with it) as every other pore is discarded since they are too far from the point so that it is possible for it to be inside them. With this, if each independent operation made by the oracle is considered equal to one FLOP, it is expected for the algorithm to perform a complete experiment in a total amount of FLOP with an upper bound given by:

$$FLOP^{N}_{total} = N \cdot m. \tag{21}$$

Hence, the total FLOPs required to perform an arbitrary MC experiment for any given geometry are determined by the number of pores contained in the geometry and the number of calls made to the oracle. If an iterative process is considered, where a total of k MC independent experiments are performed, the total FLOP is determined by the summation of the times required to execute each individual MC experiment. Assuming a process where each MC experiment is preformed using N calls to the oracle, it is expected for the algorithm to require a total number of computing operations of:

$$FLOP_{total} = k \cdot N \cdot m. \tag{22}$$

From Eq. 22, the implemented algorithm is expected to exhibit a time performance linear with both the number of pores, given by the expected number of pores, m, contained in the neighboring cell space and the base number of calls to the oracle, N. In this work, the implemented algorithm considers a constant sampling size, similar to the test procedure presented by Liu et al. [36], although an LHS strategy was adopted to improve sampling efficiency. A total of 50 strata per dimension, which translates to a sample size of 125 thousand points per MC experiment, were used. Using the LHS strategy and linked lists to identify neighboring pores, it is expected to achieve faster convergence than uniform random sampling as generated samples are non-collapsing in space and linked lists allow to dismiss verification of very distant pores and, therefore, unnecessarily to check.

2.3 Implementation

An algorithm to compute porosity percentage based on the DMCS introduced in the previous section was implemented. A complete version of the Python code can be found in "Appendix A". For the execution of the code, the usage of Python libraries numpy for scientific computing and numba, through its jit decorator, for computational optimization, along with the built-in modules math, random, and sys for file handling and arithmetic operations was required. When a geometry is under analysis, independent MC experiments are performed using a given number of samples size, N, from which the success probability, p_i, is estimated. MC experiments are performed until the standard error, $S_n/\sqrt{n}$, of the estimated success probability, $\bar{p}$, given by the samples mean, reaches a prescribed error, $\varepsilon/3$.

2.3.1 Function for retrieving geometry information contained in FEA preprocessor script

The first step in the code execution consists in gathering the information regarding the geometry under analysis which is retrieved from the corresponding input JavaScript scripts for ANSYS preprocessor. This is achieved by the sequential reading of the script file, from which general dimensions of the geometry (i.e., diameter, D, and height, H, for cylinders and the side, A, for cubes) as well as the complete lists for the pores (i.e., center position in directions x, y, z and radii). Once all the relevant information is retrieved, the file is closed and dismissed.

```python
def retrieve_file_info(inp_file):
# File is read line by line
    with open(inp_file) as ifile:
        n_void = 0
        radii, x, y, z = [], [], [], []
        for line in ifile:
            if 'var H=' in line:                                            # Cylinder height
                l_H = float(line.strip("var H=").strip(";\n"))
            if 'var D=' in line:                                            # Cylinder diameter
                l_D = float(line.strip("var D=").strip(";\n"))
            if 'var A=' in line:                                            # Cube edge size
                l_A = float(line.strip("var A=").strip(";\n"))
            if ('radio=0;' not in line and 'radio=' in line):           # Sphere radius
                n_void += 1
                radii.append(float(line.strip("radio=").strip("; \n")))
            if ('posicionx=0;' not in line and "posicionx=" in line):   # Sphere center X-coordinate
                x.append(float(line.strip("posicionx=").strip("; \n")))
            if ('posiciony=0;' not in line and "posiciony=" in line):   # Sphere center Y-coordinate
                y.append(float(line.strip("posiciony=").strip("; \n")))
            if ('posicionz=0;' not in line and "posicionz=" in line):   # Sphere center Z-coordinate
                z.append(float(line.strip("posicionz=").strip("; \n")))
        ifile.close()
        # Checks retrieved type of geometry and sets zero the unused variables
        if not 'l_A' in locals():                                           # Checks if is not a Cube
            l_A = 0
        elif not ('l_D' in locals() and 'l_H' in locals()):             # Checks if is not a Cylinder
            l_H, l_D = 0, 0
        else:                                                               # Acts if does not detect neither a cube nor a cylinder
            l_A , l_D, l_H = 0, 0, 0
            print('Error retrieving file info. Not a Cylinder nor a Cube was found.')
            sys.exit(2)
    return l_H,l_D,l_A,np.asarray(x),np.asarray(y),np.asarray(z),np.asarray(radii),n_void
```

2.3.2 Function to build the linked lists

The process of building the linked lists was capsuled in a function called build_lists. This function takes as input the information of the pores (i.e., pores' center and radii) and returns as output the linked lists head and list and two arrays containing the values of the corresponding grid lengths and the point from which the grid is deployed, respectively. First, temporal arrays are built to capture the extreme most coordinates that the spheres will have in the *n*-dimensional space (three-dimensional for this application). This is achieved by adding and subtracting the corresponding radii to each sphere center and selecting the minimum and maximum values (lines 3, 4, and 5). Then, the corresponding number of cells in each dimension is defined (line 8) by dividing the difference between the two extreme values by the biggest sphere diameter (line 6), and by the *math.ceil()* function, the closest upper integer is selected. With this, it is clear that no pore residing more than one cell away can be considered for evaluation, later by the membership oracle, as it will be away from the random point by, at least, two times the maximum radius. Later, the corresponding grid lengths are computed by dividing the corresponding dimensional span by the recently defined number of cells (line 9). Finally, for each sphere in the array, its corresponding *cell* value is computed and stored in the linked lists by filling the corresponding cell with its index in the *head* list and pushing the already stored information into the *list* list under the corresponding index.

```python
def build_lists(x_vec, y_vec, z_vec, r_vec):
    # Temp Lists
    x_ext = [np.amin(x_vec - r_vec), np.amax(x_vec + r_vec)]
    y_ext = [np.amin(y_vec - r_vec), np.amax(y_vec + r_vec)]
    z_ext = [np.amin(y_vec - r_vec), np.amax(z_vec + r_vec)]
    r_max = max(r_vec)
    # Cells definition
    C = [math.ceil((x_ext[1] - x_ext[0])/(2*r_max)), math.ceil((y_ext[1] - y_ext[0])/(2*r_max)), math.ceil((z_ext[1] - z_ext[0])/(2*r_max))]
    diff = [(x_ext[1] - x_ext[0])/C[0], (y_ext[1] - y_ext[0])/C[1], (z_ext[1] - z_ext[0])/C[2]]
    # Arrays Allocation
    list = np.zeros(len(r_vec))
    head = np.zeros((C[0],C[1],C[2]))
    # Lists Filling
    for i in range(len(r_vec)):
        CELL = [math.floor((x_vec[i]-x_ext[0])/diff[0]), math.floor((y_vec[i]-y_ext[0])/diff[1]), math.floor((z_vec[i]-z_ext[0])/diff[2])]
        list[i] =  int(head[CELL[0],CELL[1],CELL[2]])
        head[CELL[0],CELL[1],CELL[2]] = int(i+1)
    return head, list, diff, [x_ext[0], y_ext[0], z_ext[0]]
```

2.3.3 Function for the execution of independent MC experiments

Once all the information that defines the geometry to be analyzed is retrieved, an iterative cycle is generated to perform each MC experiment. For this purpose, the process was encapsulated as a function, monte_carlo_exp. Each independent simulation consists in a single uniform random sampling of each domain strata; therefore, a total of $N = \Pi_{i=1}^{3} n_i$ random points are generated throughout the entire domain, in a non-collapsing way. Each random 3D point is defined as a vector in Cartesian coordinates (i.e., *dart[i]*) and then is parsed to the membership oracle to resolve whether it corresponds to a *'hit'* or a *'miss'*. As for the cylinder-based geometries, cylindrical coordinates are used to stratify the domain; each generated random point (*temp*) must be transformed to its corresponding Cartesian coordinates (*dart*) prior to submission to the membership oracle. These random points correspond to those defined in Eq. 8, where each stratum is defined in Eq. 6 for cube-based and in Eq. 7 cylinder-based geometries.

Once the N random points are evaluated, the quotient of the summation of hits over every dimension and the total generated points is returned to the main program as the computed success probability of the experiment, p_i, according to Eq. 15.

```python
def monte_carlo_exp(H, D, A, x_vec, y_vec, z_vec, r_vec, seeds):
    # Allocates list for sampling results
    sampling_list = np.zeros([nx,ny,nz])
    for k in range(nz):
        for j in range(ny):
            for i in range(nx):
                # Takes a smple from the [i,j,k] stratum
                if A!=0:                    # CUBE
                    dart = [random.uniform(dx*(i),dx*(i+1)), random.uniform(dy*(j),dy*(j+1)), random.uniform(dz*(k),dz*(k+1))]
                if (D != 0 and H != 0):     # CYLINDER
                    tmp = [random.uniform(dr*math.sqrt(i),dr*math.sqrt(i+1)), random.uniform(dt*(j),dt*(j+1)), random.uniform(dz*(k),dz*(k+1))]
                    dart = [tmp[0]*np.cos(tmp[1]), tmp[0]*np.sin(tmp[1]), tmp[2]]
                # Evaluates the sampling point
                sampling_list[i,j,k] = member_oracle(np.asarray(dart), head, list, np.asarray([x_vec, y_vec, z_vec]), r_vec, vecDelta, vecMin)
#   # Computes expected value E[x]
    hits_num = sum(sum(sum(sampling_list)))
    # Returns the computed probability p(x,n) = E[x]/n
    return float(hits_num/seeds)
```

Results in this article were obtained by stratifying the three-dimensional space in 50 strata per spatial dimension, giving a sample size of 125 thousand points for each MC experiment.

2.3.4 Function for the evaluation of membership oracle calls

When any random sampling point, defined by its coordinates (i.e., *vecP[i]*), is parsed to the membership oracle to be evaluated, the function member_oracle was generated to answer based on the information given a priori regarding the list of voids, defined by the center positions, *vecPores*, and its radii, *vecRadii*. This function identifies the corresponding grid cell associated with the coordinates of the random point, and then, it iterates looking for the surrounding cells checking if any of the spheres that lies in the neighborhood of the corresponding cell will satisfy the *'hit'* condition, using the linked lists. This *'hit'* condition, as stated in Eq. 2, means that if the distance from the point to the center of a sphere is less than its radius, then the point lays inside the sphere, returning a 1 as a result, hence *'hit'*, stopping the iteration. In the case of after checking all the relevant cells, no *'hit'* is found, the function returns a 0, hence *'miss'*. The numba decorator jit for Just-In-Time compilation is used to speed up the code.

```python
@numba.jit
def member_oracle(vecP, head, list, vecPores, vecRadii, vecDelta, vecMin):
    # Initiates variable assuming no hit
    Cell = np.zeros(3)
    # Defines Central Cell
    Cell = [math.floor((vecP[i]-vecMin[i])/vecDelta[i]) for i in range(3)]
    # Evaluates arround the central cell
    for cell_x in [int(Cell[0]), int(Cell[0]-1), int(Cell[0]+1)]:
        for cell_y in [int(Cell[1]), int(Cell[1]-1), int(Cell[1]+1)]:
            for cell_z in [int(Cell[2]), int(Cell[2]-1), int(Cell[2]+1)]:
                test = int(head[cell_x, cell_y, cell_z])
                while test != 0:
                    #Evaluate Distance to the Pore
                    if (vecP[0]-vecPores[0,test-1])**2 + (vecP[1]-vecPores[1,test-1])**2 + (vecP[2]-vecPores[2,test-1])**2 <= vecRadii[test-1]**2:
                        return 1
                    #Update Test Pore
                    test = int(list[test-1])
    return 0
```

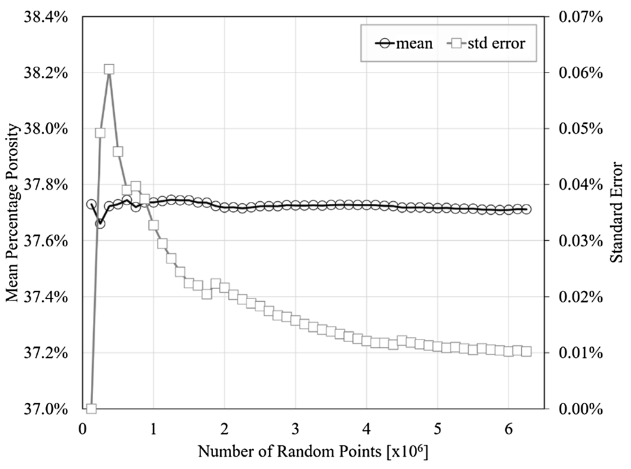


Fig. 4 Computed percentage porosity and standard error versus number of generated random points for a cube-based geometry of size 6.75 [mm] with 4000 pores (CUBE_4000_1) using 125,000 random points per iteration

2.3.5 Main program

To run the previously showed functions, a short main program is coded at the end of the Python script file, where the geometry information (i.e., filename with its extension) is provided as system argument and a few computation parameters (i.e., LHS grid size, uncertainty tolerance, and minimum MC iterations) are defined in lines 2 through 4. After that, the function retrieve_file_info is called in line 17 and all the information regarding the geometry is gathered according to the already explained structure. Subsequently, in lines 19 and 20, the corresponding domain strata are defined based on the retrieved type of geometry. Then, in line 23, the *build_lists* function is called to generate all the required data to speed up the membership oracle response. With this, from line 26 through 31, an iterative procedure is defined to perform subsequent MC experiments by calling the monte_carlo_exp function, which is only exit when both conditions (i.e., minimum number of MC experiments and standard error threshold) are achieved. Finally, in lines 34 and 35, relevant results such as the required number of MC experiments (or iterations) and computed (average) percentage porosity are printed in screen.

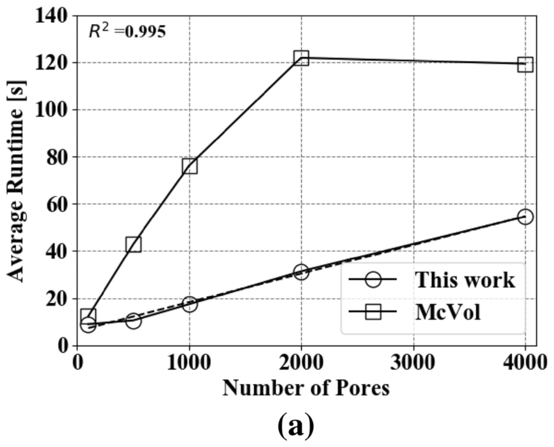
```python
# Input Parameters
n_grid = 50                  # LHS Grid size per dimension
c_int = 5.0*10**(-4)         # Confidence Interval Size
min_iter = 4                 # Minimum MC iterations
# Variables
inppath = sys.path[0]+"/"    # Work Path
c_iter = 1                   # Iterations Counter
usum = 0                     # Sum of computed porosities
sum2 = 0                     # Sum of the square of computed porosities
stderr = 1                   # Std Error of computed porosities
[nx, ny, nz] = [n_grid]*3    # Latin Hypercube Sampling Grid Size
ni = n_grid**3               # Monte Carlo Sampling Size (1x each stratum)
try:
    file = sys.argv[1]
    print("File: "+file)
    # Retrieve File Information
    H,D,A,Xvec,Yvec,Zvec,Rvec,n_sph = retrieve_file_info(inppath + file)
    # Definition of Strata Sizes
    if A != 0: [dx, dy, dz] = [A/n_grid]*3
           # Strata sizes for CUBE
    if D !=0 and H != 0: [dr, dt, dz] = [D/(2*math.sqrt(n_grid)), (2*np.pi)/n_grid, H/n_grid]   # Strata sizes for CYLINDER
    # Linked Lists Building
    print('Building Neighbour Lists')
    head, list, vecDelta, vecMin = build_lists(Xvec, Yvec, Zvec, Rvec)
    # MC Independent Simulations
    print("Computing Volume Using "+str(ni)+" MC Points.")
    while stderr >= c_int/3 or c_iter <= min_iter:
        inst_porosity = monte_carlo_exp(H,D,A,Xvec,Yvec,Zvec,Rvec,ni)
        [usum, sum2] = [usum+inst_porosity, sum2+inst_porosity**2]
        [mean_val, std_dev] = [usum/c_iter, math.sqrt((sum2-(usum**2/c_iter))/c_iter)]          # Mean Value and Std Deviation of Porosity
        stderr = std_dev/math.sqrt(c_iter)
           # Standard Error for current iteration
        c_iter += 1
    print("Porosity Computation: DONE")
    # Prints results in screen
    print('Iterations needed: '+str(c_iter-1))
    print("Computed percentage porosity: "+str(round(mean_val*100,2))+"%")
except:
    print("Error: Unexpected Exit")
    sys.exit(2)
```

3 Results

Algorithm implementations proposed in previous works are focus in using the MC method for the computation of volume in n-dimensional convex polytopes [35, 36]. Algorithms such as the one presented by Liu et al. [36], which is based on a Markov chain method, or the one presented by Emeris and Fisikopoulos [37], based on a 'hit-and-run' random walk method are simple and efficient but limited to convex geometries. On the other hand, regarding non-convex sets, algorithms such the one presented by Cazals et al. [41] and Till et al. [14] give solutions, based on decomposition and the uniformly random sample MC simulations, respectively, to a union of spheres.

In this work, a membership oracle approach has been followed to implement a simple yet efficient algorithm to estimate the actual percentage porosity of three-dimensional non-convex geometries, represented by the intersection of the exterior of a union of spheres and the interior of a bounding volume. This algorithm has been particularized for the case of porous three-dimensional cubic-based and cylinder-based geometries. Percentage porosity, for different instances containing a wide range of number of pores, has been computed using a confidence interval stopping procedure and results have been compared with the reference value obtained by importing the corresponding geometries into ANSYS v18 Design Modeler module to quantify its accuracy. Both computed percentage porosity and runtime have been compared with available program McVol, an MC-based software for protein volume computation developed by Till et al. [14], as a benchmark.

A data set with a total of 40 geometries was used. The data set is composed by 20 cube-based and 20 cylinder-based geometries, all generated using DEM software LIGGGHTS. A total of eight subsets generated by running DEM simulation using 100, 500, 1000, 2000, and 4000 elements, contained in two basic geometry types (i.e., cube and cylinder), were used. For each subset, 200 unique distributions of pores per number of elements used per basic geometry type were generated, from which 200 distinct geometries were then



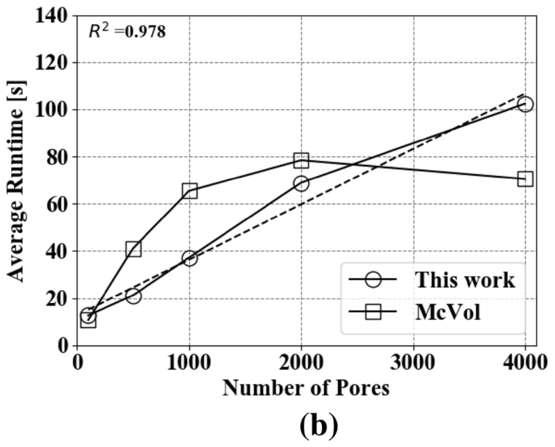


Fig. 5 Average runtime to compute the percentage porosity for geometries containing different number of pores using DMCS (this work) and McVol [14] after 20 independent runs for **a** cube-based geometries and **b** cylinder-based geometries

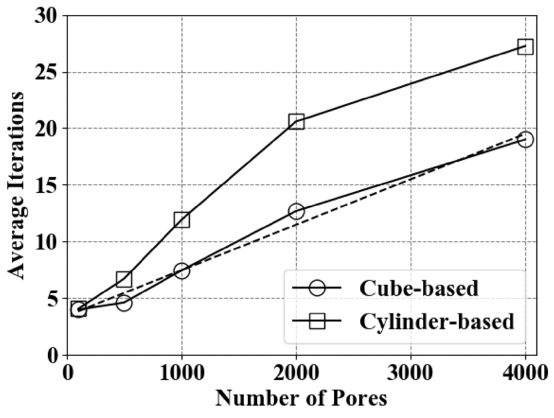


Fig. 6 Average MC iterations needed to achieve convergence of 0.5% uncertainty in standard error versus number of pores of cube-based size 6.75 mm and cylinder-based diameter 6.75 mm and height 5.4 mm foam geometries

Table 1 Average runtime of analysis of 4 different geometries after 20 independent runs using the proposed method and McVol for cube-based size 6.75 mm and cylinder-based diameter 6.75 mm and height 5.4 mm geometries containing pores of 350–450 μm in diameter

Subset	This work		McVol	
	Avg [s]	Std Dev [s]	Avg [s]	Std Dev [s]
CUBE_100	9.1	0.1	12.3	0.1
CUBE_500	10.7	0.1	42.7	0.1
CUBE_1000	17.3	0.1	76.1	0.1
CUBE_2000	31.3	0.1	121.9	0.1
CUBE_4000	54.5	0.6	119.2	12.7
CYL_100	12.8	3.7	10.7	5.2
CYL_500	21.2	2.3	41.1	5.2
CYL_1000	37.8	2.7	65.4	5.1
CYL_2000	69.0	4.4	78.4	10.2
CYL_4000	102.3	8.9	70.5	10.3

created. Later, four geometries were randomly selected for each subset, to obtain a workable size data set. Each geometry was individually identified based on the basic geometry type, the total number of spherical pores, and a correlative (e.g. CUBE_100_1, CYL_2000_3). Results using the proposed method were obtained using an LHS grid of 50 strata per dimension, representing a sample size of 125 thousand random points per iteration and an uncertainty of 0.5% was established as convergence criteria, under the three-sigma rule.

For each geometry in the data set, a total of 20 independent analysis were run using both the proposed method and McVol and loaded into ANSYS v18 Design Modeler module to retrieve the reference value of percentage porosity. To use McVol, all 40 geometries of the data set were transformed to the corresponding input file *.pqr* and scaled, so the maximum pore radius did not exceed 5 [Å], as required by the Till et al. All other setup parameters for McVol were reused from the example case provided in the documentation. For each analysis, both percentage porosity and runtime ware registered and results between the proposed method and McVol were compared. Runtimes were measured using the standard C function *time()*.

To establish the convergence character of the proposed implementation, the instantaneous percentage porosity and the standard error were registered for each iteration of the MC experiment. In Fig. 4, results from the analysis of the cube-based geometry containing 4000 pores, CUBE_4000_1, are shown, where samples of 125,000 random points per iteration were used. As it can be seen, as the number of random points increases, the corresponding mean percentage porosity stabilizes, while the standard error decreases asymptotically. This behavior is expected and predicted in Eq. 18, as the standard error normalized the standard deviation of the measured values by the square root of the number of measurements. Also, as the standard error decreases, the stabilizing behavior of the mean percentage porosity tends to the expected value of the corresponding distribution, and the uncertainty of the confidence interval decreases for a constant confidence level (see Eq. 17).

Regarding runtime requirements, the currently available method to compute the percentage porosity in this type of geometries is to import the geometry into the FEA preprocessor. This process is expensive, especially when geometries contain a large number of pores, as it can be seen in Fig. 2. As an alternative, the DMCS poses a much faster way to estimate the percentage porosity. In Fig. 5, the average runtime for the proposed implementation, for each subset, is provided and compared to that obtained by analyzing the same geometries using McVol. The average runtime was measured for all four geometries in each subset after 20 independent runs.

For the average case, based on the analyzed subsets, the runtime of the proposed algorithm scales linear with respect to the number of contained pores ($R^2 = 0.995$ and $R^2 = 0.978$). In the case of cube-based geometries, Fig. 5a shows that for all the analyzed range, the proposed algorithm is consistently faster than McVol, although, for cylinder-based geometries, in Fig. 5b, results for the proposed implementation show to be faster for geometries containing 2000 pores or less. As suspected, when comparing the average runtime of the proposed implementation for both cube-based and cylinder-based geometries, cube-based geometries exhibit faster average runtimes, of nearly half, than its cylindrical counterparts. Either way, for the analyzed range, the proposed method showed an average convergence time of order $O(n)$, where n is the number of contained pores.

The difference in runtime between cube-based and cylinder-based geometries is believed to be based on two main factors: (a) the fact that cylinder-based geometries require on average more iterations to achieve convergence and (b) each iteration is slower than for cube-based geometries as the equiprobable space is more complex and a transformation from cylindrical to cartesian coordinates must be done after each random point is generated, to be evaluated by the membership oracle. Regarding required number of iterations to achieve convergence, Fig. 6 shows the average number of iterations needed to achieve the established convergence criterion as a function of the number of pores. The average has been measured as the average of the four geometries of each subset. Examination of Fig. 6 shows that for any given number of pores, the number of iterations needed to achieve convergence is consistently higher for cylinder-based geometries than for the cube-based ones. Measurement of the cumulative time required to execute the corresponding lines to generate the 125 thousand random points per iteration have shown a consistent average time of 2.4 μs per point for the cube-based geometries versus 7.84 μs per point for cylinder-based geometries, which, in addition to the larger number of iteration needed, are supporting evidence of both assumptions. More details regarding average and standard deviation in runtime for the proposed method and McVol can be seen in Table 1. Alternatively, this difference in behavior may be influenced by a dependence of the runtime with respect to the percentage porosity, although this relation has not been addressed at this point.

The main objective of the propose method is to estimate the percentage porosity of non-convex sets, and the tested implementation is oriented toward foam geometries, represented by the intersection between the exterior of a union of spheres and the interior of a surrounding primitive geometry such as a cube or a cylinder. The computed percentage porosity obtained by the proposed method, and by McVol, was compared by measuring the absolute error respect to the reference value, which is obtained by loading the corresponding geometries into ANSYS v18 Design Modeler module and retrieving the solid volume from there. Figure 7 shows the results for the cube-based subsets, while Fig. 8 shows the results obtained for the cylinder-based subsets.

More detailed information regarding the minimum, maximum, and average computed percentage porosity, as well as the absolute error, for each of the geometries contained in the data set is provided in Table 2 of "Appendix B". Also, in "Appendix C", Table 3 provides a more detailed information regarding the minimum, maximum, and average runtime for each of the tested geometries using both the proposed implementation and McVol.

Examination of Figs. 7 and 8 shows that although the proposed method tends to produce less precise results; in general, these results are more accurate than those produced by McVol. This difference is based on a key aspect that differentiates the proposed method form it, which is the introduction of the exterior bound. While McVol defines a bounding box, which encloses the complete union of spheres, in our method, the domain is bounded by the exterior primitive volume and allows the spheres to intersect this boundary. Further examination of Fig. 7 shows that the computed results using the proposed method are bounded by 0.35%, which is lower than the 0.5% percent limit established a priori for the uncertainty. On the other hand, detailed examination of Fig. 8 shows that the absolute error distribution for cylinder-based subsets was less predictable than for cube-based ones. Figure 8d shows the widest distribution of error obtained in computing the percentage porosity of cylinder containing 2000 pores, as high as 1%, while Fig. 8a–c, e shows an excellent prediction capability for all the other cases, less than 0.2% in all cases which is, again, less than the prescribed 0.5% uncertainty.

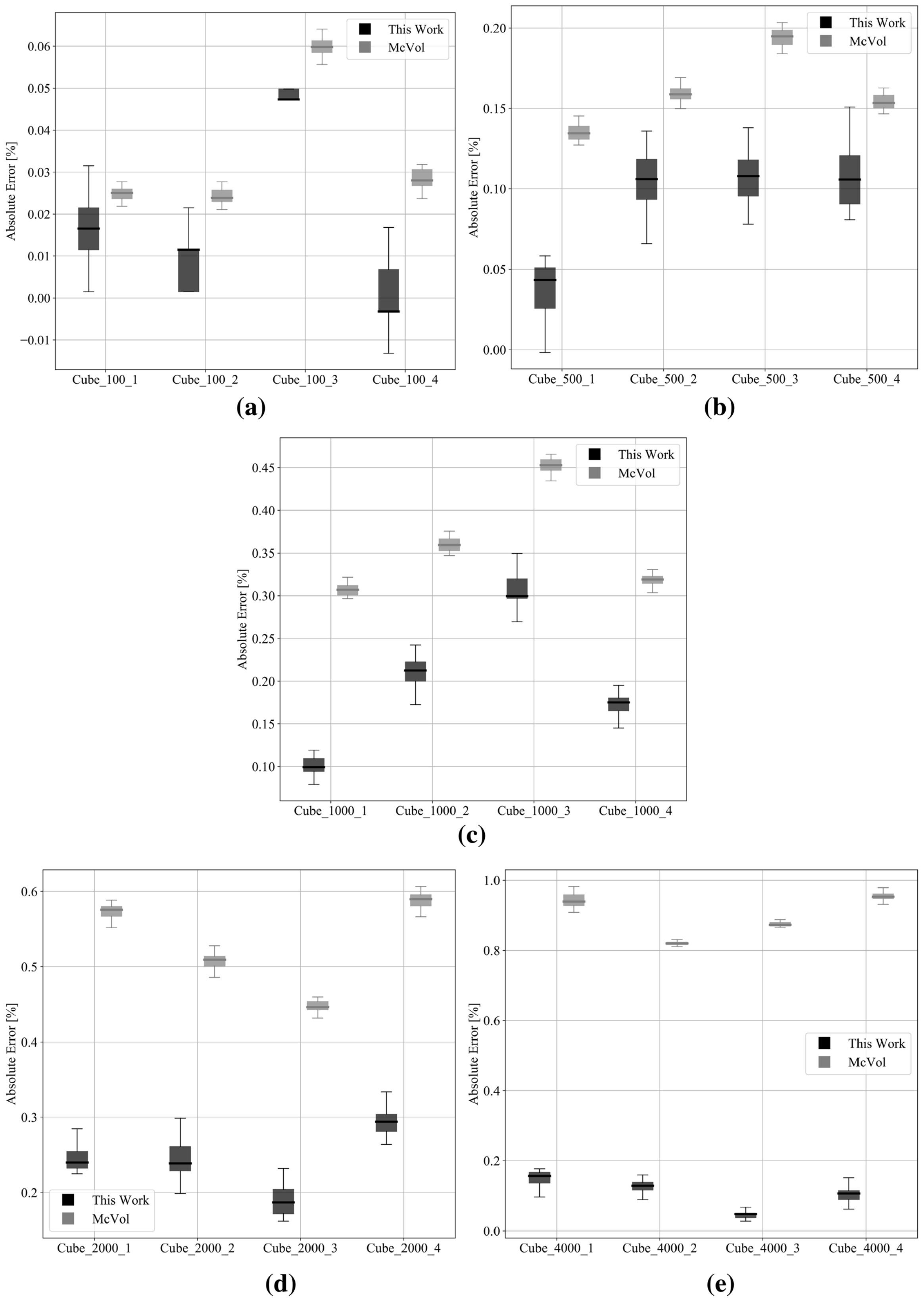


Fig. 7 Comparison of absolute error in percentage porosity computation of cube-based geometries of size 6.5 mm by the proposed method and McVol [14] for geometries containing **a** 100, **b** 500, **c** 1000, **d** 2000, and **e** 4000 distributed pores

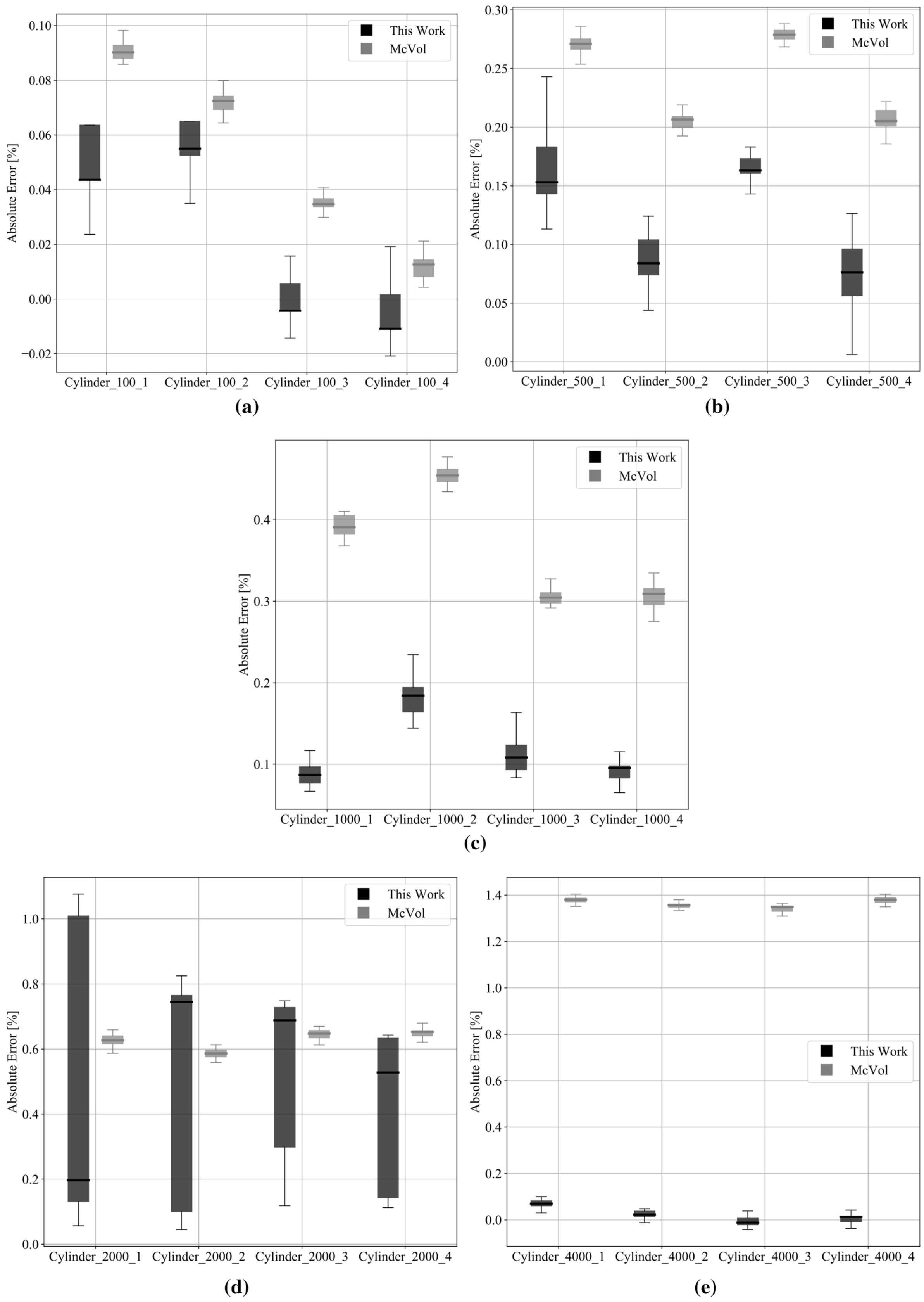


Fig. 8 Comparison of absolute error in percentage porosity computation of cylinder-based geometries of diameter 6.75 mm and height 5.4 mm by the proposed method and McVol [14] for geometries containing **a** 100, **b** 500, **c** 1000, **d** 2000, and **e** 4000 distributed pores

Measurements of the difference between the computed percentage porosity and its reference value for geometries containing a larger number of pores (i.e., greater than 4000 pores) could not be considered for this work due to the excessive time required to import each geometry into the FEA preprocessor, though similar results are expected. On the other hand, the proposed implementation considers just the use of a single-thread algorithm and, due to the potential for parallelization of the independent MC experiments, further optimizations can be done to achieve better time performance. It is known by the authors that Python libraries such as Joblib and Numba provide fast and easy alternatives for that matter.

4 Conclusions

A single-thread algorithm for DMCS, partially based on the one presented by Liu et al. [36], has been implemented in the Python language to estimate percentage porosity in cylindrical and cubical geometries containing interconnected spherical voids analytically defined with spatial distribution obtained by DEM simulation and uniformly random radii, as those required by Perez et al. [10] in the study metallic foams fabricated by means of PM and the SHP technique. The complete Python script was developed using less than 120 code lines, when comment lines are not counted, and only the Python libraries numpy and numba, along with the three build-in modules math, random, and sys have been used.

The proposed implementation showed a significant gain in performance time for the task with respect to the currently used technique, which requires the execution of the complete sequence of CAD extrusion and cut operations by the FEA preprocessor. The time consumption to compute the percentage porosity showed a reduction between 84 and 99% when geometries containing between 100 and 4000 spherical pores were analyzed.

When compared to other similar software, the proposed implementation has shown to be able to achieve consistently smaller errors in approximating the percentage porosity of foam geometries than McVol. These better results are believed to be related to a better suited definition of the domain restrictions of the set. While McVol relies on defining a bounding box that contains the whole union of spheres, the proposed implementation gives a more precise restriction of the outer boundary of the domain, whether it is a box (i.e., a cube) or a cylinder. Although, when cylinders are considered for analysis, a trade-off must be done, and sometimes, performances are loss in exchange for precision. In addition, the proposed implementation relies on a statistical criterion to stop computations, rather than a unique measure or an arbitrary samples size.

The DMCS has been showed to provide a simple yet powerful tool in estimating the porosity percentage in 3D non-convex analytical geometries, as the accuracy in the computed results provides an estimation error below the prescribed uncertainty, with respect to the percentage porosity obtained by the generation of the geometries by the FEA preprocessor, in seven of the eight subsets of the tested data set. This estimation error represents a neglectable difference for the primary purpose for which the algorithm has been developed.

In addition to this implementation, future work related to this algorithm includes further performance improvements by means of parallel computing and its extension to more general geometries such as polyhedrons and other volumes.

Appendix A

In this appendix, the complete Python code script is presented. This code runs in single core configuration allowing to estimate the porosity of cylindrical- or cubic-shaped foam geometries containing randomly distributed spherical pores when a complete analytical description of it (i.e. overall dimensions and pores location and dimension) is given in a JavaScript script file. The code uses Eq. 15 to average the results obtained by a series of Monte Carlo simulations, based on an LHS strategy, according Eqs. 5, 6, 7, and 8. The code will iterate until two established criteria are met, which are a minimum number of iteration and a maximum standard error, according to Eq. 18. This code requires the user to provide the filename with its extension as system argument (e.g., Cube_100_1.js). The path to the file is assumed to be the current work directory

```python
import numpy as np
import numba
import random
import math
import sys
######################
#      FUNCTIONS
######################
def retrieve_file_info(inp_file):
# File is read line by line
    with open(inp_file) as ifile:
        n_void = 0
        radii, x, y, z = [], [], [], []
        for line in ifile:
            if 'var H=' in line:                                        # Cylinder height
                l_H = float(line.strip("var H=").strip(";\n"))
            if 'var D=' in line:                                        # Cylinder diameter
                l_D = float(line.strip("var D=").strip(";\n"))
            if 'var A=' in line:                                        # Cube edge size
                l_A = float(line.strip("var A=").strip(";\n"))
            if ('radio=0;' not in line and 'radio=' in line):           # Sphere radius
                n_void += 1
                radii.append(float(line.strip("radio=").strip("; \n")))
            if ('posicionx=0;' not in line and "posicionx=" in line):   # Sphere center X-coordinate
                x.append(float(line.strip("posicionx=").strip("; \n")))
            if ('posiciony=0;' not in line and "posiciony=" in line):   # Sphere center Y-coordinate
                y.append(float(line.strip("posiciony=").strip("; \n")))
            if ('posicionz=0;' not in line and "posicionz=" in line):   # Sphere center Z-coordinate
                z.append(float(line.strip("posicionz=").strip("; \n")))
        ifile.close()
        # Checks retrieved type of geometry and sets zero the unused variables
        if not 'l_A' in locals():                                       # Checks if is not a Cube
            l_A = 0
        elif not ('l_D' in locals() and 'l_H' in locals()):             # Checks if is not a Cylinder
```

```
            l_H, l_D = 0, 0
        else:                                                           # Acts if does not detect neither a cube nor a cylinder
            l_A , l_D, l_H = 0, 0, 0
            print('Error retrieving file info. Not a Cylinder nor a Cube was found.')
            sys.exit(2)
    return l_H,l_D,l_A,np.asarray(x),np.asarray(y),np.asarray(z),np.asarray(radii),n_void
#
def monte_carlo_exp(H, D, A, x_vec, y_vec, z_vec, r_vec, seeds):
    # Allocates list for sampling results
    sampling_list = np.zeros([nx,ny,nz])
    for k in range(nz):
        for j in range(ny):
            for i in range(nx):
                # Takes a smple from the [i,j,k] stratum
                if A!=0:                    # CUBE
                    dart = [random.uniform(dx*(i),dx*(i+1)), random.uniform(dy*(j),dy*(j+1)), random.uniform(dz*(k),dz*(k+1))]
                if (D != 0 and H != 0):     # CYLINDER
                    tmp = [random.uniform(dr*math.sqrt(i),dr*math.sqrt(i+1)), random.uniform(dt*(j),dt*(j+1)), random.uniform(dz*(k),dz*(k+1))]
                    dart = [tmp[0]*np.cos(tmp[1]), tmp[0]*np.sin(tmp[1]), tmp[2]]
                # Evaluates the sampling point
                sampling_list[i,j,k] = member_oracle(np.asarray(dart), head, list, np.asarray([x_vec, y_vec, z_vec]), r_vec, vecDelta, vecMin)
#   # Computes expected value E[x]
    hits_num = sum(sum(sum(sampling_list)))
    # Returns the computed probability p(x,n) = E[x]/n
    return float(hits_num/seeds)
#
@numba.jit
def member_oracle(vecP, head, list, vecPores, vecRadii, vecDelta, vecMin):
    # Initiates variable assuming no hit
    Cell = np.zeros(3)
    # Defines Central Cell
    Cell = [math.floor((vecP[i]-vecMin[i])/vecDelta[i]) for i in range(3)]
    # Evaluates around the central cell
    for cell_x in [int(Cell[0]), int(Cell[0]-1), int(Cell[0]+1)]:
        for cell_y in [int(Cell[1]), int(Cell[1]-1), int(Cell[1]+1)]:
            for cell_z in [int(Cell[2]), int(Cell[2]-1), int(Cell[2]+1)]:
                test = int(head[cell_x, cell_y, cell_z])
                while test != 0:
                    #Evaluate Distance to the Pore
                    if (vecP[0]-vecPores[0,test-1])**2 + (vecP[1]-vecPores[1,test-1])**2 + (vecP[2]-vecPores[2,test-1])**2 <= vecRadii[test-1]**2:
                        return 1
                    #Update Test Pore
                    test = int(list[test-1])
    return 0
#
def build_lists(x_vec, y_vec, z_vec, r_vec):
    # Temp Lists
    x_ext = [np.amin(x_vec - r_vec), np.amax(x_vec + r_vec)]
    y_ext = [np.amin(y_vec - r_vec), np.amax(y_vec + r_vec)]
    z_ext = [np.amin(y_vec - r_vec), np.amax(z_vec + r_vec)]
    r_max = max(r_vec)
    # Cells definition
    C = [math.ceil((x_ext[1] - x_ext[0])/(2*r_max)), math.ceil((y_ext[1] - y_ext[0])/(2*r_max)), math.ceil((z_ext[1] - z_ext[0])/(2*r_max))]
    diff = [(x_ext[1] - x_ext[0])/C[0], (y_ext[1] - y_ext[0])/C[1], (z_ext[1] - z_ext[0])/C[2]]
    # Arrays Allocation
    list = np.zeros(len(r_vec))
    head = np.zeros((C[0],C[1],C[2]))
    # Lists Filling
    for i in range(len(r_vec)):
        CELL = [math.floor((x_vec[i]-x_ext[0])/diff[0]), math.floor((y_vec[i]-y_ext[0])/diff[1]), math.floor((z_vec[i]-z_ext[0])/diff[2])]
        list[i] =  int(head[CELL[0],CELL[1],CELL[2]])
        head[CELL[0],CELL[1],CELL[2]] = int(i+1)
    return head, list, diff, [x_ext[0], y_ext[0], z_ext[0]]
#
######################
#    MAIN PROGRAM
######################
# Input Parameters
n_grid = 50                   # LHS Grid size per dimension
c_int = 5.0*10**(-4)          # Confidence Interval Size
min_iter = 5                  # Minimum MC iterations
# Variables
inppath = sys.path[0]+"/"     # Work Path
c_iter = 1                    # Iterations Counter
usum = 0                      # Sum of computed porosities
sum2 = 0                      # Sum of the square of computed porosities
stderr = 1                    # Std Error of computed porosities
[nx, ny, nz] = [n_grid]*3     # Latin Hypercube Sampling Grid Size
ni = n_grid**3                # Monte Carlo Sampling Size (1x each stratum)
try:
    file = sys.argv[1]
    print("File: "+file)
    # Retrieve File Information
    H,D,A,Xvec,Yvec,Zvec,Rvec,n_sph = retrieve_file_info(inppath + file)
    # Definition of Strata Sizes
    if A != 0: [dx, dy, dz] = [A/n_grid]*3        # Strata sizes for CUBE
    if D !=0 and H != 0: [dr, dt, dz] = [D/(2*math.sqrt(n_grid)), (2*np.pi)/n_grid, H/n_grid]   # Strata sizes for CYLINDER
    # Linked Lists Building
    print('Building Neighbour Lists')
    head, list, vecDelta, vecMin = build_lists(Xvec, Yvec, Zvec, Rvec)
    # MC Independent Simulations
    print("Computing Volume Using "+str(ni)+" MC Points.")
    while stderr >= c_int/3 or c_iter <= min_iter:
        inst_porosity = monte_carlo_exp(H,D,A,Xvec,Yvec,Zvec,Rvec,ni)
        [usum, sum2] = [usum+inst_porosity, sum2+inst_porosity**2]
        [mean_val, std_dev] = [usum/c_iter, math.sqrt((sum2-(usum**2/c_iter))/c_iter)]          # Mean Value and Std Deviation of Porosity
        stderr = std_dev/math.sqrt(c_iter)        # Standard Error for current iteration
        c_iter += 1
    print("Porosity Computation: DONE")
    # Prints results in screen
    print('Iterations needed: '+str(c_iter-1))
    print("Computed percentage porosity: "+str(round(mean_val*100,2))+"%")
except:
    print("Error: Unexpected Exit")
    sys.exit(2)
```

Appendix B

In this appendix, more detailed information regarding the computed percentage porosity for the data set obtained by means of the proposed algorithm and McVol after 20 independent runs are presented in Table 2.

Table 2 Minimum, maximum, and average percentage porosity and absolute error respect to reference value obtained from ANSYS Design Modeler Module obtained after 20 independent runs using the proposed implementation and McVol

Geometry	This work				McVol			
	Min value (%)	Max value (%)	Avg value (%)	Absolute error (%)	Min value (%)	Max value (%)	Avg value (%)	Absolute error (%)
CUBE_100_1	1.07	1.11	1.09	0.016	1.10	1.11	1.10	0.025
CUBE_100_2	1.08	1.10	1.09	0.009	1.10	1.11	1.10	0.024
CUBE_100_3	1.08	1.11	1.09	0.049	1.10	1.11	1.10	0.060
CUBE_100_4	1.01	1.04	1.02	−0.001	1.05	1.06	1.05	0.028
CUBE_500_1	5.19	5.53	5.25	0.062	5.32	5.34	5.33	0.135
CUBE_500_2	5.43	5.94	5.49	0.126	5.51	5.54	5.52	0.160
CUBE_500_3	5.39	5.92	5.46	0.145	5.50	5.52	5.51	0.194
CUBE_500_4	5.37	6.29	5.44	0.151	5.43	5.46	5.44	0.154
CUBE_1000_1	9.92	10.02	9.97	0.101	10.17	10.19	10.18	0.307
CUBE_1000_2	10.70	10.77	10.74	0.210	10.87	10.90	10.89	0.360
CUBE_1000_3	10.68	10.76	10.72	0.306	10.84	10.88	10.86	0.452
CUBE_1000_4	10.65	10.75	10.68	0.175	10.81	10.84	10.82	0.319
CUBE_2000_1	20.61	20.80	20.67	0.259	20.97	21.00	20.99	0.574
CUBE_2000_2	20.64	20.85	20.70	0.260	20.93	20.97	20.95	0.508
CUBE_2000_3	20.73	21.29	20.80	0.227	21.00	21.03	21.02	0.447
CUBE_2000_4	20.65	21.14	20.71	0.321	20.95	20.99	20.97	0.589
CUBE_4000_1	37.68	37.82	37.74	0.153	38.49	38.57	38.53	0.942
CUBE_4000_2	38.29	38.37	38.34	0.125	39.00	39.06	39.03	0.820
CUBE_4000_3	37.77	37.90	37.80	0.049	38.62	38.66	38.63	0.877
CUBE_4000_4	37.93	38.08	38.03	0.100	38.86	38.91	38.88	0.954
CYL_100_1	1.73	1.77	1.76	0.049	1.79	1.80	1.80	0.091
CYL_100_2	1.70	1.73	1.72	0.055	1.73	1.74	1.74	0.072
CYL_100_3	1.62	1.66	1.64	−0.003	1.67	1.68	1.68	0.035
CYL_100_4	1.66	1.71	1.68	−0.006	1.70	1.71	1.70	0.012
CYL_500_1	8.34	8.68	8.40	0.176	8.48	8.51	8.50	0.271
CYL_500_2	8.54	9.22	8.61	0.116	8.68	8.71	8.70	0.204
CYL_500_3	8.35	8.93	8.42	0.191	8.49	8.53	8.51	0.279
CYL_500_4	8.44	8.75	8.52	0.090	8.62	8.66	8.64	0.206
CYL_1000_1	16.14	16.24	16.19	0.087	16.47	16.51	16.49	0.391
CYL_1000_2	16.47	16.64	16.51	0.188	16.76	16.80	16.78	0.454
CYL_1000_3	16.73	16.84	16.76	0.116	16.94	16.97	16.95	0.305
CYL_1000_4	16.75	16.88	16.82	0.089	17.01	17.07	17.04	0.305
CYL_2000_1	32.00	33.02	32.47	0.528	32.53	32.60	32.57	0.627
CYL_2000_2	31.73	32.51	32.24	0.551	32.24	32.30	32.27	0.586
CYL_2000_3	32.01	32.64	32.43	0.537	32.50	32.58	32.54	0.647
CYL_2000_4	31.56	32.09	31.85	0.400	32.07	32.13	32.10	0.650
CYL_4000_1	56.93	57.07	57.01	0.067	58.29	58.34	58.32	1.379
CYL_4000_2	56.78	56.93	56.82	0.029	58.12	58.17	58.15	1.354
CYL_4000_3	56.76	56.87	56.82	−0.010	58.14	58.20	58.17	1.343
CYL_4000_4	56.97	57.07	57.02	0.009	58.36	58.41	58.39	1.379

Appendix C

In this appendix, more detailed information regarding the computed runtimes for the data set for the proposed algorithm and McVol after 20 independent runs are presented in Table 3.

Table 3 Minimum, maximum, and average runtime after 20 independent runs using the proposed implementation and McVol

	This Work			Runtime		
Geometry	Minimum	Maximum	Average	Minimum	Maximum	Average
CUBE_100_1	8.90	9.40	9.08	12.40	12.60	12.46
CUBE_100_2	8.90	9.30	9.08	12.20	12.80	12.34
CUBE_100_3	8.90	9.20	9.01	12.20	12.30	12.28
CUBE_100_4	8.90	9.80	9.09	12.10	12.50	12.23
CUBE_500_1	9.10	13.60	9.79	33.50	34.00	33.79
CUBE_500_2	9.20	21.80	11.57	45.80	46.50	46.12
CUBE_500_3	9.20	17.50	10.50	45.40	46.30	45.72
CUBE_500_4	9.20	19.70	10.83	45.00	45.60	45.29
CUBE_1000_1	9.60	22.90	13.36	57.90	59.00	58.51
CUBE_1000_2	10.10	48.60	22.43	82.90	84.10	83.45
CUBE_1000_3	9.70	33.50	18.01	80.60	83.40	82.15
CUBE_1000_4	9.70	29.50	15.59	79.50	81.30	80.13
CUBE_2000_1	10.50	62.90	33.33	120.10	121.60	120.86
CUBE_2000_2	10.60	63.00	29.98	120.50	122.50	121.25
CUBE_2000_3	10.60	55.50	32.47	120.20	124.20	121.50
CUBE_2000_4	10.60	53.00	29.51	122.70	124.80	123.81
CUBE_4000_1	12.20	95.50	52.63	118.60	122.80	120.61
CUBE_4000_2	12.30	92.70	54.18	117.80	121.50	119.51
CUBE_4000_3	12.20	96.40	59.27	115.40	119.10	117.48
CUBE_4000_4	12.20	94.60	51.94	117.80	121.80	119.34
CYL_100_1	12.30	13.10	12.62	10.30	10.60	10.39
CYL_100_2	12.20	18.50	13.07	10.80	11.10	10.94
CYL_100_3	12.10	13.30	12.63	10.90	11.10	11.03
CYL_100_4	12.70	15.40	12.76	10.40	10.90	10.56
CYL_500_1	12.50	45.80	22.14	37.60	38.50	37.92
CYL_500_2	12.80	46.30	20.73	42.00	42.50	42.21
CYL_500_3	12.80	44.50	22.91	41.70	42.70	42.12
CYL_500_4	12.70	39.80	18.96	41.90	42.50	42.12
CYL_1000_1	13.00	67.10	33.93	58.40	58.90	58.64
CYL_1000_2	13.40	73.70	36.73	67.70	70.60	68.53
CYL_1000_3	13.30	66.10	39.01	67.00	68.00	67.46
CYL_1000_4	13.10	74.90	41.36	66.60	67.50	66.99
CYL_2000_1	14.30	116.50	55.92	77.50	79.20	78.04
CYL_2000_2	14.60	123.80	78.36	78.60	79.70	78.93
CYL_2000_3	14.30	135.20	59.70	78.70	79.90	79.10
CYL_2000_4	14.50	126.40	81.97	77.20	78.20	77.55
CYL_4000_1	16.10	181.40	91.04	64.00	79.70	71.35
CYL_4000_2	15.90	158.30	95.72	63.60	75.20	65.11
CYL_4000_3	16.00	166.70	100.98	67.20	74.50	69.64
CYL_4000_4	16.00	188.70	121.47	70.50	77.40	75.76

References

1. Banhart J (2001) Manufacture, characterisation and application of cellular metals and metal foams. Prog Mater Sci 46(6):559–632. https://doi.org/10.1016/S0079-6425(00)00002-5
2. Ashby MF, Evans A, Fleck NA, Gibson LJ, Hutchinson JW, Wadley HNG, Delale F (2001) Metal foams: a design guide. Appl Mech Rev 54:B105. https://doi.org/10.1016/s0261-3069(01)00049-8
3. Hasan A (2010) An improved model for FE modeling and simulation of closed cell Al-alloy foams. Adv Mater Sci Eng 1:12. https://doi.org/10.1155/2010/567390
4. Geers MG, Kouznetsova VG, Brekelmans WA (2010) Multi-scale computational homogenization: trends and challenges. J Comput Appl Math 234(7):2175–2182. https://doi.org/10.1016/j.cam.2009.08.077
5. Kanit T, Forest S, Galliet I, Mouroury V, Jeulin D (2003) Determination of the size of the representative volume element for random composites: statistical and numerical approach. Int J Solids Struct 40(13–14):3647–3679. https://doi.org/10.1016/S0020-7683(03)00143-4
6. Kari S, Berger H, Rodriguez-Ramos R, Gabbert U (2007) Computational evaluation of effective material properties of composites reinforced by randomly distributed spherical particles. Compos Struct 77:223–231. https://doi.org/10.1016/j.compstruct.2005.07.003
7. Stefanou G, Savvas D, Papadrakakis M (2017) Stochastic finite element analysis of composite structures based on mesoscale random fields of material properties. Comput Methods Appl Mech Eng 326:319–337. https://doi.org/10.1016/j.cma.2017.08.002
8. Cadena JH, Alfonso I, Ramírez JH, Rodriguez-Iglesias V, Figueroa IA, Aguilar C (2014) Improvement of FEA estimations for compression behavior of Mg foams based on experimental observations. Comput Mater Sci 91:359–363. https://doi.org/10.1016/j.commatsci.2014.04.065
9. Pérez L, Lascano S, Aguilar C, Domancic D, Alfonso I (2015) Simplified fractal FEA model for the estimation of the Young's modulus of Ti foams obtained by powder metallurgy. Mater Des 83:276–283. https://doi.org/10.1016/j.matdes.2015.06.038
10. Pérez L, Lascano S, Aguilar C, Estay D, Messner U, Figueroa IA, Alfonso I (2015) DEM–FEA estimation of pores arrangement effect on the compressive Young's modulus for Mg foams. Comput Mater Sci 110:281–286. https://doi.org/10.1016/j.commatsci.2015.08.042
11. Pérez L, Mercado R, Alfonso I (2017) Young's modulus estimation for CNT reinforced metallic foams obtained using different space holder particles. Compos Struct 168:26–32. https://doi.org/10.1016/j.compstruct.2017.02.017
12. Pérez L, Cabrera I, Santiago AA, Vargas J, Beltrán A, Alfonso I (2018) Effect of the Al–CNT interlayer on the tensile elastic modulus of Al matrix composites with random dispersion of CNTs. J Braz Soc Mech Sci Eng 40(11):550. https://doi.org/10.1007/s40430-018-1473-1
13. Janssen H (2013) Monte-Carlo based uncertainty analysis: sampling efficiency and sampling convergence. Reliab Eng Syst Saf 109:123–132. https://doi.org/10.1016/j.ress.2012.08.003
14. Till MS, Ullmann GM (2009) McVol—A program for calculating protein volumes and identifying cavities by a Monte Carlo algorithm. J Mol Model 16(3):419–429. https://doi.org/10.1007/s000894-009-0541-y
15. Cundall P, Stack O (1979) A discrete numerical model for granular assemblies. Geotechnique 29(1):47–65. https://doi.org/10.1680/geot.1979.29.1.47
16. Torres Y, Pavón JJ, Nieto I, Rodriguez JA (2011) Conventional powder metallurgy process and characterization of porous titanium for biomedical applications. Metal Mater Trans B 42(4):891–900. https://doi.org/10.1007/s11663-011-9521-6
17. Jha N, Mondal DP, Majumdar JD, Badkul A, Khane AK (2013) Highly porous open cell Ti-foam using NaCl as temporary space holder through powder metallurgy route. Mater Des 47:810–819. https://doi.org/10.1016/j.matdes.2013.01.005
18. Li K, Gao XL, Subhash G (2005) Effect of cell shape and cell wall thickness variations on the elastic properties of two-dimensional cellular solids. Int J Solids Struct 42(5–6):1777–1795. https://doi.org/10.1016/j.ijsolstr.2004.08.005
19. Nitka M, Combe G, Dascalu C, Desrues J (2011) Two-scale modeling of granular materials: a DEM-FEM approach. Granular Matter 13(3):277–281. https://doi.org/10.1007/s10035-011-0255-6
20. Soro N, Brassart L, Chen Y, Veidt M, Attar H, Dargusch MS (2018) Finite element analysis of porous commercially pure titanium for biomedical implant application. Mater Sci Eng A 725:43–50. https://doi.org/10.1016/j.msea.2018.04.009
21. Schröder J, Balzani D, Brands D (2010) Approximation of random microstructures by periodic statistically similar RVE based on linear-path functions. Arch Appl Mech 81(7):975–997. https://doi.org/10.1007/s00419-010-0462-3
22. Smit R, Brekelmans W, Meijer H (1998) Prediction of the mechanical behavior of nonlinear heterogeneous systems by multi-level finite element modeling. Comput Methods Appl Mech Eng 155(1–2):181–192. https://doi.org/10.1016/s0045-7825(97)00139-4
23. Báez-Pimiento S, Hernández-Rojas M, Palomar-Pardavé M (2015) Processing and characterization of open-cell aluminum foams obtained through infiltration process. Proc Mater Sci 9:54–61. https://doi.org/10.1016/j.mspro.2015.04.007
24. Orbulov IN (2013) Metal matrix syntactic foams produced by pressure infiltration—the effect of infiltration parameters. Mater Sci Eng A 583:11–19. https://doi.org/10.1016/j.msea.2013.06.066
25. Cleary PW (2005) A multiscale method for including fine particle effects in DEM models of grinding mills. Miner Eng 84:88–99. https://doi.org/10.1016/j.mineng.2015.10.008
26. Coetzee C (2017) Review: calibration of the discrete element method. Powder Technol 310:104–142. https://doi.org/10.1016/j.powtec.2017.01.015
27. Boemer D, Ponthot J-P (2016) DEM modeling of ball mills with experimental validation: influence of contact parameters on charge motion and power draw. Comput Part Mech 4(1):53–67. https://doi.org/10.1007/s40571-016-0125-4
28. Kloss C, Goniva C, Hager A, Amberger S, Pirker S (2012) Models, algorithms and validation for opensource DEM and CFD-DEM. Prog Comput Fluid Dyn Int J 12(2/3):140–152. https://doi.org/10.1504/pcfd.2012.047457
29. ASTM (2018) Standard testing methods of compression testing of metallic materials in room temperature. ASTM International. https://doi.org/10.1520/f0067-13r17
30. Dyer ME, Frieze AM (1988) On the complexity of computing the volume of a polyhedron. SIAM J Comput. https://doi.org/10.1137/0217060
31. LG Khachiyan (1988) On the complexity of computing the volume of a polytope. Izvestia Ajad. Nauk SSSR, Engineering Cybertics. 216–217
32. Khachiyan LG (1989) The problem of calculating the volume of a polyhedron is enumerably hard. Russ Math Surv 44(3):199–200. https://doi.org/10.1070/rm1989v044n03abeh002136
33. Chazelle BM (1981) Convex decompositions of polyhedra. In: Proceedings of the thirteenth annual ACM symposium on Theory of computing, Milwaukee. https://doi.org/10.1145/800076.802459
34. Bueler B, Enge A, Fukuda K (2000) Exact volume computation for polytopes: a practical study. Polytopes—combinatorics and computation. Springer Basel AG, Basel, pp 131–154

35. Ge C, Ma F (2015) A fast and practical method to estimate volumes of convex polytopes. In: International Workshop of Frontiers in Algorithms, 2015. https://doi.org/10.1007/978-3-319-19647-3_6
36. Liu S, Zhang J, Zhu B (2007) Volume computation using a direct Monte Carlo method. Comput Comb Banff. https://doi.org/10.1007/978-3-540-73545-8_21
37. Emiris I, Fisikopoulos V (2014) Efficient random-walk methods for approximation polytope volume. Proceedings of the thirtieth annual symposium on Computational geometry, Kyoto. https://doi.org/10.1145/2582112.2582133
38. Lien JM, Amato N (2007) Approximate convex decomposition of polyhedra. In: Proceedings of the 2007 ACM symposium on Solid and physical modeling, Beijing, China, 2007. https://doi.org/10.1145/1236246.1236265
39. Morris C, Stark R (2015) Finite mathematics: models and applications. John Wiley & Sons, Hoboken
40. Suadhakar Y, Wall W (2013) Quadrature schemes for arbitrary convex/concave volumes and integration of weak form in enriched partition of unity methods. Comput Methods Appl Mech Eng 258:39–54. https://doi.org/10.1016/j.cma.2013.01.007
41. Cazals F, Kanhere H, Loriot S (2011) Computing the volume of a union of balls: a certified algorithm. ACM Trans Math Softw 38(1):1–20. https://doi.org/10.1145/2049662.2049665
42. Kaas R, Buhrman JM (1980) Mean, median and mode in binomial distributions. Stat Neerl 34(1):13–18. https://doi.org/10.1111/j.1467-9574.1980.tb00681.x
43. Canavos GC (1984) Applied probability and statistical methods. Little, Brown
44. Ballio F, Guadagnini A (2004) Convergence assessment of numerical Monte Carlo simulations in groundwater hydrology. Water Resour Res. https://doi.org/10.1029/2003wr002876
45. Gilman M (1968) A brief survey of stopping rules for Monte Carlo. In: Second conference on applications of simulations, New York, NY