

Tree codes and sort-and-sweep algorithms for neighborhood computation: A cache-conscious comparison

Dominik Krengel^{a,*}, Yuki Watanabe^b, Ko Kandori^b, Jian Chen^c, Hans-Georg Matuttis^b

^a*Department of Marine Resources and Energy, Tokyo University of Marine Science and Technology, 4-5-7, Konan, Minato, 108-8477, Japan*

^b*Department of Mechanical and Intelligent Systems Engineering, The University of Electro-Communications, 1-5-1, Chofugaoka, Chofu, 182-8585, Japan*

^c*Center for Mathematical Science and Advanced Technology, Japan Agency for Marine-Earth Science and Technology, 3173-25, Showa-machi, Kanazawa-ku, Yokohama, 236-0001, Japan*

Abstract

Neighborhood algorithms may take a considerable percentage of computer time in discrete element methods (DEM). While the sort-and-sweep algorithm is ideal in some ways, as it only deal with particles whose relative positions change in one coordinate direction, the other directions must be processed too, for all particles. In contrast, tree-codes deal only with adjacent particles. We compare sort-and-sweep and tree-code neighborhood algorithms for two-dimensional DEM simulations of polygonal particles in a rotating drum with up to 12000 particles. We discuss the effects of system size and inlining on the performance with respect to the cache memory. For the tree code, the performance is slightly better, at the cost of significantly increased cyclomatic complexity. In particular, one benefit is improved possibilities for shared memory parallelization.

Keywords: Discrete element method, Neighborhood computations, Tree codes, Sort-and-sweep, Performance optimization

1. Introduction

For the discrete element method (DEM), it is necessary to compute the interaction of all particles in contact. To efficiently evaluate short term contact forces between contacting particles and skip computations for non-contacting particles, a variety of different neighborhood algorithms have been devised (see e.g. Ref. [1] for an overview). Some neighborhood algorithms (such as Verlet tables or the linked cell approach) must either rebuild the contact lists from scratch for all particles in each timestep, or make risky assumptions about the maximum range of particle motion when rebuilding the list after a certain number of timesteps. Preferably, there should be less work if there is less motion in the system, especially for dense systems where the range of motion is limited by adjacent particles and by the maximum permissible stepsize of the time integrator. Ideally, contact (or neighborhood) lists (the pairs of indices

19 fore the particles which can be in contact on geometrical grounds) in DEM simulations are only updated after changes in relative positions. Algorithms that allow such an update-oriented implementation for bounding boxes (the extremal coordinates) are the sort-and-sweep algorithm [1] as well as tree codes [2, 3], which will be described further in Section 2 and 3, respectively.

Updating only the changes in the data structure while keeping the unchanged interacting pairs in the contact list not only increases the efficiency for the neighborhood computation, but also has another advantage compared to rebuilding the neighborhoods from scratch: One can, additionally to the particle indices, store a “payload”, i.e. additional information about the contacting particles, e.g. the friction value for the previous time step which is needed for Cundall–Strack friction [4] or the first time of contact for other hysteretic friction forces which change with the time of stick [5]. While for simulations of non-spherical particles in three dimensions, the bulk of the computational effort is spent in the overlap computation [6], for spherical particles or two-dimensional simulations, the neighborhood algorithm may become a performance-limiting factor.

*Corresponding author
Email address: dominik.krengel@kaiyodai.ac.jp
(Dominik Krengel)

42 The computational complexity of both the sort-and-
 43 sweep algorithm and the tree codes for N objects built
 44 from scratch is $O(N \log N)$. Both algorithms profit from
 45 the use of axis-aligned bounding boxes, i.e. the extremal
 46 coordinates, instead of using the full geometrical infor-
 47 mation of the particle outline. When bounding boxes
 48 are updated, only changes in the relative positions of the
 49 particles will affect the neighborhood list, which is only
 50 a fraction of the actual particles. The computational ef-
 51 fort is $O(N)$ for updating the bounding boxes and ver-
 52 ifying the relative position of the particles, but the re-
 53 arrangement of the information related to the changed
 54 neighborhoods is considerably less than N operations.
 55 To verify the theoretical overall $O(N)$ -complexity, we
 56 will analyze the performance of DEM simulations with
 57 implementations of the sort-and-sweep algorithm and
 58 the tree codes.

59 Conventional complexity analysis in computer sci-
 60 ence focuses on the number of necessary operations and
 61 neglects the time of transferring the necessary data be-
 62 tween memory and CPU. This may lead to an actual
 63 performance which deviates from the theoretical pre-
 64 dictions, as in cache-based processor architectures the
 65 performance drops due to cache misses and reloading of
 66 data from the next larger, lower memory unit whenever
 67 an amount of data corresponding to the cache size has
 68 been used. This effect can downgrade the overall perfor-
 69 mance significantly but is generally not reflected in the
 70 “cache-oblivious” complexity analysis. For linear alge-
 71 bra, pioneering work on cache effects has been done by
 72 benchmarking of vectorized matrix-matrix multiplica-
 73 tions with different kernels, where the CPU time could
 74 increase from the theoretical $O(N^3)$ by one to two orders
 75 for unfavorable algorithms and matrix sizes [7]. The
 76 theoretical evaluation of cache misses is tedious and de-
 77 pends on hardware details which change with the the
 78 bus, i.e. with every different mainboard. Nevertheless,
 79 a cache-conscious working knowledge of algorithms is
 80 necessary to decide on applicability based on the per-
 81 formance analysis of actual implementations for both
 82 sort-and-sweep and tree codes.

83 In this work, we explore the relative merits of sort-
 84 and-sweep implementations compared to tree codes for
 85 two dimensions. We use a DEM code for polygons [1]
 86 with slightly elongated particles, which should be rep-
 87 resentative of any (nearly) impenetrable DEM particles
 88 of approximately the same size, see Fig. 1. To under-
 89 stand the impact of implementation details, we bench-
 90 mark both the sort-and-sweep algorithm and the tree
 91 code. We measure the relative performance of the two
 92 algorithms on machines with different cache sizes (see
 93 Tab. 1). The main part of the timings are measured for

94 MATLAB interpreter code. Additionally, the codes are
 95 translated, compiled and benchmarked as C-Code via
 96 the “MATLAB Coder” to see how compiler language
 97 code should perform. Further, we check the influence
 98 of the function inlining and analyze the algorithm com-
 99 plexity in terms of cyclomatic complexity [8, 9]. The
 100 benchmarking and analysis in this study should help to
 101 understand and improve the performance of neighbor-
 102 hood detection for DEM simulations. Both sort-and-
 103 sweep algorithms and tree codes based on updating the
 104 bounding boxes are very efficient for “solid” particles
 105 with contact interactions. Nevertheless, there are draw-
 106 backs: These codes may be less efficient for large over-
 107 laps, long range forces and many interactions due to too
 108 many changes in the neighborhoods, which result in too
 109 many updating operations per timestep. Examples for
 110 such systems are the representative masses in methods
 111 like smoothed particle hydrodynamics (SPH) [10] and
 112 moving particle semi-implicit (MPS) [11]. For the same
 113 reason, the algorithms discussed in this paper may also
 114 be less efficient for simulations with additional or ex-
 115 clusive long-range forces, such as DEM with van der
 116 Waals forces for clay [12] where the cutoff potential is
 117 too long, as well as for gravitational and electrostatic
 118 Coulomb forces.

2. Sort-and-sweep neighborhood computations

119 “Sort-and-sweep” for neighborhood computations
 120 was introduced by Baraff [13] and later referred to as
 121 “sort-and-prune” [14]. The extremal coordinates of par-
 122 ticles (referred to as “bounding boxes”) are sorted in a
 123 list, together with the ownership index of the bounding
 124 boxes. Whenever a lower bounding box coordinate has
 125 moved below the top bounding box of another particle,
 126 a collision becomes possible. In Fig. 2, three bounding
 127 boxes are shown. From timestep n to timestep $n+1$, par-
 128 ticles 1 and 2 have started to overlap. First, the positions
 129 of the bounding boxes in the list ordered according to
 130 timestep n are updated with the new positions. During
 131 re-sorting, it turns out that the top coordinate t_1 of parti-
 132 cle 1 and the bottom coordinate b_2 of particle 2 have to
 133 be exchanged, which means that there is a possible col-
 134 lision between particles 1 and 2. When for two dimen-
 135 sions the same algorithm is used, particles that move
 136 diagonally towards each other could be entered twice
 137 in the neighborhood list. This can be avoided by using
 138 additionally a list of old bounding boxes. New over-
 139 laps along the x-coordinates are always entered, while
 140 overlaps along the y-axis are only entered if there was
 141 a previous overlap in the x-direction. For sorting, pair-
 142 wise exchange (“bubble sort”) can be used, as the list is

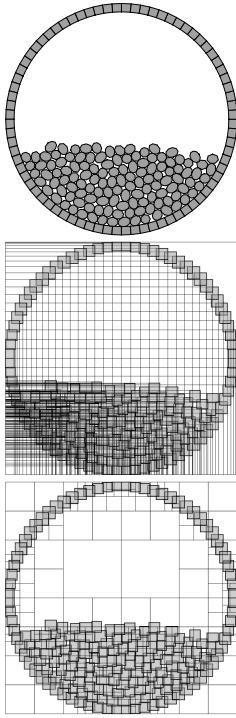


Figure 1: A downscaled drum with fewer particles (top) with (only) upper bounding box coordinates in x- and y-direction for the sort-and-sweep algorithm, which exceed the number of possible particle contacts (middle), compared to the bounding boxes in the tree code which are based partitionings of the system (bottom).

165
166
167
168
169
170
171
172
173

we have devised an algorithm that in each timestep only updates existing tree structures. With two-dimensional (quad-) trees, instead of left and right as in binary trees, we have to deal with four directions: Northeast (NE), Northwest (NW), Southeast (SE), and Southwest (SW), as in Fig. 3 (upper left). Child nodes are obtained by subdividing a region into four symmetric quadrants (upper right), with the old node of the original larger region becoming the parent node.

144 already partially sorted. At the initialization, for unfa-
145 vorable orderings, this might lead to many $O(N^2)$ opera-
146 tions. At the initialization, we can enter first the centers
147 of mass instead of the bounding boxes and order those
148 with a faster sorting algorithm like quicksort. Then we
149 replace the centers of mass with the bounding boxes and
150 run the incremental sort over this list to obtain the ini-
151 tial contact list. For the release of contact, another loop
152 should be used which eliminates all pairs of particles
153 where the bounding boxes have no overlap anymore.

174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196

Next, when the particle occupation is drawn in, one sees that there are two possibilities for the partitioning of the cells and the book-keeping for the tree: A “minimum tree” uses on all levels of the tree the largest possible cells so that neighboring cells can be of different sizes (Fig. 3, lower left). The alternative approach (chosen by Vemuri et al. [2, 3], albeit in three dimensions) assigns only equally sized, minimum cells, all on the lowest, “leaf”-level of the tree (Fig. 3, lower right). In that case, the art is the “pruning” of the stored tree structure so that only occupied branches have to be administered. It will later turn out that our “minimum tree” allows a search of neighbors in linear time, because the tree structure contains the information about neighboring occupied cells. In contrast, Vemuri’s approach needs $N \log N$ operations, as for all N cells, neighboring cells may be connected to tree branches at any of the $\log N$ higher levels. For the main part of this article, we treat particles with a small size dispersion and slight elongation, see Fig. 1. Wall particles and other large particles are sub-partitioned into several to many bounding boxes, and the treatment will be explained in section 3.4.

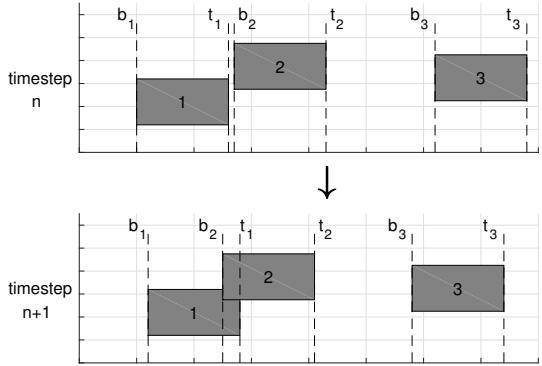


Figure 2: Effective step of the sort-and-sweep algorithm where from timestep n to timestep $n + 1$, the bounding box coordinates b_2 and t_1 changing their order along the x-axis, so that the particle pair 1 and 2 has to be registered in the neighborhood list.

154 **3. Tree codes for neighborhood computation**
155 As in the case of the sort-and-sweep approach, it
156 is reasonable to also use the tree code for bounding
157 boxes instead of detailed particle geometries. In each
158 timestep, flexible restructuring of the tree is necessary
159 after the positions have been updated, which is a more
160 complex task than the updating of bounding box lists by
161 re-sorting. An algorithm that builds tree structures from
162 scratch at every timestep at a cost $N \log N$ will not be
163 competitive with a sort-and-sweep approach which only
164 updates the list and deals with the changes. Therefore,

Northwest (NW)	Northeast (NE)
Southwest (SW)	Southeast (SE)

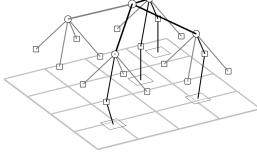
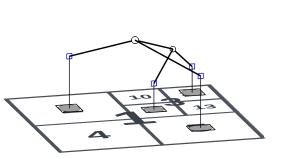
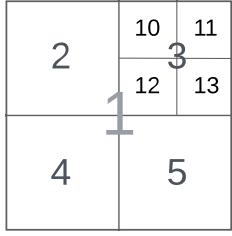


Figure 3: Directions in the quadtree (upper left), an example of cells numbering with occupied cells (upper right) and, for the same occupation, our “minimal tree” (lower left), in contrast to the “pruned tree” of Vemuri et al. [2, 3] (lower right), where only the black branches are used while the gray ones may become activated when the particles move.

197 3.1. History of tree codes

The terminology of “trees” was introduced by Cayley into the mathematical context during the 19th century and the use of tree data structures in computer science gained momentum from the 1960s onward. Finkel and Bentley [15] introduced the terminology of quadtrees for two-dimensional and octrees for three-dimensional structures. Binary (one-dimensional), quad- and octrees operate on fixed boundaries, while for k -d-trees [16], possible changes of the boundaries are implied. Tree structures were first introduced into physics simulations to reduce the computational complexity $O(N^2)$ of “long-range” interactions from gravitation in the 1980s [17]. Such simulations did not use trees for neighborhood computations, as all particles interact anyway, but for successive force summations and truncated calculation of forces via the locally obtained centers of mass [18]. In the field of rigid-body-solid modeling, M. C. Lin [19] used tree codes in a closest-feature algorithm for polytopes and proposed tree codes for neighborhood algorithms, but did not implement them. Vemuri et al. [2, 3] implemented in three dimensions an algorithm that needs $O(N)$ for the updates but has $N \log(N)$ complexity for neighborhood finding as the worst case for monodisperse spherical particles. We will compare this later with our approach with small polydispersity and different spatial partitioning. Wackenhorst [20] implemented a tree code for polydisperse round particles, but no implementation details, timings, or successive publications are available. Schwartz et al. [21] used for a soft sphere DEM a high-performance parallel gravity

228 tree code (originally designed for astrophysics applications by Stadel [22] and Richardson [23]) which generated contact lists in $N \log(N)$ time, but no further algorithmic details are available. In the context of tree codes 229 in DEM, Duriez et al. [24] used tree structures for DEM 230 particles where the interaction computation was defined 231 via level sets.

235 3.2. Binary tree in one dimension

The salient points of the neighborhood computation 236 via tree codes are better explained for one dimension 237 first. First, the spatial partitionings are iteratively halved 238 until only a single center of mass is contained in a single 239 partitioning (“Before” in Fig. 4). Particles in adjacent 240 partitionings (in the “leaf nodes” of the “Tree before 241 updates” in Fig. 4) are entered in the contact list as 242 potential collision partners. (In the case where they cannot 243 interact on geometrical grounds, they are leapfrogged in 244 the overlap computation based on their non-overlapping 245 bounding boxes.) In Fig. 4, the leaf nodes with the 246 particle numbers $i1 \dots i8$ are drawn in color, and the parent 247 nodes are in gray. The indices of the array elements 248 (while the tree structure can also be constructed using 249 pointers, in our approach we use list-based trees) where 250 the nodes of our list-based tree are stored are given by 251 the black pairs of numbers in brackets (m, n) , where m 252 indicates the level ($m = 1$ is the root, i.e. the top most 253 node of the tree), and n indicates the occupied node in 254 the tree. The tree structure is obtained by recursively 255 partitioning the system until each leaf node contains 256 only a single bounding box. The red nodes are leaf 257 nodes that are assigned to only one bounding box in a 258 single cell. In our approach leaf nodes designate cells of 259 varying sizes. The parent nodes don’t “own” any particle 260 (bounding box). The nodes with yellow and orange 261 rims have to be newly created during node partitioning, 262 when temporarily two or more bounding boxes are 263 assigned to them during node splitting. Before the update, 264 we have eight particles, $i1 \dots i8$, at different positions in 265 cells of different sizes, at the leaves (lowest ends of each 266 branch) in the tree. Next, $i5$ (in green) moves physically 267 to the “east” over the center of the system, into the cell 268 which is currently occupied by $i6$. Accordingly, in step 269 1, $i5$ must move upward in the tree structure to the highest 270 node of the tree which is suitable for its new position. 271 In step 2 in Fig. 4, particle 5 has begun its descent 272 downwards in the tree, towards the leaf nodes. When, in 273 step 3, it encounters the node that is already occupied by 274 $i6$, two new leaf nodes must be created one level lower, 275 and the cell size is reduced. In our approach, in one 276 dimension, cells are of different sizes and all of them are 277 occupied, while the approach by Vemuri et al. [2, 3], all 278

279 cells would have the same size, but some of them would
280 not be occupied.

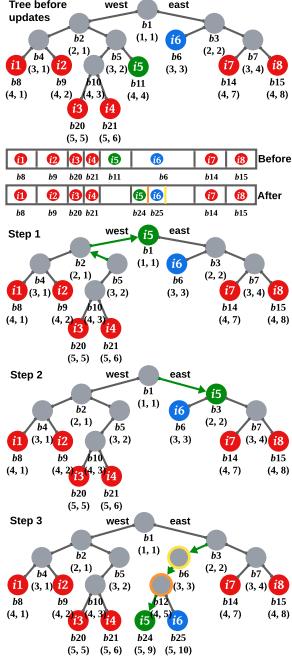


Figure 4: Example for the tree code in one dimension, with the original tree structure on top, followed by the occupation and size of the cells before and after the movement of particle i_5 . The following three trees show the rising of the particles to the highest node (Step 1), followed by the descent (Step 2) until an occupied node is met, and the creation of two leaf nodes on the lower level (Step 3).

281 3.3. Implementation of tree codes in two dimensions

282 Figure 5 shows the program flow for the two-
283 dimensional quadtree: First, the tree is initialized and
284 updated for equal-sized particles, so that the neighbor-
285 hood bounding box pair list can be initialized and up-
286 dated. The initialization of the tree structure consists of
287 two parts: The first is the construction of the tree struc-
288 ture, where the system is divided regularly to create a
289 coordinate array of dividing lines. The set of bound-
290 ing boxes is distributed according to the dividing lines.
291 Then, bounding boxes are assigned to new child nodes
292 that are independent of each other, which may necessi-
293 tate adding new nodes to the tree. The second part is the
294 construction of the list of pairs of neighboring bound-
295 ing boxes, based on the relative positions of nodes and
296 the bounding boxes allotted to them. The full tree is
297 constructed only at the beginning, and later only incre-
298 mental changes are performed during updating, which
299 reduces the amount of computation, as particles with
300 unchanged relative position do not have to be treated.

The computational effort during the simulation depends only on the time that is necessary for the update, while the initialization process occurs only once at the beginning of the program so its computational cost is negligible.

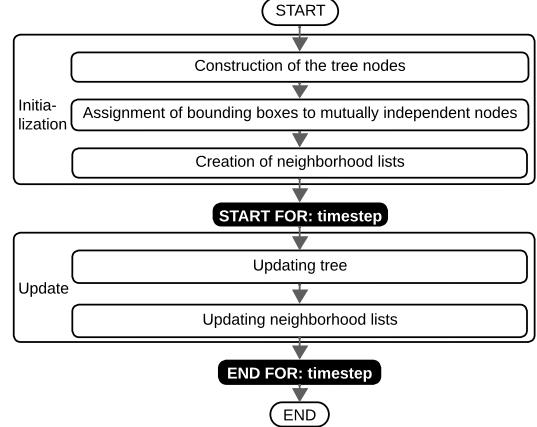


Figure 5: Simplified flowchart for the quadtree neighborhood algorithm in two dimensions.

306 For quadtrees in two dimensions, division into child
307 nodes is analogous to division in the one-dimensional
308 binary trees. One difference is that in a quadtree, there
309 may be empty leaf nodes that have no bounding boxes
310 assigned to them. The update of the tree in Fig. 6, af-
311 ter the particles have new positions, proceeds as in one
312 dimension: The particles move up to the highest pos-
313 sible parent node that can be computed from their xy-
314 coordinates, then they climb down the tree according
315 to the directions from their coordinates. At the low-
316 est level, they are either assigned to existing empty leaf
317 nodes, or, if the leaf nodes at the respective position
318 are already occupied, the leaf node is split and the two
319 particles are assigned to a new leaf node on a level be-
320 low. The updating is inherently sequential, at least in
321 the current program version, but, as will be discussed in
322 section 4, less costly than the updating in the sort-and-
323 sweep algorithm.

324 While in one dimension, only particles in the neigh-
325 boring cells along one direction (towards East and West)
326 exist, in two dimensions, four directions (Fig. 3, upper
327 left) must be dealt with. The existence of empty nodes
328 must also be taken into account when traversing the tree
329 during the construction of the contact list. While for the
330 one-dimensional tree, the search for collision partners
331 could be terminated in a direction where particles were
332 too far away, in two dimensions it may be that a possi-
333 ble interaction partner is stored in a cell that is beyond
334 an empty cell. Additionally, there may be close particles

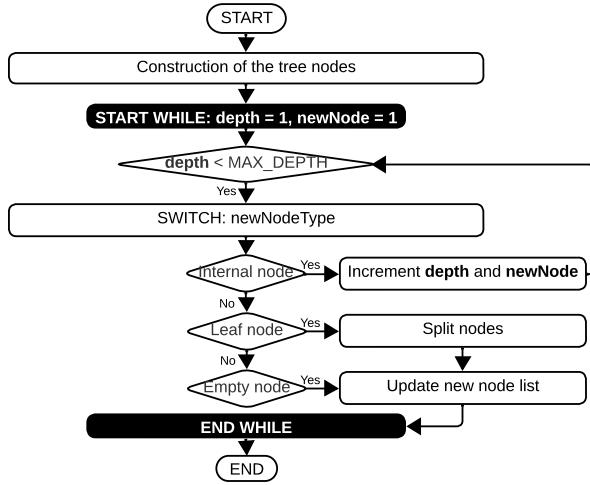


Figure 6: Flowchart of Quadtree updates

along the corners, which have three possible neighboring cells. Therefore, recursive processing is required to find all the neighboring nodes. In addition, since we are dealing with elongated bounding boxes, the extension to two dimensions requires extending the possible neighborhood not only to the nearest but also to the next nearest neighbors. Therefore, the algorithm changes significantly between 1D and 2D not only in the tree construction and the updating but also in the construction of the neighborhood list.

3.4. Large and wall particles

Up to now, we have discussed only particles of approximately the same size. In many DEM simulations, walls are needed which span the whole system size. Composing straight walls from smaller particles introduces additional normal directions where the particles are joined, which may lead to effectively uneven surfaces. Krijgsman et al. developed a neighborhood algorithm with hierarchical grid structures, based on the claim that tree codes could not be efficient enough in the case of larger size dispersion: “The tree data structure for contact detection does not allow to choose cell sizes at every level of hierarchy independently, therefore, leaving no room for optimization for various distribution of particle sizes [...]” [25]. Moreover, accessing neighbor sub-cubes in the tree is not straightforward since there can be nodes of different tree branches; no more details are given here since this method is not used any further. In contrast, our method allows partitioning larger particles (in particular walls that span the whole height or width of the system) into (partially overlapping) bounding boxes of the size of the granular parti-

cles as shown in Fig. 7. Therefore, size dispersion as such is not an issue for our type of implementation.

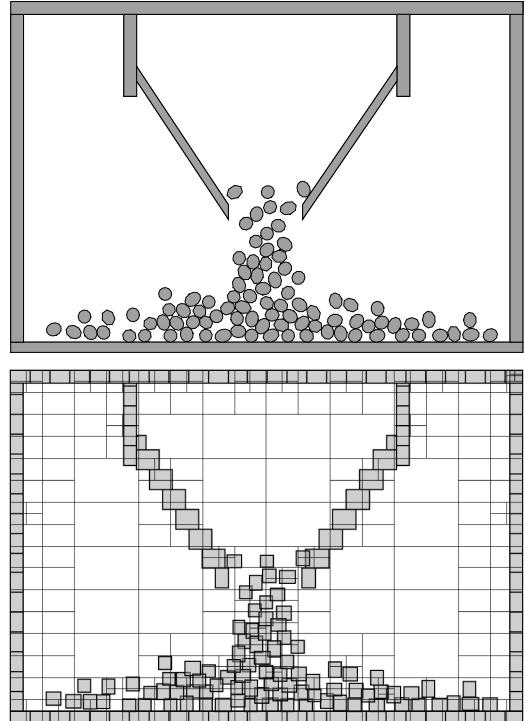


Figure 7: Actual DEM-configuration (above) and corresponding bounding boxes (some with particles, some empty) of varying size for the tree code (below).

4. Performance comparison between sort-and-sweep vs. quadtrees

We used MATLAB to implement and compare the performance of both neighborhood algorithms. MATLAB is an interpreter language but uses partially compiled functions to eliminate overhead. Additionally, inlining is possible, so that typical features of compiler languages can be reproduced. The simulation used mostly the two-dimensional polygonal DEM code from the monograph [1], but with the friction implementation of Krengel and Matuttis [26]. Only axis-aligned bounding boxes were treated instead of oriented bounding boxes, all computational tasks of dealing with oblique contacts were relegated to the overlap computation of the DEM. We evaluated the performance for 1000 to over 10000 bounding boxes with the machines in Tab. 1. As can be seen in Fig. 8, the program executed faster on the Xeon56 with a lower clock rate but DDR4-RAM, so we used that for further performance evaluation. The

388 number of cores in Tab. 1 is not relevant for the timings
 389 of the neighborhood routines we discuss in this article,
 390 because they are computationally cheap compared to the
 391 force computation. Nevertheless, to speed up the sim-
 392 ulation itself, we parallelized the force computation, as
 393 the largest simulations are still of the order of one week
 394 turnaround time for the largest system for single core
 395 execution.

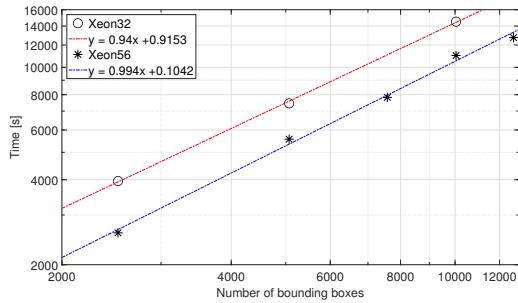


Figure 8: Execution times for the tree code run on our compute servers Xeon32 with 3.3 GHz and 512KB L1 cache and DDR3 memory versus Xeon56 with 2.4 GHz and 896KB L1 cache, which is faster despite the lower clock rate due to more advanced memory.

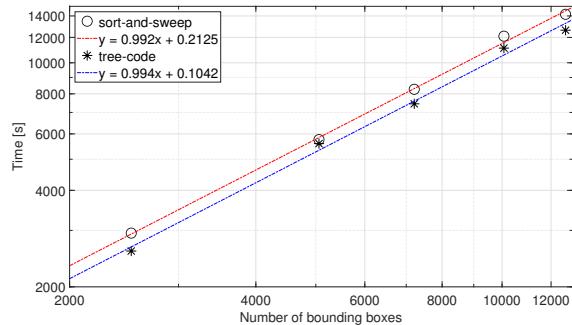


Figure 9: Time consumption of the neighborhood part of the DEM simulation executed on Xeon56. The relation is linear between time consumption and the number of particles, i.e. the complexity is $O(N)$. The tree code needs only about 90 % of the sort-and-sweep algorithm over the range of system sizes.

396 As during the research, more performance oriented
 397 Apple Silicon processors became available for the MAC
 398 mini, we added them in the comparison. Compared to
 399 the Intel processors, they run different cores simulta-
 400 neously on different clock rates, higher for the (“per-
 401 formance”) P-cores and lower for the (“economy”) E-
 402 cores, see Tab. 1. Level 1 caches for the Apple sili-
 403 con are smaller compared to the Xeon processors, while
 404 Level 2 caches are larger for the P-cores, but not for the
 405 E-cores. Level 3 caches on Apple silicon seem to exist,

406 but no specs are available.

407 4.1. Total CPU time

408 For drum systems like in Fig. 1, both algorithms take
 409 $O(N)$, but the tree code took about 90% of the CPU-time
 410 necessary for the sort-and-sweep approach, see Fig. 9.
 411 Admittedly, the rotating drum places the tree code at
 412 an advantage, as perpetually a change of bounding box
 413 coordinates along both dimensions is enforced for par-
 414 ticles which cannot have any interaction. For this type
 415 of system, the quadtree neighborhood algorithm is su-
 416 perior. Moreover, the tree updating costs only one-tenth
 417 of the time required for sort-and-sweep updating, see
 418 Fig. 10. This means that 80 % of the CPU-time in the
 419 treecode goes into the construction of the contact list,
 420 which is a double loop over the neighboring cells and
 421 would allow fine-grained parallelization. The sort-and-
 422 sweep algorithm allows only coarse-grained paralleliza-
 423 tion (independence for each direction) [6], all our at-
 424 tempts for fine-grained parallelization of the re-sorting
 425 with partitioned neighborhood lists failed, not only in
 426 MATLAB, but also in FORTRAN [27]. This means that
 427 the tree code in its scalar version is not only faster so
 428 the scalar portion reducing the parallelization efficiency
 429 according to Amdahl’s law will be smaller, and more
 430 processors can be used. It also has a larger fraction of
 431 code which itself allows fine-grained parallelization.

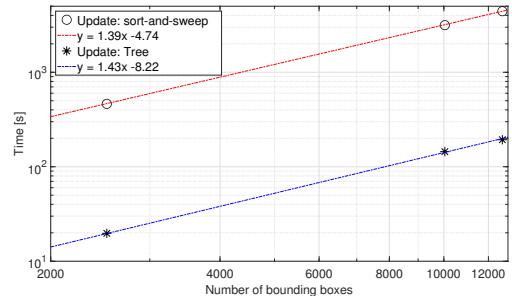


Figure 10: The updating part of the tree code (executed on Xeon56) takes only one-tenth of the time of that for the sort-and-sweep algorithm. In these data, the creation and updating of bounding boxes are not included, which leads to shorter times compared to Fig. 9.

432 4.2. Effects of cache-size on performance

433 For programs which use significant amounts of data,
 434 like computer simulations, not the CPU’s clock rate is
 435 the limitation, but the data transfer speed from and to
 436 memory, which is influenced by the cache size, the bus
 437 speed, the width of the cache line (the bus which trans-
 438 fers data between memory and CPU) and data loading

Table 1: Specification of the processors. The force computation of the polygonal simulation was computed via thread parallelization, while the evaluation of the neighborhood routines was without parallelization.

Processor	Intel® Xeon®		Apple M2		Apple M4	
	Xeon32 E5-2667v2	Xeon56 E5-2680v4	P-core	E-core	P-core	E-core
Clock [GHz]	3.3	2.4	3.5	2.4	4.4	2.6
# of Proc./Cores	2 / 32	2 / 56	1 / 4	/ 4	1 / 4	/ 6
Max. # of threads	64	112		8		10
Memory	DDR3	DDR4		LPDDR5		LPDDR5X
L1 cache [KB]	512	896	128	64	128	64
L2 cache [MB]	4	7	16 shared,	4 shared	16 shared,	4 shared
L3 cache [MB]	50	70				undisclosed

and storing strategies of the operating system. It is difficult to theoretically predict performance based on raw processor specifications, which on top of everything are very often not available either. When we compress the tree code running on a Xeon32 and a Xeon56 processor, see Fig. 8, surprising the processor with the faster clock rate takes longer, which reflects the difference in the memory technology (DDR3 vs. DDR4). This is a fair warning that even for the neighborhood algorithm alone, performance may be affected more by memory management than by the CPU clock rate.

5. Inlining

When sub-functions in computer programs are called, some time is lost when the data of the calling routine are “pushed onto the stack” (memory), and the CPU continues with only those data which belong to the function. When the program returns to the calling routine, its data have to be retrieved from the stack. Inlining is the concept of writing sub-functions literally into the calling routine, which has the advantage that the time loss for transferring the data to and from the stack can be avoided. The disadvantage is that now all data, those of the calling routine and that of the called sub-function, must be held available in the cache. When the amount of data is larger, this can lead to cache misses and slower execution times. In other words, there are two counteracting mechanisms, and a priori it is difficult to tell whether inlining has a positive effect on performance or not. In MATLAB (similar to many compiler languages via setting of optimization flags), there is a possibility for code inlining. While computers try to hold as much data in the cache memory as possible, at some critical amount of data which exceeds the cache size, swapping into the memory must occur. In Fig. 11, one can see that for a system size of about 5000 particles

(bounding boxes), the time consumption is above the fitted lines, and in that region, the crossover occurs where the tree code with inlining is more efficient than without inlining. For small systems, up to a few thousand particles, one doesn’t benefit from inlining (or is even conversely affected by its overhead). Gains from inlining start to appear instead only for large scale systems beyond 10000 particles. 5000 bounding boxes already translate into an amount of data of about 5000·8·4 kB. While this is only 1/6th of the cache size (of 896 kB), the actual number of nodes for the different system sizes are shown in Tab. 2. Even when the 87381 nodes are counted only as 4-Byte integers, the necessary memory, together with the data necessary for the bounding boxes, is close to the size of the cache. The induced cache misses for 5058 boxes lead to data points which are above the fitted curves for both the implementations with and without inlining in Fig. 11.

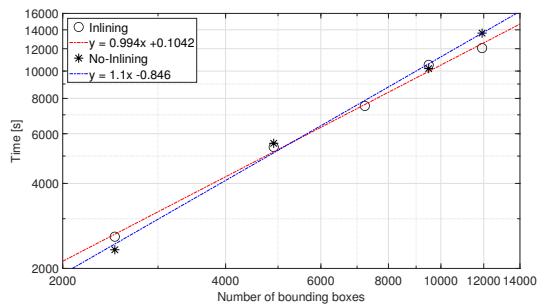


Figure 11: Execution times for the treecode with and without inlining (executed on Xeon56).

6. Comparison of cyclomatic complexity

Up to now, we have discussed only computational complexity, i.e. how the effort (in computer time or

Table 2: Number of bounding boxes, depth of the tree, and number of necessary nodes in the tree.

No. bound. boxes	Max. tree depth	Max. nodes
2522	9	87381
5058	9	87381
7599	10	349525
10014	10	349525
12651	10	349525

number of operations) scales in terms of the data dimensions, in our case the number of bounding boxes. Another issue of interest in the choice of algorithms is the complexity of the program structure. It has ramifications for the implementation, maintenance, modification, and extension of the code, and is commonly measured in terms of “cyclomatic complexity” [8]. Instead of the graph-theoretical intricacies of the original work, it is better to focus on “the amount of decision logic in a source code function” [9]. Each piece of code (main program, subroutine, function) starts at 1 and for each decision element and for each structure (`if`, `for`, `while`, …, as well as logical operators related to them (see e.g. [28]), the value is increased by 1. In MATLAB, it can be measured automatically by the `checkcode`-function. The cyclomatic complexity of our tree code with and without inlining, and for the sort-and-sweep approach, is shown in Tab. 3. As we have various nested loops and if-conditions in our neighborhood algorithms, the complexity is rather high. Inlining predictably increases the complexity, as all functionality from called functions is transferred into the calling function. A typical classification appraises cyclomatic complexity somehow like this [29]:

- 1-4: Low complexity
- 5-7: Moderate complexity.
- Acceptable, but not ideal.
- 8-10: High complexity.
- Should be refactored to make testing easier.
- 11-49: Very high complexity.
- Very difficult to test and maintain.
- Redesign and/or rewrite.
- ≤ 50: “Untestable” [28].

This classification aligns with pedagogical principles of introductory programming courses, which emphasize simplicity over performance. Such an approach is beneficial in general software engineering, such as application development, where the ability to handle diverse data types is critical, but the datasets are relatively small with fewer dependencies. In contrast, in scientific and

engineering programming, the situation markedly differs: there is typically a narrower range of data types, but the amount of data processed is significantly larger. In addition, computer simulations that address scientific and engineering challenges often require processing extensive datasets over extended periods of time, from days to weeks or even months. Given these differences, when analyzing DEM simulations, it is important to apply cautiously the classification based on cyclomatic complexity. A first program version of the tree code we had written with “low complexity functions” was consistently slower than the sort-and-sweep algorithm. The cyclomatic complexity of the final version of our tree code with and without inlining, and for the sort-and-sweep approach, is shown in Tab. 3. As we have various nested loops and if-conditions in our neighborhood algorithms, the complexity is rather high. For sort-and-sweep and for the tree code without inlining, the cyclomatic complexity is about 70. For the tree code with inlining the complexity is significantly higher at 273, as all functionality from called functions is transferred into the calling function. While general programming pedagogy would consider either approach untestable, performance considerations in DEM simulations necessitate the complexity of the code.

Table 3: Cyclomatic complexity of Tree codes compared to sort-and-sweep.

	sort & sweep	tree codes	
		Inlining	No inlining
Initialization	4	19	19
Updating data structure	64	45	25
Updating contact list	2	209	33
Total	70	273	77

7. Comparison between interpreted and compiled MATLAB code

For convenience, up to here we have used MATLAB-interpreter code for the profiling, based on previous experience that MATLAB interpreter code for the execution of linear algebra was not significantly slower than compiled code. This is due to MATLAB’s pre-compilation of blocks of code, together with internal functions which are pre-compiled and optimized for array-based operations. Nevertheless, during the

562 progress of this research, it turned out that there are sig-
 563 nificantly drawbacks in speed for the DEM-code, be-
 564 cause it is rich in if-conditions and other non-arithmetic
 565 operations. Accordingly, we refactored the simulation
 566 by eliminating “C-averse” MATLAB constructs, so that
 567 the code could be translated into C via the MATLAB’s
 568 “Coder”. This C-code was then compiled in MAT-
 569 LAB’s mex-command, which in this case for the imple-
 570 mentation on Apple silicon M4, used apples C-compiler
 571 with optimization option -O2. The object files are then
 572 linked into the MATLAB main program and executed
 573 together with the remaining MATLAB-code (for graph-
 574 ics etc.) via the MATLAB-GUI. As can be seen in
 575 Fig. 12, the compiled code was about one order of mag-
 576 nitude faster than the interpreted code. The speedup
 577 increased for increasing system sizes, which gives the
 578 impression that the compiled code is better at avoiding
 579 cache misses than the interpreted code. Reference runs
 580 with -O3 gave no significant changes.

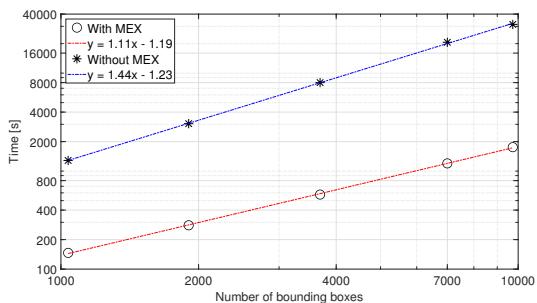


Figure 12: Comparison for the execution time for the compiled (“with MEX”) and the interpreted (“without MEX”) simulation, where the speedup varies from a factor about 8 for the smallest system to about a factor of 18 for the largest system. The benchmark was executed on an Apple M4 processor in MATLAB R2025b, which should have the most favorable speed for the interpreter code.

581 Performance data for Intel vs. the Apple silicon chips
 582 are given in Fig. 13. The M2 performs faster according
 583 to the proportion of its higher clock-rate for the P-cores,
 584 while the M4 performs nearly twice as fast relative to
 585 the clockrate, but drops a bit in performance for larger
 586 systems. This indicates a more complex internal archi-
 587 tecture, as the memory bandwidth (maximal 100 MB/s
 588 for M2 vs. 120 MB/s for M4) is comparable.

589 8. Conclusions

590 We have shown that neighborhood searches for DEM
 591 simulations can be performed not only with sort-and-
 592 sweep, but also with “minimal” tree codes in linear
 593 time indicating an $O(N)$ -complexity. For systems of

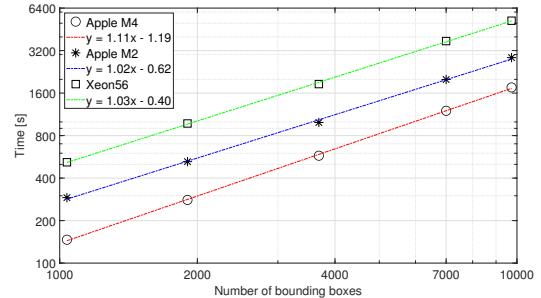


Figure 13: Execution times for the treecode for Xeon56 as well as M2 and M4 chip with MEX-files (compiled execution). The M2 faster by a factor of about 1.8 than the Xeon56, close to the relation of the clock rate (about 1.8) for the M2’s P-cores. The M4 is faster than the Xeon56 factor of about 3.5 for small systems and drops to 3 for large systems.

594 many moving particles, tree codes have an edge in speed
 595 in two dimensions, and also better prospects for par-
 596 allelization. For systems with large size dispersion,
 597 the larger particles have to be decomposed into smaller
 598 bounding boxes which are then treated individually in
 599 the tree. The relative performance of different system
 600 sizes and the algorithms did not change for the two mod-
 601 els used, so we assume the tendencies are also valid for
 602 newer memory types and processors. While we have
 603 tweaked the system geometry in a way that favors the
 604 tree codes, for three dimensions, the advantage can be
 605 even more marked, as the amount of irrelevant bound-
 606 ing box geometries in linear dimension increases. For
 607 large configurations of nearly static particles, the ad-
 608 vantages of tree codes become less marked. For DEM
 609 systems with a lot of motion, like granular gases, the
 610 improvement should be significant, as a lot of neigh-
 611 borhood bookkeeping faces a relatively small amount of
 612 interaction computations. For penetrating particles like
 613 those in SPH and MPS, there is so much variation in
 614 the neighborhood that neither the use of tree codes nor
 615 the sort-and-sweep algorithm can be advised. A further
 616 possible application is in adaptive meshing algorithms
 617 for the finite element method, as the neighborhoods
 618 are relatively limited. Apart from the use in structural
 619 mechanics [30], the geometric information contained in
 620 trees can also be used to help construct meshes for fluid
 621 dynamics with flows of large variation or meshes be-
 622 tween particles. Nevertheless, tree codes for neighbor-
 623 hood computation come at the cost of significant algo-
 624 rithmic complexity, beyond “untestable” according to
 625 the valuation criteria of computer science.

626 **Declaration of competing interests**

627 The authors declare that they have no known competing
628 financial interests or personal relationships that could
629 have appeared to influence the work reported in this pa-
630 per.

631 **Data availability**

632 An example implementation of the quadtree is available
633 at [repository]. The example simulates the movement
634 of ~ 1600 free polygonal particles in a rotating drum.
635 For easy visualisation, the program is implemented in
636 MATLAB.

637 **Acknowledgments**

638 D. Krengel and J. Chen would like to acknowledge
639 funding through a Grant-in-Aid for Scientific Research
640 (JP21H01422, JP21KK0071, and JP21K04265) from
641 the Japan Society for the Promotion of Science (JSPS).

642 **References**

- 643 [1] H.-G. Matuttis, J. Chen, Understanding the Dis-
644 crete Element Method: Simulation of Non-
645 Spherical Particles for Granular and Multi-body
646 Systems, 1st Edition, Wiley, Singapore, 2014.
- 647 [2] B. C. Vemuri, Y. Cao, L. Chen, Fast collision
648 detection algorithms with applications to particle
649 flow, Computer Graphics Forum 17 (2) (1998)
650 121–134. doi:10.1111/1467-8659.00233.
- 651 [3] B. Vemuri, L. Chen, L. Vu-Quoc, X. Zhang,
652 O. Walton, Efficient and accurate collision detec-
653 tion for granular flow simulation, Graphical Mod-
654 els and Image Processing 60 (6) (1998) 403–422.
655 doi:10.1006/gmip.1998.0479.
- 656 [4] P. A. Cundall, O. D. L. Strack, A discrete numer-
657 ical model for granular assemblies, Géotechnique
658 29 (1) (1979) 47–65.
- 659 [5] J. Chen, D. Krengel, D. Nishiura, M. Furuichi, H.-
660 G. Matuttis, A force–displacement relation based
661 on the jkr theory for dem simulations of adhesive
662 particles, Powder Technology 427 (2023) 118742.
663 doi:10.1016/j.powtec.2023.118742.

- 664 [6] J. Chen, H.-G. Matuttis, Optimization and
665 openmp parallelization of a discrete element
666 code for convex polyhedra on multi-core
667 machines, International Journal of Modern
668 Physics C 24 (02) (2013) 1350001.
669 doi:https://doi.org/10.1142/S0129183113500010.
- 670 [7] J.-F. Hake, W. Homberg, The impact of mem-
671 ory organization on the performance of matrix
672 calculations, Parallel Computing 17 (2) (1991)
673 311–327. doi:https://doi.org/10.1016/S0167-
674 8191(05)80116-4.
- 675 [8] T. J. McCabe, A complexity mea-
676 sure, IEEE Transactions on Software
677 Engineering SE-2 (4) (1976) 308–320.
678 doi:http://dx.doi.org/10.1109/TSE.1976.233837.
- 679 [9] A. H. Watson, T. J. McCabe, Structured testing:
680 a testing methodology using the cyclomatic com-
681 plexity metric, Tech. rep. (1996).
- 682 [10] J. J. Monaghan, Smoothed particle hy-
683 drodynamics, Annual Review of Astron-
684 omy and Astrophysics 30 (1992) 543–574.
685 doi:10.1146/annurev.aa.30.090192.002551.
- 686 [11] S. Koshizuka, Y. Oka, Moving-particle
687 semi-implicit method for fragmentation
688 of incompressible fluid, Nuclear Science
689 and Engineering 123 (1996) 421–434.
690 doi:https://doi.org/10.13182/NSE96-A24205.
- 691 [12] D. Krengel, J. Chen, Z. Yu, H.-G. Matuttis, T. Mat-
692 ushima, Implementing van der waals forces for
693 polytope dem particles, IOP Conference Series:
694 Earth and Environmental Science 1480 (1) (2025)
695 012030. doi:10.1088/1755-1315/1480/1/012030.
696 URL [https://doi.org/10.1088/1755-1315/
697 1480/1/012030](https://doi.org/10.1088/1755-1315/1480/1/012030)
- 698 [13] D. Baraff, Rigid body simulation, in: M. Kass
699 (Ed.), SIGGRAPH 93: An Introduction to phys-
700 ical Modeling, ACM, 1993, pp. H1–H68.
- 701 [14] J. D. Cohen, M. C. Lin, D. Manocha, M. Ponamgi,
702 I-COLLIDE: an interactive and exact collision de-
703 tection system for large-scale environments, in:
704 I3D ’95: Proceedings of the 1995 symposium on
705 Interactive 3D graphics, Association for Comput-
706 ing Machinery, New York, NY, USA, 1995, pp.
707 189–197.
- 708 [15] R. A. Finkel, J. L. Bentley, Quad trees a data struc-
709 ture for retrieval on composite keys, Acta Infor-
710 matica 4 (1) (1974) 1–9. doi:10.1007/bf00288933.

- 711 [16] J. L. Bentley, Multidimensional binary 756
 712 search trees used for associative search- 757
 713 ing, *Commun. ACM* 18 (1975) 509–517.
 714 doi:10.1145/361002.361007.
- 715 [17] A. W. Appel, An efficient program for many- 760
 716 body simulation, *SIAM Journal on Scientific 761
 717 and Statistical Computing* 6 (1) (1985) 85–103.
 718 doi:10.1137/0906008.
- 719 [18] J. Barnes, P. Hut, A hierarchical $O(n \log n)$ force- 764
 720 calculation algorithm, *Nature* 324 (6096) (1986) 765
 721 446–449. doi:10.1038/324446a0.
- 722 [19] M. C. Lin, Efficient collision detection for anima- 766
 723 tion and robotics, Ph.D. thesis, University of Cali- 767
 724 fornia, Berkeley, Berkeley, CA (1993).
- 725 [20] M. Wackenhorst, Fast algorithms to simulate ex- 768
 726 tremely polydisperse media, Ph.D. thesis, Univer- 769
 727 sity of Stuttgart, Department of Mathematics and 770
 728 Physics (2006).
- 729 [21] S. R. Schwartz, D. C. Richardson, P. Michel, An 771
 730 implementation of the soft-sphere discrete ele- 772
 731 ment method in a high-performance parallel grav- 773
 732 ity tree-code, *Granular Matter* 14 (3) (2012) 363–
 733 380. doi:10.1007/s10035-012-0346-z.
- 734 [22] J. G. Stadel, Cosmological n-body simulations and 774
 735 their analysis, Ph.D. thesis, University of Wash-
 736 ington (2001).
- 737 [23] D. Richardson, Direct large-scale n-body simu- 775
 738 lations of planetesimal dynamics, *Icarus* 143 (1)
 739 (2000) 45–59. doi:10.1006/icar.1999.6243.
- 740 [24] J. Duriez, C. Galusinsky, Level set representation 776
 741 on octree for granular material with arbitrary grain 777
 742 shape, in: *Topical Problems of Fluid Mechanics*
 743 2020, TPFM, Institute of Thermomechanics, AS
 744 CR, v.v.i., 2020, pp. 64–71.
- 745 [25] D. Krijgsman, V. Ogarko, S. Luding, Optimal 778
 746 parameters for a hierarchical grid data structure for
 747 contact detection in arbitrarily polydisperse par-
 748 ticle systems, *Computational Particle Mechanics*
 749 1 (3) (2014) 357–372. doi:10.1007/s40571-014-
 750 0020-9.
- 751 [26] D. Krengel, H.-G. Matuttis, Implementation 779
 752 of static friction for many-body problems 780
 753 in two dimensions, *Journal of the Physical 781
 754 Society of Japan* 87 (12) (2018) 124402.
 755 doi:https://doi.org/10.7566/JPSJ.87.124402.
- 756 [27] R. Tenhagen, H.-G. Matuttis, Parallelization of 782
 757 spherical particle systems, Tech. rep. (2013).
- 758 [28] TheMathworks, Measure code com- 783
 759 plexity using cyclomatic complexity,
https://www.mathworks.com/help/matlab/matlab_prog/measure-code-complexity-using-cyclomatic-complexity.html, last
 760 accessed: 2/2024 (Updated: Oct. 23, 2023).
- 761 [29] Incusdata, Cyclomatic complexity, <https://incusdata.com/blog/cyclomatic-complexity>, last accessed: 2/2024 (Revised: Jul. 19, 2023).
- 762 [30] C. Lee, S. Natarajan, Adaptive quadtree 784
 763 polygonal based edge-based smoothed fi- 785
 764 nite element method for quasi-incompressible 786
 765 hyperelastic solids, *Engineering Analysis with Boundary Elements* 155 (2023) 973–994.
 766 doi:<https://doi.org/10.1016/j.enganabound.2023.07.003>.