Institut für Thermische Strömungsmaschinen
Prof. Dr.-Ing. H.-J. Bauer, Ord.

Implementation of a Nearest-Neighbor Search for Particle Simulation in Sparse Computational Domains

Energietechnisches Praktikum
von
Jens Peifle, B.Sc.

Betreuer: **Marc Keller, M.Sc.**

Februar 2019

Karlsruher Institut für Technologie

Ich versichere, die Arbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die wörtlich oder inhaltlich übernommenen Stellen sind als solche kenntlich gemacht. Die Satzung des Karlsruher Instituts für Technologie (KIT) zur Sicherung guter wissenschaftlicher Praxis wurde von mir in der aktuellen Fassung beachtet.

Karlsruhe, February 6, 2019

Contents

1 Introduction

Turbofan engines are widely used in commercial aircraft. New, more stringent requirements with regards to emissions, fuel consumption and noise pollution are putting pressure on aircraft engine manufacturers to increase the efficiency of their products. From a technical point of view, there are two options: increase the thermal efficiency of the turbine stage or increase the bypass ratio of the engine. The bypass ratio is the amount of air that passes around the turbine core rather than through it. Generally, a higher bypass ratio results in a quieter, more efficient engine. To increase the bypass ratio, the diameter of the engine's fan blades must be increased. However, increasing the fan diameter leads to an increase in the circumferential velocity of the fan blade tips. At very high speeds, large increases in losses and noise emissions are observed.

In conventional turbofans, the fan and the low-pressure compressor and turbine which drives the fan are attached to single shaft. In this configuration, the speed of the compressor stage is limited by the blade tip speeds to the fan. With the addition of a gearbox between the fan and the compressor and turbine, the fan speed can be greatly reduced while the compressor and turbine can rotate much faster. Both components can therefore operate at their optimal speeds, greatly increasing efficiency and reducing noise.

However, geared turbofans are not without drawbacks. In addition to increased complexity and manufacturing cost, a significant amount of energy is lost as heat within the gearbox. The cooling and lubrication of the gearbox are key challenges. These functions are realized with oil jets arranged around the gears. The interaction between the oil jets and the gear surfaces determines the cooling and lubrication performance as well as the further propagation of the oil within the gearbox.

Therefore, this interaction is a current focus of research at the Institute of Thermal Turbomachinery at the Karlsruhe Institute of Technology. Experimental investigations in this area are difficult due to the small time scale and inaccessible location of the interactions. However, Computational Fluid Dynamics (CFD) methods offer possibilities for detailed investigation. One such method is Smoothed Particle Hydrodynamics (SPH), a particle-based method that is well-suited to modeling free surface flows and moving boundaries. SPH, like other approaches to fluid dynamics modeling, is very computationally expensive. It is desirable to reduce the required computation time as much as possible. For this purpose, this work examines an approach to increasing the computational efficiency of the SPH solver and therefore reducing the required computation time.

2 Motivation & Objective

2.1 Motivation

Smoothed-particle hydrodynamics is a computational method which can be used to simulate mechanics of solids and fluids. It is mesh-free and employs the Lagrangian approach, which makes it well-suited for complex problems with free surface flows and moving boundaries.

Compared to mesh-based methods, SPH requires a very large number of particles to ensure an equivalent resolution. However, in applications where there is relatively little high-density fluid (e.g. oil) in a computational space filled with low-density fluid (e.g. air), the low-density phase cam be completely omitted. This is results in a sparse computational domain, in which the amount of particles is small compared to the size of the domain.

During an SPH-method simulation, particles interact locally within a characteristic radius ("smoothing length"). In other words, each particle's behavior is influenced only by the particles surrounding it within a certain range. Therefore, for each particle p_i in the domain, all points within a certain cut-off radius r around the particle must be determined. This type of search is called *fixed-radius near neighbor* (frNN) search.

In the in-house code currently in use at the Institute for Thermal Turbomachinery, the fixed-radius near neighbors search is implemented using cell linked-lists (CLL, also cell lists, linked-cell method). In sparsely-filled computational domains as seen in simulations using the SPH method, other methods for frNN search may yield a performance advantage over the CLL method, in the form of reduced run time or memory use.

2.2 Problem

The fixed-radius near-neighbor search in this case is specified as follows:

For each point p_i within the computational domain, find all neighbors p_j that lie less than a cut-off radius $r = 3d_x$, away from the particle, where d_x is the mean particle spacing. This includes the particle $p_1$itself. The result is a set of interactions $p_i \leftrightarrow p_j$, where $p_j \leftrightarrow p_i$ is considered identical to $p_i \leftrightarrow p_j$ and is not repeated.

2.3 Objective

The objective of this work is to compare alternative methods for solving the described fixed-radius near neighbor search problem to the existing solution using CLL. For this purpose, a benchmarking framework is created in which the a comparison can take place independently of the rest of the SPH code. Alternative solutions are researched and the most promising methods are implemented within the framework along with the CLL method. The benchmarks measure process runtime and memory use, which are then used to compare the search methods.

3 Neighbor Search Methods

The following sections describe tools that can be used to solve the fixed-radius near neighbor search problem described in section 2.2. These are implemented in C++ 11 and outfitted with timing code in order to measure their execution times.

3.1 Cell Linked-Lists Method

The cell-linked lists method (also called linked-cell method or cell-lists method) divides the calculation domain into cells of edge lengths equal to or greater than the cutoff radius of the interaction search. Therefore, to find the neighbors of a particle, only the cells adjacent to the cell containing the particle must be searched. In the 3D case, the potential neighbors of a particle are found within the 27 cells directly surrounding the particle (Weygand (2018)).

The particles themselves are first sorted into two lists, $first$ and $next$. The list $first$ contains the indices of the first particle in the cell for each cell, i.e. $first[i]$ is the first particle in the i-th cell. The list *next* links a particle i to the index of the next particle j in the same cell, i.e. $next[i] = j$. If there are no further particles in the cell, *next* contains -1. In this manner, by starting with the first particle and following the links until the next index is -1, all particles in a cell can be visited in an efficient manner.

To create particle interaction lists for the full domain, the search loops through each cell pair. A cell pair is the current cell and itself, or the current cell and an adjacent cell. For each particle in the current cell, the distance to each particle in the other cell is calculated. If the distance is smaller than the cutoff radius, the interaction is added to the interaction pair lists. For increased efficiency, there is no need to check "backwards" into previous adjacent cells, as any interactions between the current cell and the previous cell have already been found.

3.2 The *ANN* Library

The ANN library (Mount und Arya (2010a)) implements k-nearest neighbor search using methods based on orthogonal space decomposition. The particles in the search domain are spatially sorted into special data structures. ANN supports kd-trees and box-decomposition trees (bd-trees). The bd-tree includes more decomposition methods than the kd-tree and is more robust for highly clustered data sets.

ANN supports exact and approximate nearest neighbor searching and a number and a number of distance metrics. In this case, the ANN library is used to perform exact fixed-radius NN searching using the Euclidean (L_2) distance metric. Exact searches can be performed by setting the tolerance ε to 0. For fixed-radius searching, ANN provides a procedure *annkFRSearch*, which returns the k closest points that lie within the radius bound. Because ANN statically allocates its arrays, the *annFRSearch* function must be called twice to find all points within the radius: once to find the number of points within the radius, and a second time to actually find the points.

The procedure as implemented in this work is therefore as follows: First, the search tree structure is initialized using the all data points (particles) in the search domain. Then the *annkFRSearch* procedure is called twice for each data point. Initially, a search is performed using *annkFRSearch* with $k = 0$ to find the number of points k' that lie within the radius bound. Then, the appropriate arrays of size k' are allocated, and filled by calling the procedure a second time with $k = k'$. At this point, the neighbors and therefore the interactions of the current point are known. These must then be inserted into the interaction lists, making sure not to duplicate any interactions.

If there is a known upper bound on the possible number of points within the search radius, it would be possible to pre-allocate arrays that are as large as the upper bound, and therefore need to perform the search only once. It is assumed that this would come at the cost of higher memory use, but it was not tested in the scope of this work. For more detailed information about the ANN library, see the ANN Programming Manual (Mount und Arya (2010b)).

3.3 The *nanoflann* Library

nanoflann is a header-only library for C++11 for KD-Tree structures. It does not support approximate nearest neighbors search. It includes a number of optimizations for increased efficiency. For instance, there are no provisions for choosing between or adding custom NN-search algorithms, and by using STL containers for the output data, there is no need to call the fixed-radius search method twice, as is the case with ANN (Blanco und Rai (2014)).

These optimizations could lead to performance advantages of *nanoflann* over the *ANN* library. It is also able to work with dynamic point clouds without rebuilding the entire kd-tree index. For these reasons, it could be promising for the final application and is mentioned here and included in the source files. However, a comparison to the ANN library is not included in this report.

4 Project Structure and Requirements

This chapter describes the structure of the project. First the overall organization and then the individual components are explained. Following that, the prerequisites and processes for building and/or running the code used in various parts of the project are briefly described.

4.1 Requirements

This project requires a Linux system with *gcc*, *bash*, and a recent version of Python 3 with the *numpy* package to run the benchmarks. Additional Python packages are required to view and execute the analysis notebooks. *Jupyter Notebook*, *matplotlib*, *pandas*, and *seaborne* can be installed using the python package installer *pip*. Note: Jupyter requires at least Python 3.3.

4.2 Directory Structure

Figure 4.1 shows the file structure of the project. First of all, the documentation (i.e. this pdf) is in the *docs* folder, and the latex source files are in *latex*. The *ann_1.2.2* directory contains the neighbor search C++ libraries that is examined. The *nanoflann* directory is an additional neighbor search library. The folders *src* and *bin* contain the C++ source files (described in Section 4.3) and their compiled binaries, respectively. The *scripts* directory is home to the test case generation scripts (described in Section 4.4) and test execution scripts (see Chapter 5). The *test* directory contains the working directories of the test runs and the result data. Finally, the folder *analysis* contains the iPython notebooks that were used in the examination and analysis of the results. These are described in more detail in Section 4.5.

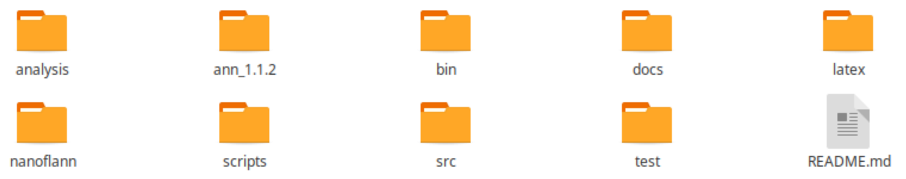

Figure 4.1: The root-level structure of the project repository.

4.3 Search Methods

The nearest neighbor search methods are implemented in C++ 11. The *src* directory contains the source files. The linked-cell method is implemented in *fr_cellLinkedList.cpp*. For the ANN method, there are multiple variants. First, *fr_ann_query.cpp* will only query a query single point for its neighbors. For the full nearest neighbor search, a method that builds the interaction pair lists (*fr_ann.cpp*) and a method that skips this step (*fr_ann_nolist.cpp*) are provided. This was necessary due to the extremely long time required for generating the interaction pair lists.

The *src* directory also contains a Makefile which can be used to build any or all of the source files. The resulting executable files are placed in the *bin* directory upon a successful build. Finally, source files for the Nanoflann library are provided (*fr_nanoflann.cpp* and *utils.h*) for possible future work evaluating this additional nearest neighbor search method.

4.4 Test Case Scripts

To be able to evaluate the performance of the different nearest neighbor search algorithms, a number of different test cases are required. These take the form of a list of data points which are processed by the search method executable. The test cases differ by the distribution of the data points in the search domain and are described in detail in Section 5.1.

To generate the test cases, different scripts are used. Python was chosen as the scripting language due to familiarity and the availability of easy-to-use plotting libraries for visualization of the data point distributions. In the *scripts* directory, test case scripts are provided for four different distribution types examined in this work. With the exception of *test3d_full*, which simply generates a filled domain, each of the scripts take certain parameters which control the fill and spacing of the points. Note that due to the method used to generate the distributions, the fill factor passed to the script does not specify the fill directly - some experimentation is necessary to get the desired fill.

The test case scripts are meant to be run within the benchmarking framework, as they also generate a statistics file for later use in the analysis. However, the scripts can also be run separately. Copy the script and the file *config.py* anywhere and execute the test case script with the Python interpreter. The data points are written to *data.pts* and can be visualized in 2D or 3D with scripts *plot_2d.py* and *plot_3d.py*.

4.5 Jupyter Notebooks for Analysis

To examine and compare the test results, Python is again used in the form of Jupyter Notebooks found in the *analysis* directory. To access the notebooks, make sure to have the appropriate packages installed (see Section 4.1). The Jupyter Notebook server can be started from the project root with the command *jupyter notebook* and notebooks are viewed and executed via a web browser.

The notebook files themselves are generally self explanatory. The overall structure of the analysis is as follows. In the data-preparation notebook, data is read from the results files in the *tests* directory, cleaned and labeled, and then written to a CSV file. The following notebooks read the nicely-formatted data from the CSV file and generate various tables and plots. In other words, if the data in the *tests* directory changes, the data-preparation notebook must be rerun in order to update the CSV file.

5 Test Cases & Benchmarking Methodology

This chapter first describes the test cases and shows how the data points are distributed for each case. Then, the benchmarking process and the tracked values are explained.

5.1 Test Cases

As in the SPH simulation, particles can be distributed within the domain in a variety of different ways. They can be arranged in many small clusters, or large groups. The cluster edges could be horizontal and vertical (i.e. parallel to the domain bounds) or at an angle. Depending on the contents of the simulation domain, there can be many points in the domain or only relatively few. The test cases are designed to cover these different varieties and allow for a comparison between them.

The test cases are differentiated by two important parameters, *fill* and *filltype*. The distribution and orientation of the points is determined by the fill type. Specifically, there are four different fill types: *full*, *clusters*, *corners*, and *diagonal*. Each are generated using the appropriate test case script as described in 4.4.

The second important parameter is the $fill$. This is defined as the fraction of actual points in the domain over the number of points in a fully-filled domain. The test case *full*, consisting of a fully-filled domain, has by definition a fill of 1.0. For a domain size of 0.005x0.01x0.005 and a particle spacing of 10^{-4}, the *full* test case contains 250000 points distributed on a regular grid. The number of points in the other test cases is therefore always a fraction of 250000.

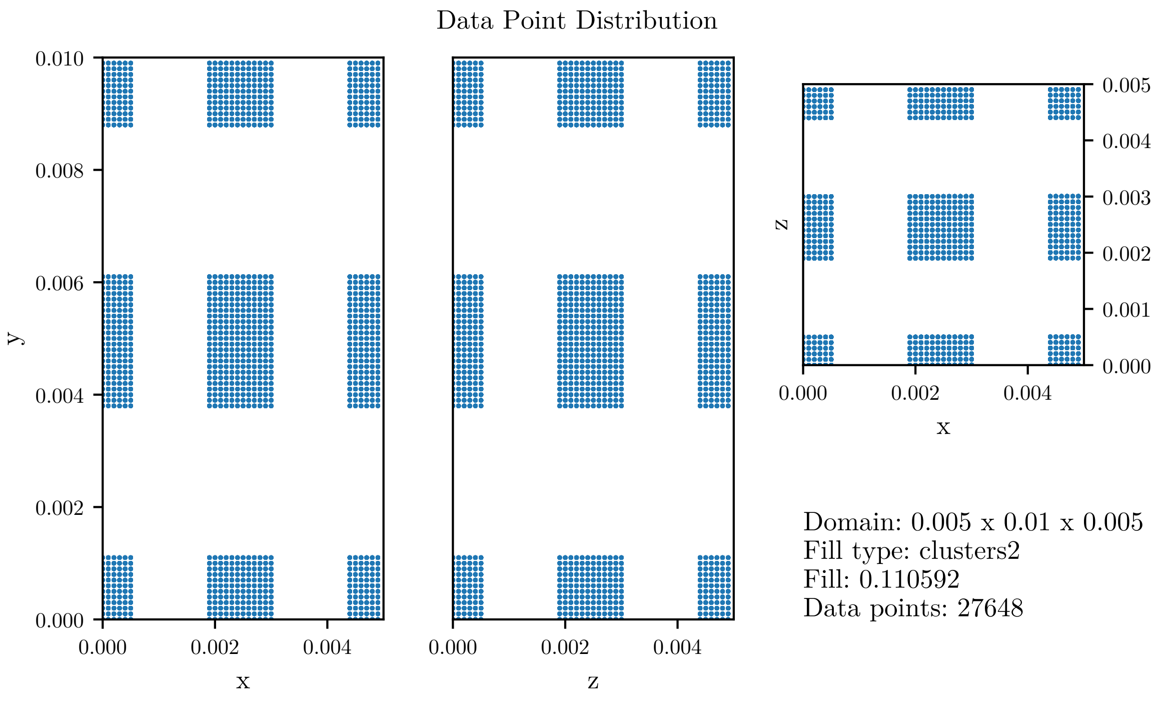

Figure 5.1: Data points distribution for the clusters2 test case with 11% fill.

The *clusters* test case is made up of rectangular, box-shaped clusters of points. Within the clusters, the points are distributed on a regular grid. The *clusters* cases are denoted *clustersN*, where the cluster size parameter *N* specifies the number of clusters distributed throughout the space, i.e. a higher value of *N* leads to more, but smaller clusters. The *clusters* test cases are examined for fill levels of 11% and 51% with varying cluster sizes. The cluster number parameter *N* is set as an argument to the test case generation script. Exemplary *clusters* test cases for the same fill but different *N* values can be see in the figures 5.1 and 5.2.

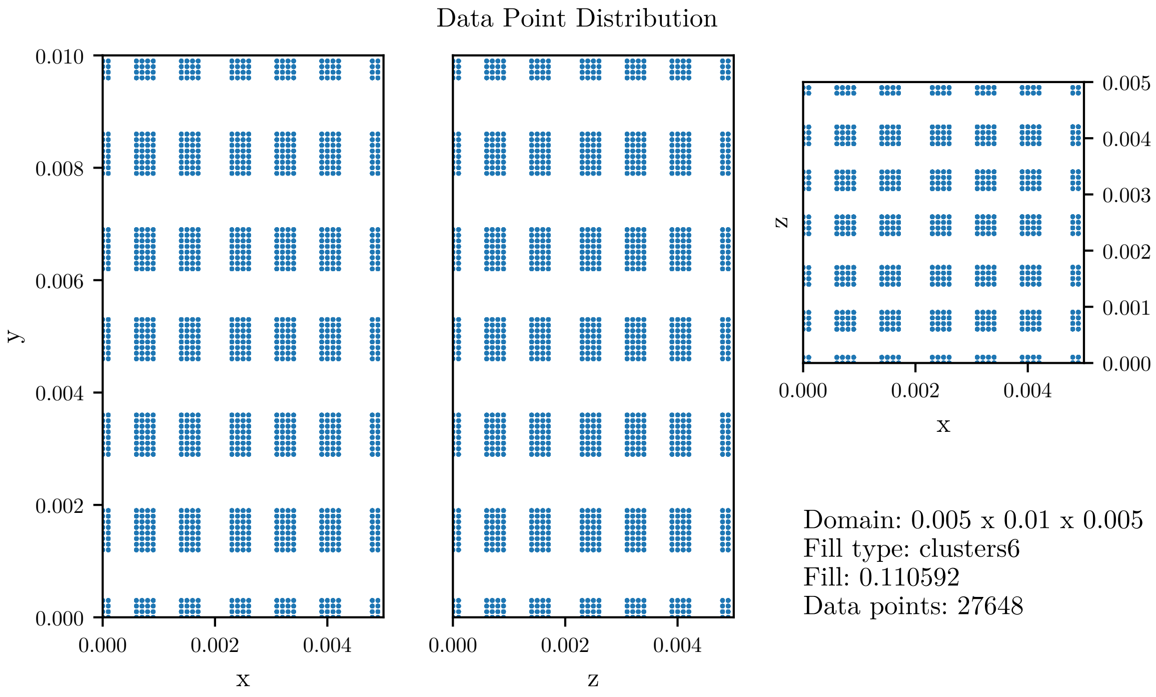

Figure 5.2: Data points distribution for the clusters6 test case with 11% fill.

The *corners* test case represent large groups of particles. The points are distributed regularly within two rectangular boxes extending from opposite corners of the domain towards the point in the center. This is visualized in Figure 5.3. The maximum possible fill with this fill type is achieved when both boxes extend from their corners to the center. In this case two quarters of the domain are filled, leading to a fill of 50%. The *corners* test case is examined for 2%, 11%, and 14% fill.

The *diagonals* test case is the only case in which the boundaries of the clusters are not parallel to the boundaries of the domain. The distribution is created by filling in the test domain from all eight corners up to a plane angled at 45 degrees from the domain boundaries. See Figure 5.4 for a 2D view of the *diagonals* test case with 11% fill. This distribution is however best viewed in 3D using the *plot_3d* script.

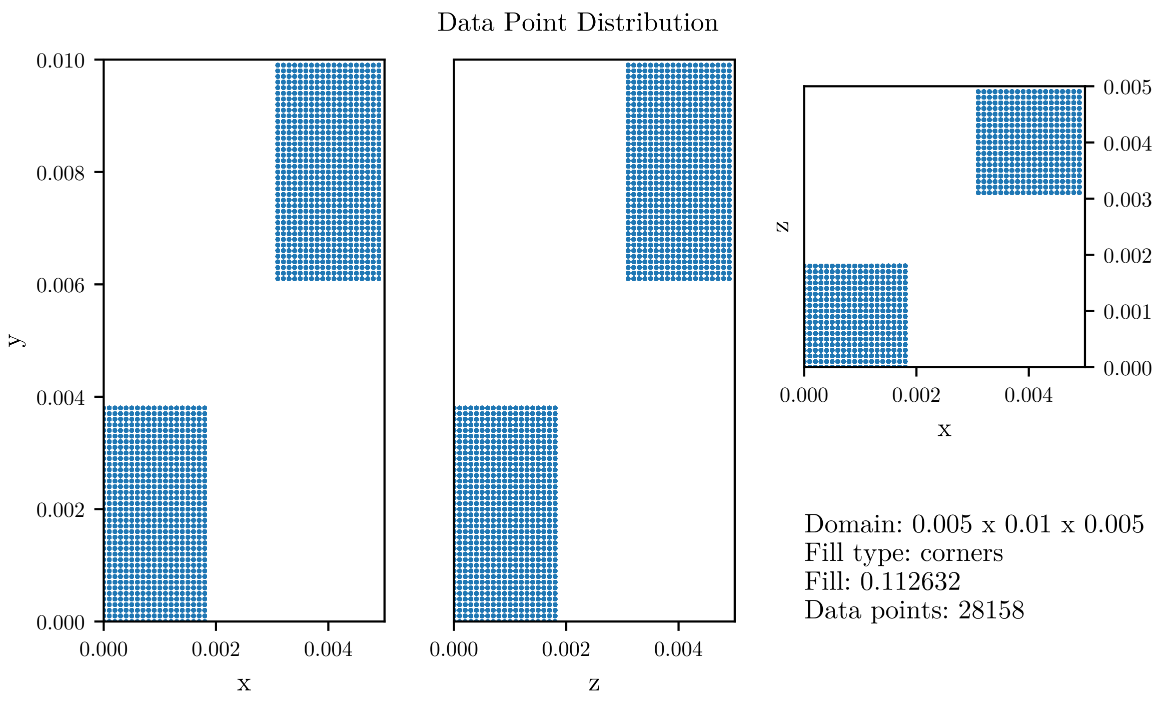

Figure 5.3: Data points distribution for the corners test case with 11% fill.

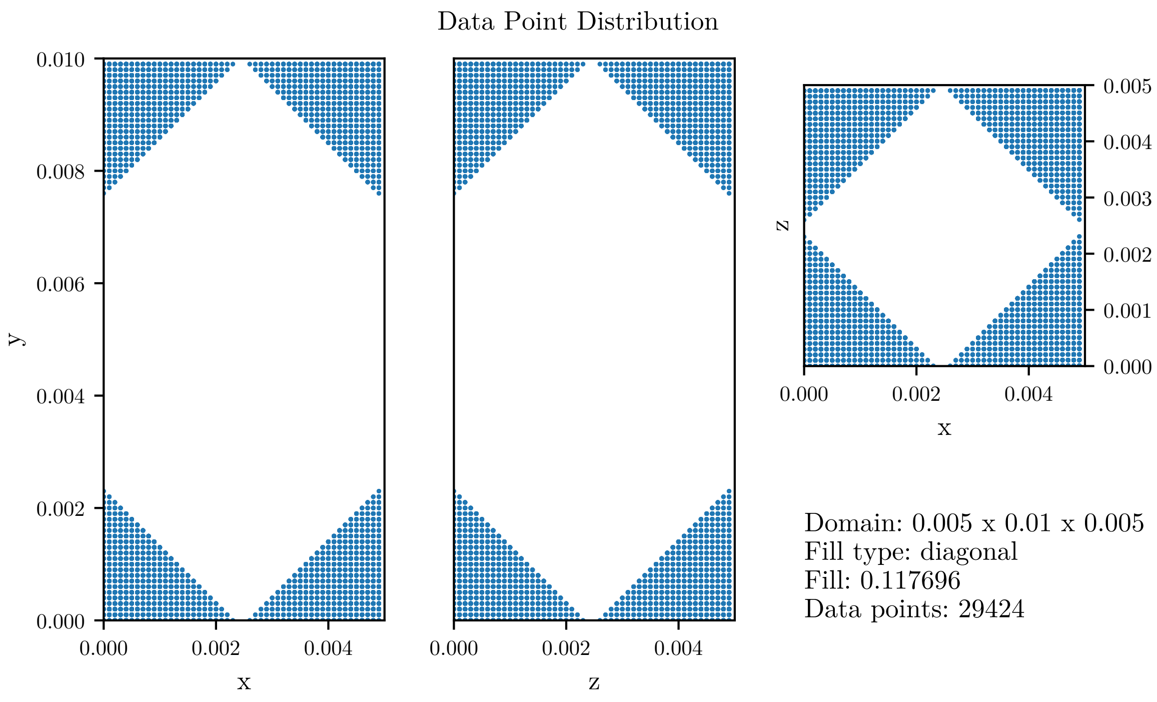

Figure 5.4: Data points distribution for the diagonal test case.

5.2 Benchmarking

5.2.1 Tracked values

In order to compare the performance of the search methods, two characteristic values are tracked. The first is the run time of the search: how much time it takes for the program to solve the frNN search problem and compile the results into a useful data format. This is achieved by using timers within the C++ implementation of the search methods. For all search methods, the total time, *ttotal*, is tracked. This total time does not include reading or writing data. For the ANN search, the time is additionally split up into the time spent determining the number of neighbors, *tksearch*, the time spent retrieving the neighbors from the kdtree structure, *tfrsearch*, and the time spent processing the interaction pair lists, *tprocessing*. The second tracked value is the memory use of the search method executable. This is measured by the *time* program, which outputs the maximum resident set size of the process. The resident set size is the space occupied by the process in the main memory (RAM).

5.2.2 Benchmarking Process

The benchmark is run via bash scripts for each search method, which define the test cases to run and also take care of tracking the memory use of the search processes. After the test case scripts and search method executables are copied to the test directory, the process as shown in Figure 5.5 is executed for each test case.

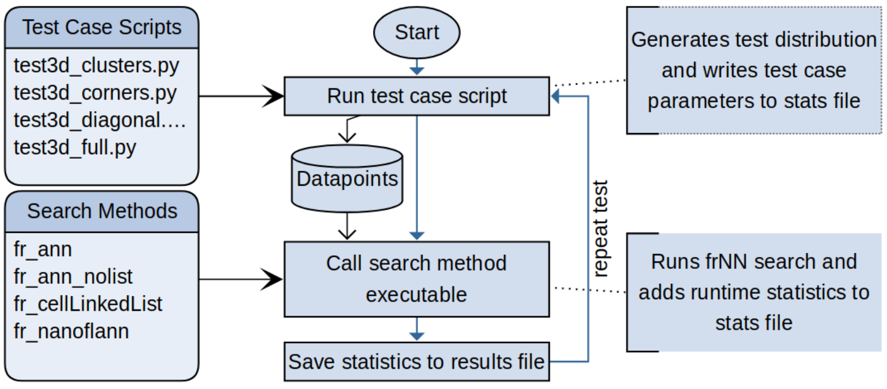

Figure 5.5: The program flow for a test run with a certain test case and search method.

First, the test case script is called, which writes the data points and statistics file. The search method executable ist then executed using the *time* command to track the memory use of the process. The search method executable adds its own time results to the statistics file and once it is finished the memory use results are also added. This is repeated multiple times to get a statistically valid sample. The script then moves on to the next test case, repeating the described process.

6 Results and Discussion

The results of the benchmarks were aggregated by mean over at least five repeated runs. The aggregated results can be found in Table 6.1 at the end of this chapter. Take special note of the extremely long runtimes for the ANN runs which include the interaction list processing. What follows is a graphical analysis of the benchmark results. First comes an overview of the results, followed by a more detailed breakdown of the search methods into the components and comparison over different fill types and fill percentages.

Figure 6.1 shows the total runtime of the frNN search over various fill types. Each color represents a different a search method. The cell linked-list method (*CLL*) is in blue, the ANN library with interaction list processing in green, and the ANN library but without list processing (ANN-NoList, or *ANN-NL*) in orange. Note that there are no results for ANN for some test cases as the processing times for these cases exceeded reasonable values. The scattering in the x-axis does not have a meaning and is only to prevent overlapping of data points.

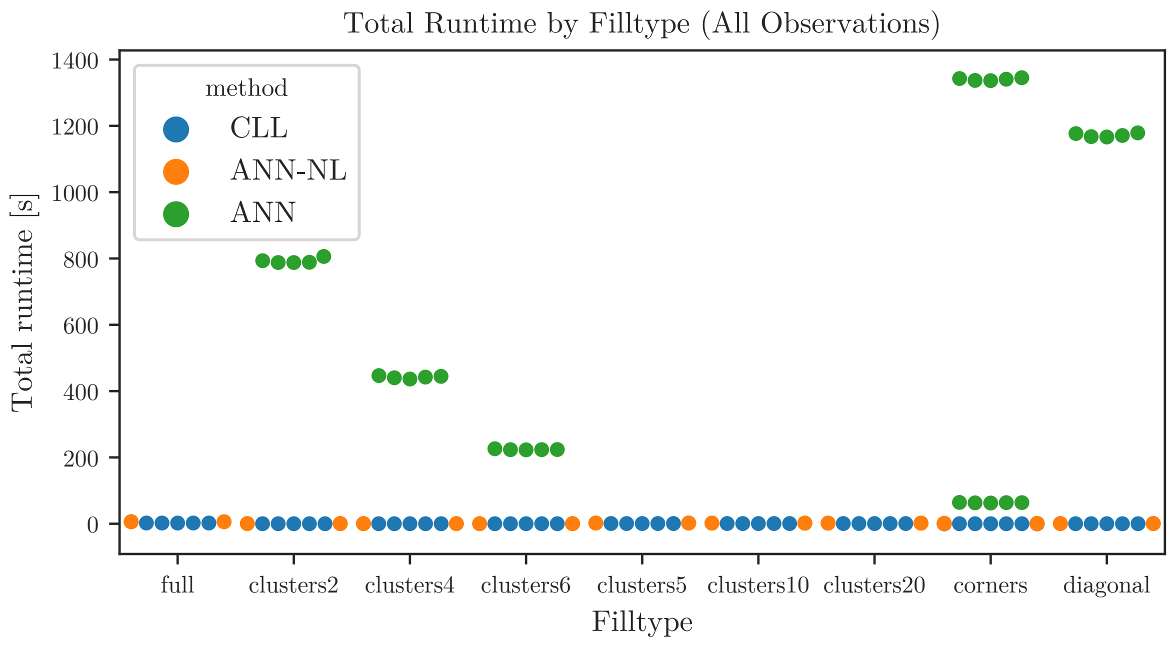

Figure 6.1: Mean total runtime grouped by fill type.

It is immediately apparent that the ANN runs take much longer than the other methods. In Figure 6.2, ANN has been removed from the plot and only the CLL and ANN-NL runs are shown. With a more reasonable y-axis scale, we see that runtimes for ANN-NL are still significantly longer than CLL runtimes in almost all cases. In other words, the CLL method is faster and outputs the results in the correct format, while the ANN search itself takes longer and does not include the processing step.

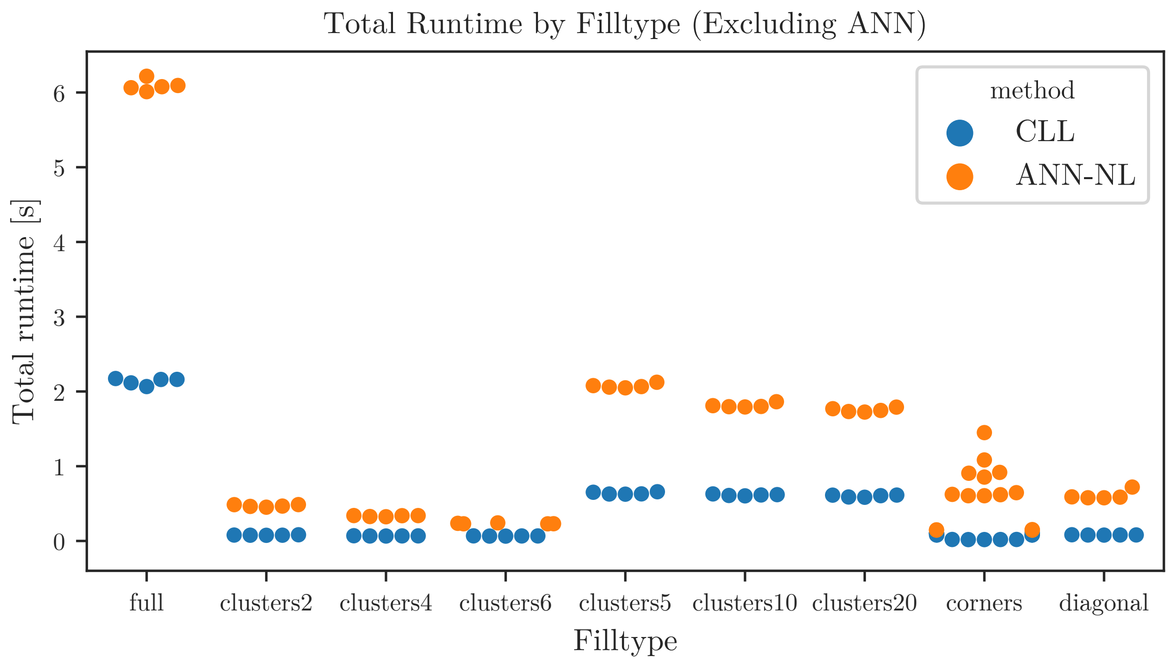

Figure 6.2: Total process runtime grouped by fill type, excluding ANN

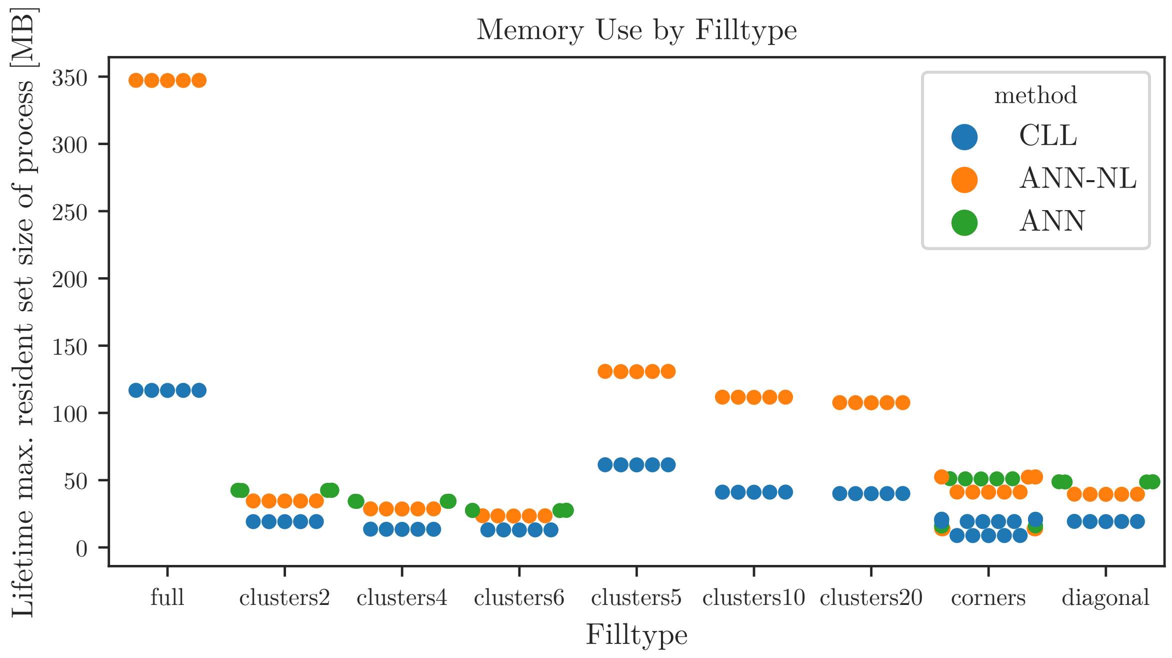

Figure 6.3: Mean maximum memory use grouped by filltype.

Comparing memory use in Figure 6.3, the ANN and ANN-NL methods are similar, with ANN-NL slightly below ANN. This is not surprising, as the memory requirements for the list generation are small compared to the kd-tree structure. However, the CLL method uses about one-half of the amount of memory that the ANN methods use.

Figure 6.4 makes the expense of the list generation very clear by splitting the total runtime of the ANN search method into the search time and the list processing time. The runtimes are grouped by fill type and are averaged over different fills. The search time is made up of the time to find the number of neighbors of each point (*tksearch*, blue) and the time to retrieve the neighbors from the data structure (*tfrsearch*, yellow). The list processing time (*tprocessing*) is shown in a separate chart. Note the different scales of the y-axes.

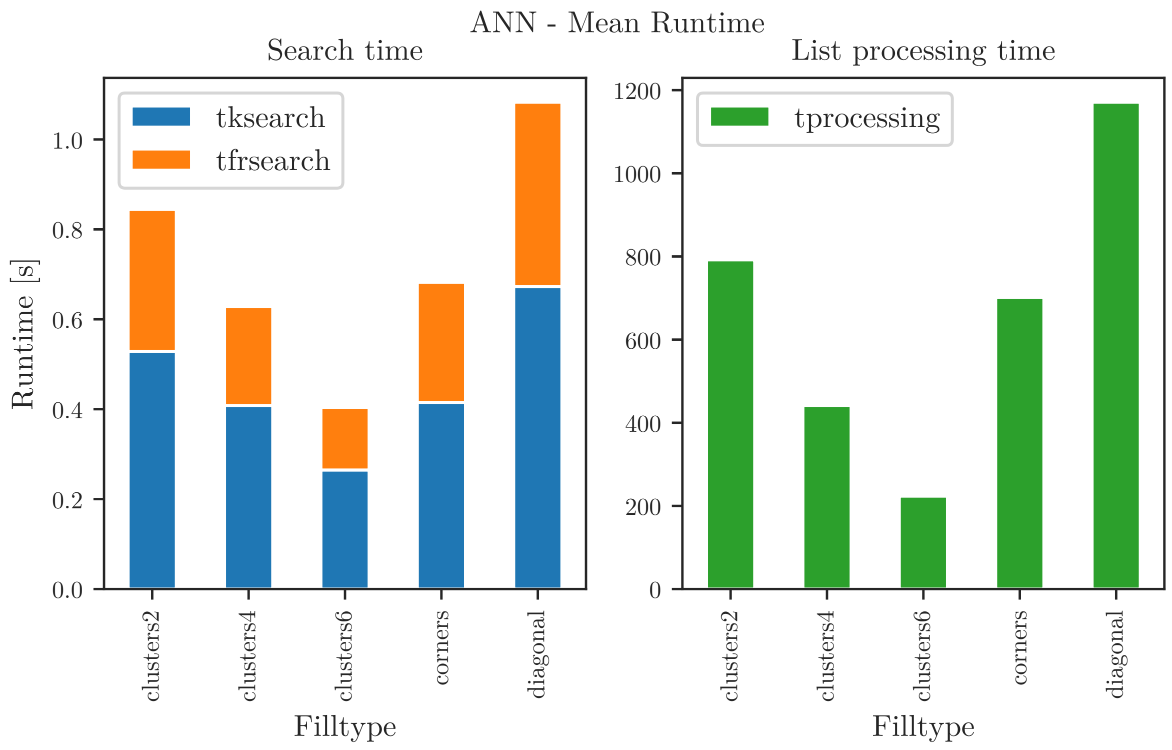

Figure 6.4: Total process runtime of the ANN search method split into search time (left) and interaction list processing time (right).

The search times lie in a range of about 0.4 to 1.2 seconds. About $\frac{2}{3}$ of the search time is spend determining the neighbors and $\frac{1}{3}$ of it is spent retrieving the neighbors. The list processing times range from 200 to 1200 seconds, i.e. generating the interaction pair lists is 100x more expensive than the search itself. Therefore, for the ANN search method to be at all reasonable, either a significant performance increase in the list processing or a different approach that does not require the interaction lists at all must be found.

6.1 Effect of Fill

The fill was expected to have a significant effect on the runtime and memory use. Figure 6.5 shows, for each search type, the runtimes sorted from left to right by increasing fill percentage. For the ANN and ANN-NL methods, the total time is split into the neighbor search, neighbor retrieval, and list processing times. Some bars are very small due to the y-axis scale. See Table

6.1 for the actual values. The runtimes for the test cases with 11% fill are also discussed in more detail in the next section.

For the ANN search type, some results are not available as these benchmarks did not complete in reasonable time. Furthermore, the search and retrieval times are so small compared to the list processing time that they are not visible in the bar chart. Here it is only apparent that the fill percentage has the largest overall effect on the runtime. Going from 2% to 11% fill leads in the best case to a 3.5-fold increase in the total runtime. In the worst case, the total runtime for 11% fill is 21 times that of the case with 2% fill. With 51% fill, the runtime exceeds 8 hours. Although not as extreme as the fill, the fill type also clearly has a large effect. This is discussed in more detail in Section 6.2.

For CLL and ANN-NL the total runtime also generally increases with increasing fill percentage. From 11% to 51% fill, the CLL method runtime increases about 7.8 times and the ANN-NL method increases 4.1 times. From 51% to 100% fill, the CLL method runtime increases about 3.5 times and the ANN-NL method increases 3.3 times. Therefore we could say that the CLL method is probably more sensitive to the fill percentage in sparsely filled domains ($<$ 50% fill) than the ANN-NL search. However, in all cases examined here, the CLL method was still significantly faster than ANN-NL even at very low fills. For example, at 2% fill, ANN-NL requires 0.15s while CLL requires only 0.02s and and already outputs the interaction pair lists as required by the current SPH implementation.

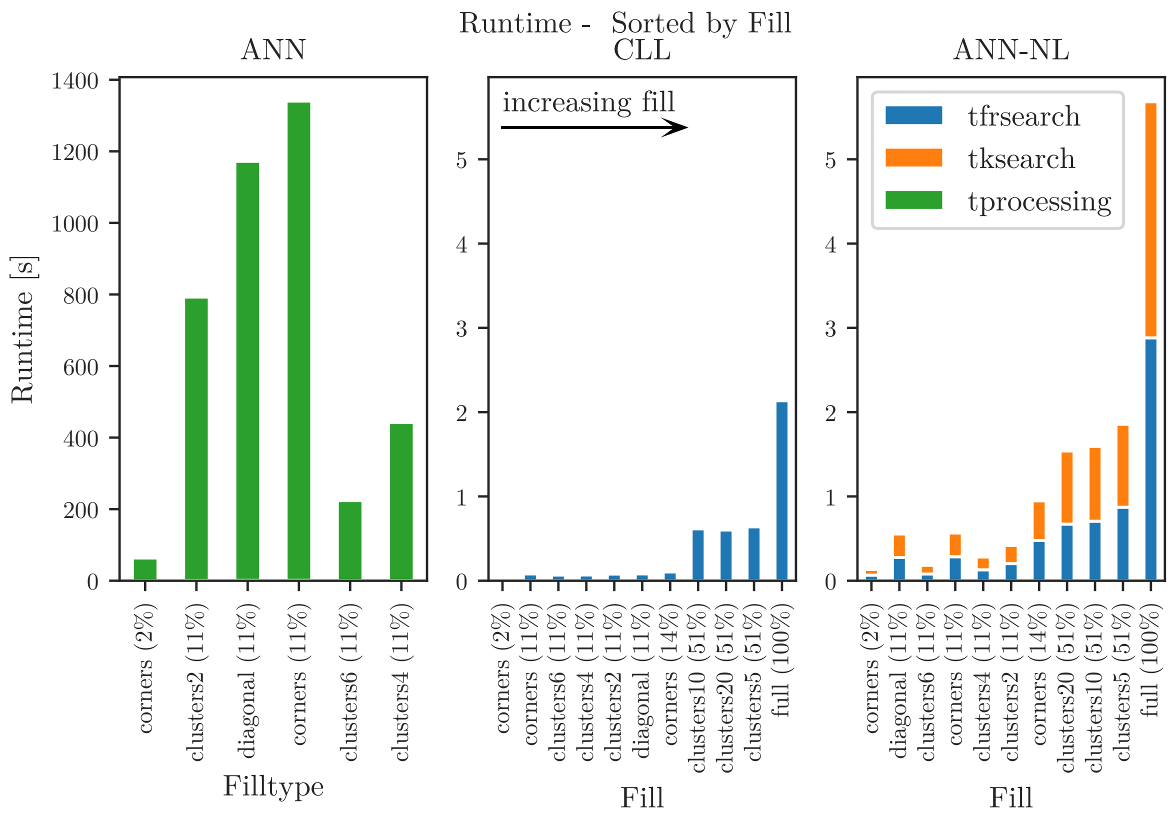

Figure 6.5: Total process runtimes sorted by fill percentage, for different search methods

Figure 6.6 shows the maximum memory use of the search processes for each search type, with the fill again increasing from left to right. The memory use generally shows the same trends as the runtime results. Here as well CLL has an advantage over the ANN and ANN-NL search methods. The difference between ANN and ANN-NL is nowhere near as great as in the runtime. ANN-NL unsurprisingly requires less memory than ANN, as it does not have to store the interaction lists. But its clear that the bottleneck in the list generation step for high fills is in processing power and not memory.

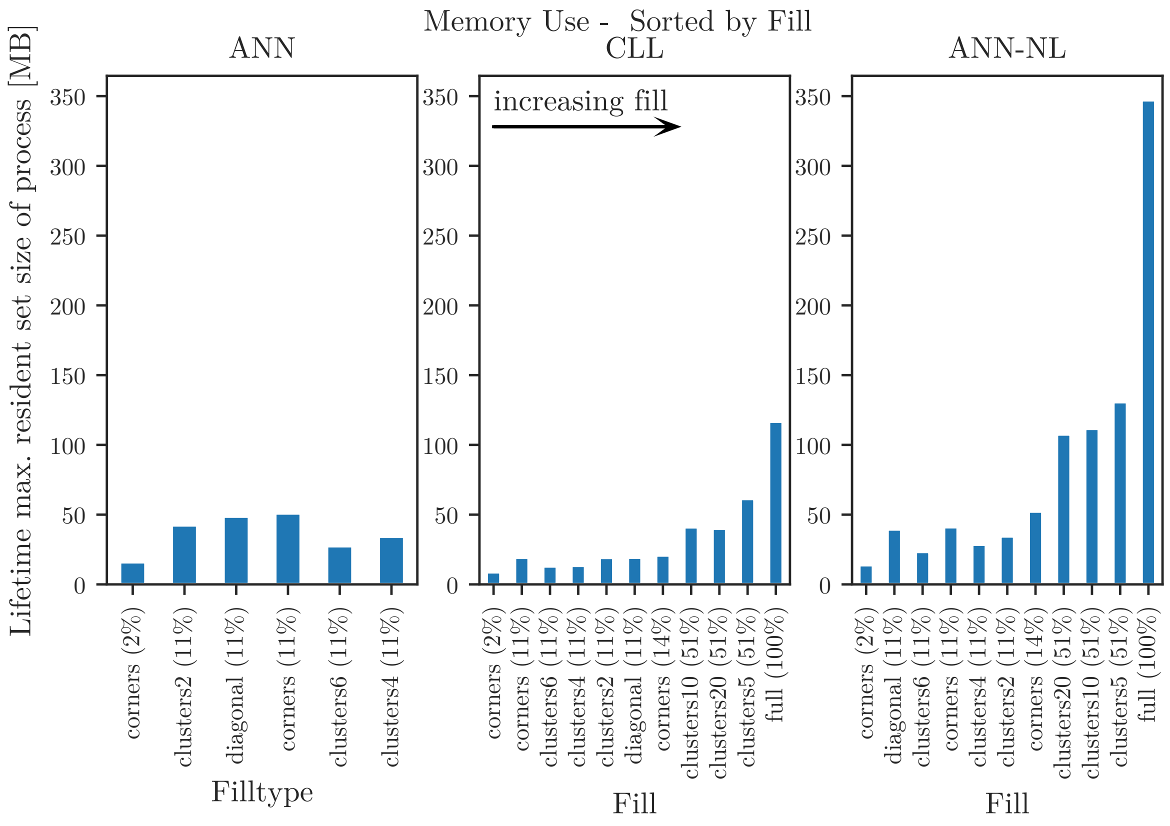

Figure 6.6: Maximum memory use sorted by fill percentage, for different search methods

6.2 Effect of Fill Type

This section discusses the effect of effect of various particle distributions. For example, there could be only a few larger clusters in the domain, or many small clusters. The effect of vertical/horizontal or angled cluster boundaries is also examined briefly. In the following, various fill types are compared for the same amount of particles in the domain. The test cases following in this section are made up of between 27648 and 29424 points in the domain, which equates to between 11.06 and 11.70 percent fill.

Figure 6.7 shows the total runtime for each filltype for each search method. Only the results with a fill of 11% are shown, and the total time is split into neighbor search, neighbor retrieval, and list processing time for ANN and ANN-NL search methods. The runtime on the y-axis of the figure is cut off at 1.5 seconds to facilitate comparison between the search methods. To see total runtimes for the ANN method refer back to Figure 6.5.

A note on the cluster sizes for the 5 fill types shown here: The *corners* fill type has the largest clusters of all fill types, as all points are arrayed in two clusters in opposite corners of the domain. The *diagonals* follows close behind, with the particles distributed in 8 clusters. Then, for the *clusters* cases, the *clusters2* fill type has the largest clusters, *clusters4* is in-between, and *clusters6* has the smallest clusters. Refer also to the figures in Section 5.1.

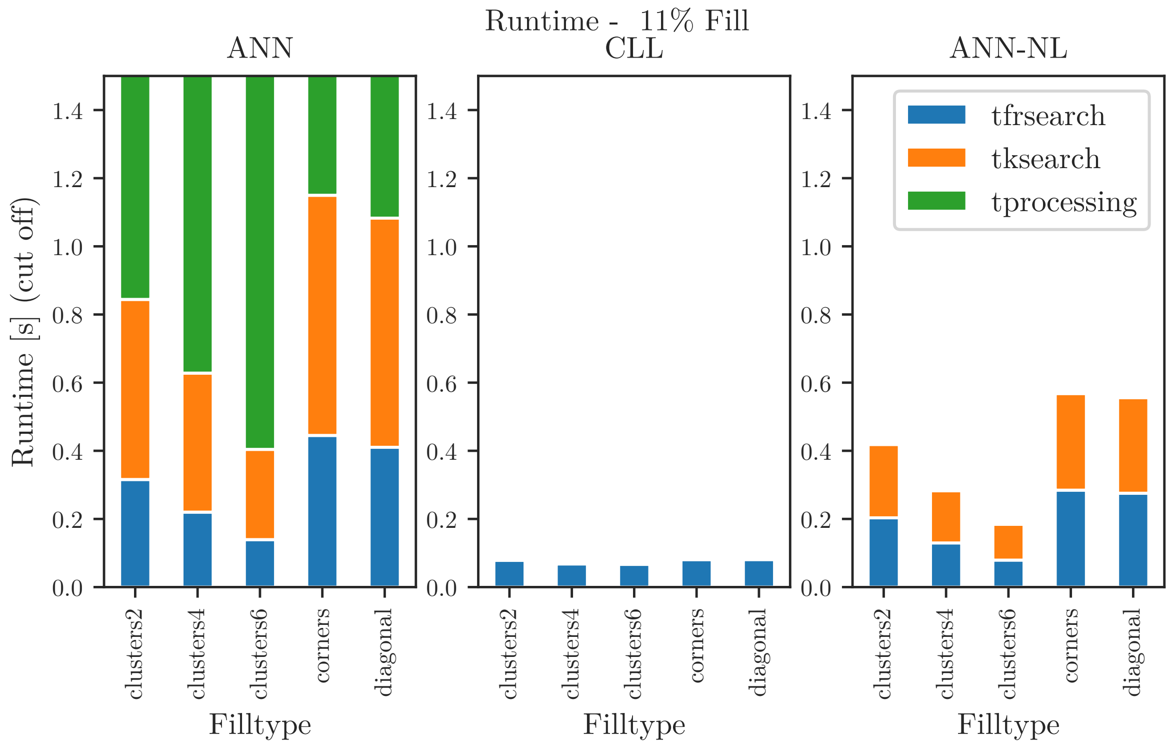

Figure 6.7: Total process runtimes for various fill types at 11% fill.

The total runtime of the CLL method is not significantly affected by the fill type. The results for the ANN methods do show an effect. It can be seen in the *clusters* cases that smaller clusters generally lead to a decrease in the runtime.

There is no significant difference between the *diagonal* and *corners* fill types. However, it was expected that the diagonal case would faster than the corners case, based on the difference in cluster sizes. So there could in fact be a negative impact on performance when the cluster boundaries are not orthogonal or parallel to the domain bounds. However, proving this definitively requires more test cases.

Interestingly, when ignoring the list processing time, ANN-NL method is significantly faster than the ANN method. This was not expected, as the only difference between the ANN and ANN-NL methods is the commenting-out of the source code for the list processing. Without closer examination, this is chalked up to some compiler optimizations or similar effects.

In Figure 6.8 the memory use of the search methods for each fill type for a fill of 11% are shown. The trend again is aligned closely with the results for the runtime, with smaller clusters reducing the memory required and little difference between the *diagonal* and *corners* fill types. ANN-NL requires a bit less memory than ANN, as expected. For the CLL search method, there is a small difference visible in the memory use for the smallest clusters (*clusters4* and *clusters6*).

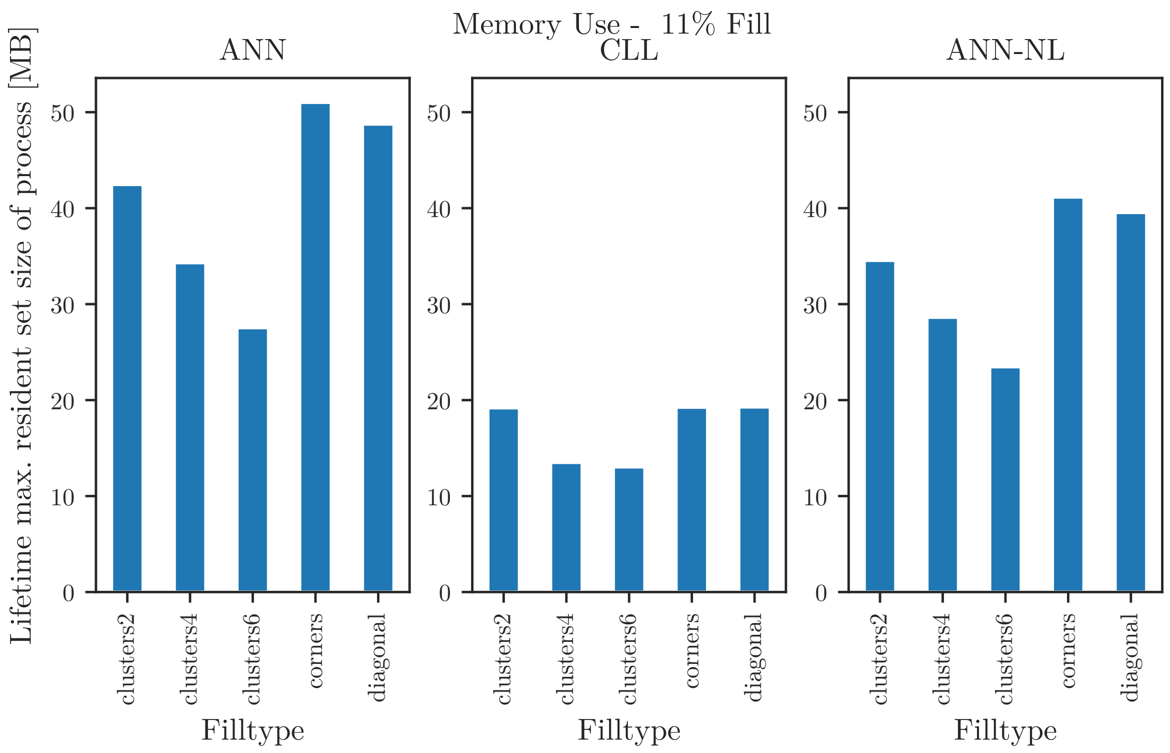

Figure 6.8: Maximum memory use for various fill types at 11% fill.

Table 6.1: All benchmarking results aggregated by mean.

fill	filltype	method	Datapts	Memory	$t_{frsearch}$	$t_{tksearch}$	$t_{processing}$	t_{total}
2	corners	ANN	7200	16.02	0.09	0.13	63.17	63.40
		ANN-NL	7200	13.93	0.07	0.07	0.00	0.15
		CLL	7200	8.84	0.00	0.00	0.00	0.02
11	clusters2	ANN	27648	42.46	0.32	0.53	791.79	792.70
		ANN-NL	27648	34.55	0.20	0.22	0.00	0.47
		CLL	27648	19.18	0.00	0.00	0.00	0.08
	clusters4	ANN	27648	34.30	0.22	0.41	441.36	442.04
		ANN-NL	27648	28.62	0.13	0.15	0.00	0.33
		CLL	27648	13.50	0.00	0.00	0.00	0.07
	clusters6	ANN	27648	27.53	0.14	0.26	223.46	223.92
		ANN-NL	27648	23.46	0.08	0.11	0.00	0.23
		CLL	27648	13.03	0.00	0.00	0.00	0.07
	corners	ANN	28158	51.01	0.45	0.70	1339.18	1340.39
		ANN-NL	28158	41.14	0.28	0.28	0.00	0.62
		CLL	28158	19.25	0.00	0.00	0.00	0.08
	diagonal	ANN	29424	48.75	0.41	0.67	1170.92	1172.07
		ANN-NL	29424	39.53	0.28	0.28	0.00	0.61
		CLL	29424	19.27	0.00	0.00	0.00	0.08
14	corners	ANN-NL	37044	52.37	0.48	0.47	0.00	1.04
		CLL	37044	20.88	0.00	0.00	0.00	0.10
51	clusters10	ANN-NL	128000	111.68	0.71	0.89	0.00	1.81
		CLL	128000	41.08	0.00	0.00	0.00	0.62
	clusters20	ANN-NL	128000	107.62	0.67	0.87	0.00	1.75
		CLL	128000	40.04	0.00	0.00	0.00	0.60
	clusters5	ANN-NL	128000	130.79	0.87	0.99	0.00	2.07
		CLL	128000	61.42	0.00	0.00	0.00	0.64
100	full	ANN-NL	250000	347.17	2.88	2.80	0.01	6.09
		CLL	250000	116.75	0.00	0.00	0.00	2.14

7 Summary and Outlook

Increasingly stringent efficiency and emissions requirements for commercial turbofan engines have spurred the development of geared engine designs. To master the challenge presented by these engines in the form of gearbox cooling and lubrication, accurate and efficient simulation tools are currently being developed and improved. The smoothed-particle hydrodynamics method (SPH) is one such computational fluid dynamics tool that is used to evaluate the flow and distribution of oil in such a gearbox. The goal of this work was to examine an approach to increasing the computational efficiency of an SPH solver.

As a particle-based method, SPH requires an interaction-search step, where the solver determines which particles in the computational domain interact with each other. The approach in this work targets this interaction search step, which is mathematically a fixed-radius near-neighbor search. Currently, the solver uses the cell linked-lists method. It was expected that in sparsely-filled computational domains with few particles, other methods would have a performance advantage over the CLL method in the form of reduced run time or memory use.

To examine this hypothesis, a benchmarking framework was developed in which different search methods can be evaluated without running the full SPH code. The tracked values include the runtime of the search process or individual sections of the process, and the maximum memory use of the process. A modular approach was chosen in which the test cases, search methods, and results evaluation code can be replaced as required.

The search methods are implemented in the same programming language as the SPH code, C++. They read in a list of particles representing a test case, and evaluate the neighbor search. The search methods include additional timing code to enable monitoring of the runtime. In addition to the CLL method, a C++ library called *ANN*, which implements a kd-tree data structure specifically for point neighbor searches, is evaluated. A third library, *nanoflann*, is mentioned but was not evaluated in detail due to time constraints.

Various test cases are generated using Python scripts. The test cases differ in two ways. First in the fill percentage, or how much of the domain contains particles. The second differentiating factor is the fill type, or how these points are distributed in the domain. For example, fewer but larger clusters of particles are expected to have different properties for neighbor searches then many clusters containing fewer particles. Clusters with boundaries that are not orthogonal or parallel to the edges of the domain are also tested.

After evaluating each of the test cases with the CLL and ANN search methods, it is immediately apparent that the ANN search method is much more computationally expensive than the CLL method in all test cases. This is due to the interaction pair list generation step, which must be completed separately when using ANN. Even with this step removed from the code (see ANN-NL results), the ANN method is significantly slower than the existing CLL method, which already includes the interaction list generation step. Futhermore, the CLL method uses about one-half of the amount of memory that the ANN method requires.

In addition to the comparison between the CLL and ANN methods. The effect of the parameters fill percentage and fill type were evaluated. Generally, processing time and memory use increases with the fill percentage due to the larger number of particles to be processed. The CLL method is not significantly influenced by the fill type. The ANN method is more sensitive and performs better (less memory and lower runtime) as the clusters get smaller. An influence of the cluster boundary orientation (orthogonal/parallel or diagonal) was seen, with diagonal boundaries leading to worse serach performance. However, due to the limited number of test cases this can not be definitively stated as fact without a more thorough examination.

Overall, the results show that the ANN search method approach does not yield any kind of performance improvement for the SPH simulation. The neighbor search itself is significantly slower than the CLL method. Even if the search itself were faster, the additionally required list processing step as implemented here is extremely expensive computationally. In the current version of the SPH solver, which requires the interaction pair lists, the ANN method is not usable.

In future work, the Nanoflann library should be evaluated more closely. If the Nanoflann library is faster than the ANN library, and if the SPH code can be refactored to make better use of the kd-tree data structure, performance improvements for sparse domains are still conceivable. Specifically, nanoflann theoretically supports updating parts of the search structure instead of rebuilding the entire structure. Alternative data storage methods such as bd-trees may also yield increased performance over kd-trees. For this and other alternative approaches, this work provides a good basis for further investigation.

Bibliography

Blanco, Jose Luis und Rai, Pranjal Kumar (2014): *nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees*. `https://github.com/jlblancoc/nanoflann`.

Mount, David M. und Arya, Sunil (2010a): *ANN: A Library for Approximate Nearest Neighbor Searching*. http://www.cs.umd.edu/ mount/ANN/.

Mount, David M. und Arya, Sunil (2010b): *ANN Programming Manual, V1.1.1*.

Weygand, Daniel (2018): *Lecture notes in Wissenschaftliches Progammieren fuer Ingenieure*.