

INFOB3CC: Assignment 3

Quickhull

Trevor L. McDonell & Ivo Gabe de Wolff & Tom Smeding

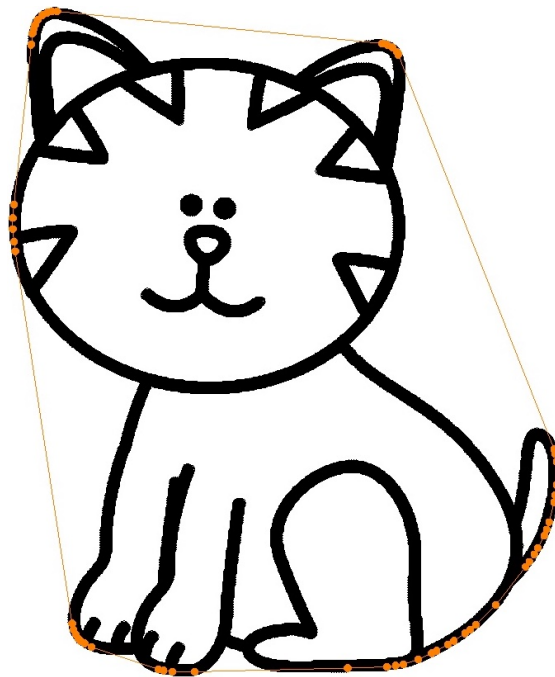
Deadline: Friday, 23 January 2026, 23:59

Change Log

2025-12-18: Initial release

Introduction

In this assignment you will design and implement a data-parallel version of *Quickhull*, an algorithm to compute the smallest convex polygon containing a given set of points. A shape is *convex* if it does not have any dents. Formally, for any two points on the shape, the line segment between those shapes must also be fully contained in the shape.



As you might guess by its name, the Quickhull algorithm has some similarities with Quicksort: it is a divide-and-conquer algorithm which partitions the input data and recurses on these sub-partitions. In this practical, instead of using recursion to process the sub-partitions in task-parallel, you will implement this algorithm in data-parallel, where all segments are handled at once. To facilitate this you will use *Accelerate*, an embedded language in Haskell for data-parallel array computations that can run on the CPU or GPU. We have discussed Accelerate in the lectures, and you can find more information about it in the library's reference documentation¹.

Starting Framework

A few remarks regarding the starting framework:

- Find the starting framework on the website:
`https://ics-websites.science.uu.nl/docs/vakken/b3cc/assessment.html`
Remember to run `cabal update`, otherwise you might not have the required dependencies.
- Installation instructions can be found on the website: `https://ics-websites.science.uu.nl/docs/vakken/b3cc/haskell-setup-instructions.html`
 - Note: It might be necessary to run `cabal build` once before the Haskell language server (IDE integration) starts working in VSCode.
- The starting framework builds two executables from your code: `quickhull` and `quickhull-test`. Both executables can take a number of command-line flags.
 - Without flags, `quickhull` will open a graphical window in which you can visually step through the algorithm, using your implementation. (Use the arrow keys, and escape to close.) By default, a random input is generated; use the `--random` and `--seed` flags (see `cabal run quickhull -- --help`) to customise the randomiser. With the `--file` flag you can load one of the data files in `input/`.
 - With `--benchmark`, `quickhull` runs in benchmark mode. It still accepts `--random`, `--seed` and `--file` as above.
 - `quickhull-test` is a small test suite that tests your individual functions against simple, sequential reference implementations. To run a subset of the tests you can use the `--pattern` flag, for example:

```
cabal run quickhull-test -- --pattern 'segmented'
```

To run one of these executables, use e.g. `cabal run quickhull-test`. To pass arguments, put them after `--`; for example:

```
cabal run quickhull -- --benchmark --file input/10.dat
```

¹<https://ics-websites.science.uu.nl/docs/vakken/b3cc/resources/acc-head-docs/Data-Array-Accelerate.html> (press s to search)

will benchmark your code on the cat input.

For Windows users: if `cabal` seems to hang after the actual program is already finished, kill it using Task Manager and use the provided `build_and_run.cmd` and `build_and_test.cmd` scripts instead. For usage info, open the scripts in your text editor; they contain a `REM` block explaining what they do.

- Use the benchmark to gauge the performance of your solution and the speedup you achieve as you add more threads. These are the results from running a model solution on a 4-core laptop (Intel i5-8259U):

```
$ cabal run quickhull -- --benchmark --random 100000 --seed 1234
benchmarking quickhull
time                6.920 ms    (6.679 ms .. 7.240 ms)
                    0.992 R2    (0.986 R2 .. 0.998 R2)
mean                9.471 ms    (8.862 ms .. 10.89 ms)
std dev             2.536 ms    (1.766 ms .. 3.672 ms)
variance introduced by outliers: 90% (severely inflated)
```

When using the scripts on Windows, use this command instead:

```
.\build_and_run --benchmark --random 100000 --seed 1234
```

On Linux and Mac, it is possible to control the number of threads used by the CPU backend as follows:

```
ACCELERATE_LLVM_NATIVE_THREADS=2 cabal run quickhull -- ...
```

Be sure to check your task manager to see if it worked. This method unfortunately does not work on Windows.²

Data representation

The algorithm of the starting template uses a head-flags array to distinguish the different sections of the hull. It has the type:

```
type SegmentedPoints = (Vector Bool, Vector Point)
```

The two arrays are always of the same length. A flag value of `True` indicates that the corresponding point in the points vector is definitely on the convex hull. Let that point be called p_1 . Let the next (left-to-right) point whose flag value is `True` be called p_2 . We say that all of the points between p_1 and p_2 are in the same *segment*.

The elements whose head flag is `True` form the boundaries of the segments. These elements are known to be on the convex hull, and for the other elements the algorithm hasn't decided yet. The algorithm will proceed by repeatedly choosing one element per segment which is on the convex hull, removing elements which definitely do not belong to the convex hull, and partitioning the other elements to form new, smaller segments.

²If you have Git installed, you *may* be able to do this if you prefix the command with `env.exe`.

Initial partition

The main part of the algorithm always works on a segment of the array, formed by a line and the points on one side of that line. This is encoded in the type `SegmentedPoints` as described above. To start the algorithm, we must define the two initial segments by choosing a line (p_1, p_2) and splitting the remaining points into a set of points above this line, and a set of points below it. The output of this step be arrays of the form:

```
flags = [True, False, False, ..., True, False, False, ..., True]
points = [p1, a1, a2, ..., p2, b1, b2, ..., p1 ]
```

Where a_1, a_2, \dots , are the points above the line (p_1, p_2) , and b_1, b_2, \dots , are those points below it. Placing the point p_1 again at the end will make parts of the rest of the implementation more convenient, but must be removed at the end. Some general hints:

- We must choose points p_1 and p_2 which are definitely on the convex hull.
- Points which lie *on* the line (p_1, p_2) should not be placed in either partition, as they are not part of the convex hull. One consequence of this is that the size of the output arrays can not be determined directly from the size of the input. Make sure that the middle point(s) along a line can not be chosen as the points p_1 or p_2 , nor as the furthest point.
- Do not physically split the input into separate segments which are then concatenated, for example by using `filter` and `(++)`. This approach will not scale to the recursive step of the algorithm (more generally, consider how could you efficiently concatenate thousands of array segments?) and thus will earn you no marks.

Partition

After the initial partition is created, the `partition` step is executed repeatedly on the segmented representation until no undecided points remain. Each segment of the input consists of a different line segment. The process is similar to what was performed in the initial partitioning step, however now we need to perform the process over all line segments at once. Thus, instead of regular functions such as `scanl1` which operate over the entire array, we must use segmented versions which operate on each segment of a segmented array.

General remarks

- Make sure your program compiles using `cabal build`.
- The documentation for Accelerate on Hackage is somewhat outdated; reference documentation for the version used by the Quickhull template can be found here: <https://ics-websites.science.uu.nl/docs/vakken/b3cc/resources/acc-head-docs/Data-Array-Accelerate.html>

- It is not allowed to concatenate arrays, for example with `(++)`. You must process all line segments at once in data-parallel!
- The various `scan*` and `permute` operators are very useful for this practical, pay attention to what they do.
- Include *useful* comments in your code. Do not paraphrase code but describe the structure of your program, special cases, preconditions, et cetera.
- Try to write readable and idiomatic code. Style influences the grade! For example, use indentation of 2 spaces. If you prefer an automatic formatter, you could use e.g. `ormolu`.³
- Efficiency (speed) of your implementation influences the grade.
- Copying solutions—from other people, the internet, LLMs like GitHub Copilot, or elsewhere—is not allowed.
- You cannot use external packages other than the dependencies which are already included in the template.
- For troubleshooting if things don't work, double-check if you have followed all the relevant instructions on the setup page⁴, and that your problem is not listed there in the Troubleshooting section. When in doubt, ask in a working group session or ask the teachers.

Submission

- This assignment may be submitted individually or in pairs.
- The deadline for this assignment is **Friday, 23 January 2026, 23:59**.
- Submission is via Brightspace; join one of the P3 groups and submit `src/Quickhull.hs` via Brightspace. As before, *only change* the file `src/Quickhull.hs`, otherwise the graders will not be able to compile your code.

Grading

1. (0.5 pt) Implement the functions `shiftHeadFlagsL` and `shiftHeadFlagsR`, which given an array shifts the values one element to the left or right respectively.
2. (1 pt) Implement the functions `segmentedScanl1` and `segmentedScanr1`, which are segmented variants of the inclusive scan operators `scanl1` and `scanr1`, respectively. These operators use a head flags array to indicate (by the value `True`) where each new segment should begin. You can assume that the first value in the flags array (left or right-most element, respectively) is `True`.

³<https://hackage.haskell.org/package/ormolu>

⁴<https://ics-websites.science.uu.nl/docs/vakken/b3cc/haskell-setup-instructions.html>

3. (0.5 pt) Implement the functions `propagateL` and `propagateR`, which propagate (copy) the value whose corresponding head flag is `True` to respectively the *right* or the *left*, until another head flag with value `True` is encountered.
4. (3 pt) Implement the function `initialPartition`.
5. (4 pt) Implement the function `partition`.
6. (1 pt) Finally, complete the algorithm by implementing the function `quickhull`.

Help/troubleshooting

- Always double-check the setup instructions on the website, including the Troubleshooting section at the bottom, and force-reload the page to ensure you have the latest one:
<https://ics-websites.science.uu.nl/docs/vakken/b3cc/haskell-setup-instructions.html>
- When running your program, you may get output like this intermixed with the normal output of your program (e.g. the test suite):

```
Creating library C:\Users\name\AppData\Local\accelerate\accelerate-llvm-1.3.0.0\accelerate-llvm-native-1.3.0.0\llvmp-19.1.0\x86_64-pc-windows-msvc\alderlake\rel\meep834f39da884b09540be452029780f0afde075050a932683f4b66c6e44ca61b04.lib and object C:\Users\name\AppData\Local\accelerate\accelerate-llvm-1.3.0.0\accelerate-llvm-native-1.3.0.0\llvmp-19.1.0\x86_64-pc-windows-msvc\alderlake\rel\meep834f39da884b09540be452029780f0afde075050a932683f4b66c6e44ca61b04.exp
```

This is output produced by `clang` when compiling the kernels resulting from your Accelerate code. Unfortunately we cannot disable this output. However, compiled kernels are cached, so if you run e.g. the test suite a second time, you shouldn't get this distracting output any more.
- If you get odd characters printed in the test suite, like `Γ£ö`, then that's a general Unicode issue; this does not affect correctness of the test suite.
- If Accelerate functions seem to all be out of scope in `cabal repl`, try either:
 - `:add *Quickhull` inside `cabal repl`
 - using `cabal repl -fno-object-code` instead of `cabal repl`