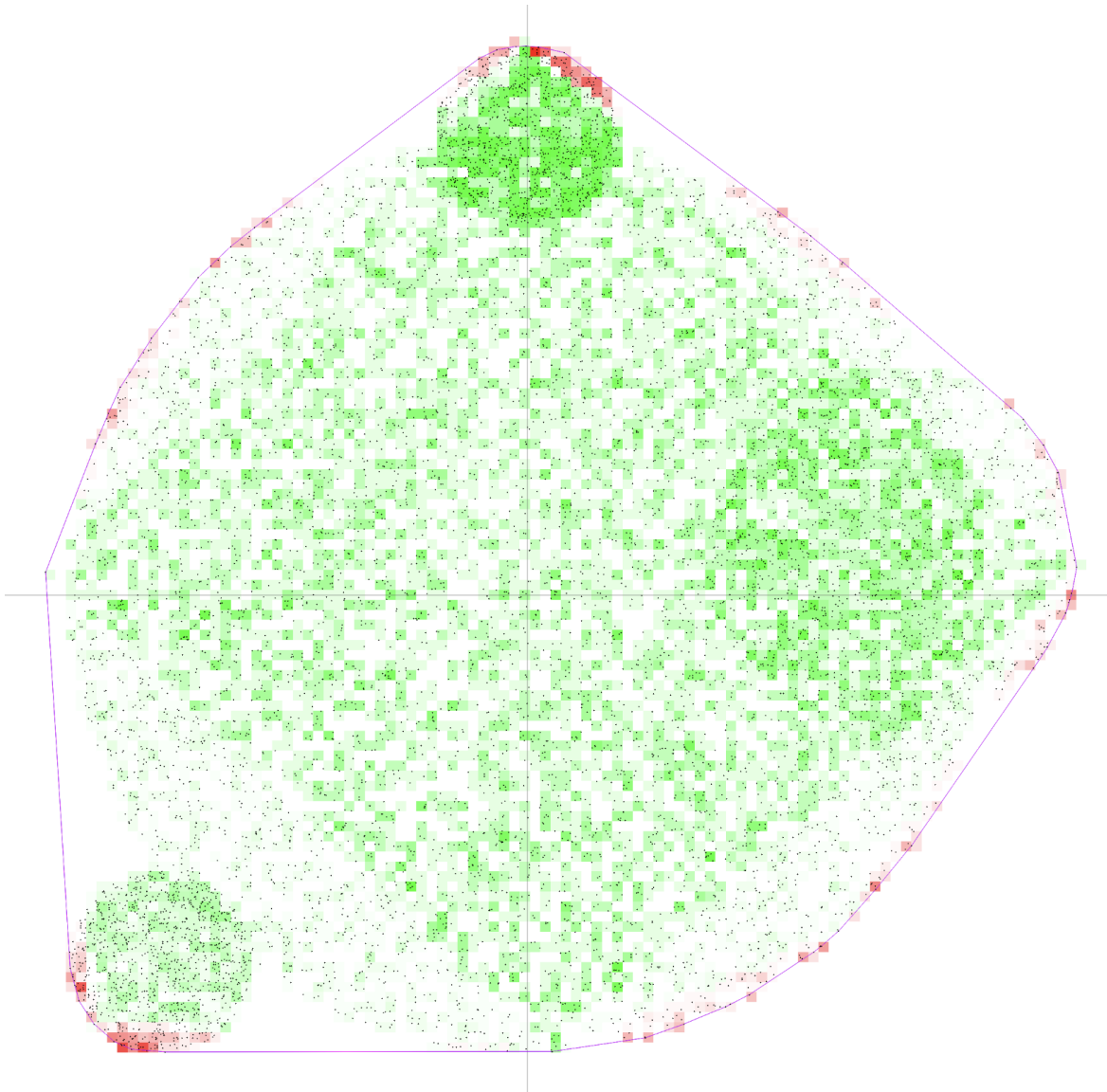


PPA-F25 hw5



Logistics

This homework consists of 100 points total, with points for individual tasks as indicated below. There is 1 coding task in this assignment.

Please submit your work on Gradescope.

For the coding tasks, you will edit and submit `MyQuickhull.sml`.

Due Date: This submission is due at **5:00pm EDT on Tuesday, Oct 14**. Course policy on late submissions is available on the course website. (<https://cs.nyu.edu/~shw8119/courses/f25/3033-121-ppa/>)

Setup

- Follow the instructions on the course website to access one of the **crunchy** compute servers.
- Install MaPLe. For the **crunchy** servers, you can:
 - Download <https://cs.nyu.edu/~shw8119/courses/f25/3033-121-ppa/resources/impl-845f76f.tgz> and copy it to the server
 - Unpack by running `tar xzf impl-845f76f.tgz`. This will create a directory `impl-845f76f/`
 - The compiler, `mpl`, is located at `impl-845f76f/bin/mpl`
 - Add `impl-845f76f/bin` to your `PATH` so that you can access it easily: `export PATH="$(pwd -P)/impl-845f76f/bin:$PATH"`. We recommend updating your `~/.bashrc` file or other configuration file as appropriate.

Optimizing Quickhull

In `ReferenceQuickhull.sml`, there is an implementation of the quickhull algorithm similar to what we discussed in class.

Recall that this algorithm makes extensive use of a primitive, `tri_area(p,q,r)`, which computes the signed area of the triangle with vertices `p`, `q`, and `r`. The resulting quantity is positive if the path `p->q->r` is a counter-clockwise turn, and negative if clockwise. Equivalently, `tri_area(p,q,r)` is positive if `r` is above the line `p->q`, oriented with `p` on the left and `q` on the right.

Your task in this homework assignment is optimize this implementation, producing a more efficient implementation of the quickhull algorithm.

Task 1 (100 points). In `MyQuickhull.sml`, implement the function `hull`:

```
val hull: (real * real) Seq.t -> int Seq.t
```

This function takes a sequence of 2D points `(x,y)` as input and returns the indices of the points on the convex hull, sorted clockwise, starting from the left-most (smallest `x`-coordinate) point.

Your implementation must meet the following performance goals:

- In comparison to `ReferenceQuickhull.hull`, your implementation should perform **half as many calls** to `tri_area`, or fewer.
- In comparison to `ReferenceQuickhull.hull`, your implementation should be **at least 2x faster**.

The points are allocated as follows:

- 55 points: correct implementation that passes all test cases and reduces the number of calls to `tri_area` by 50% or more.
- 45 points for raw performance, allocated as follows. **These points are only possible if you get the full 55 points for the first part.**
 - 15 points: at least 2x faster than the reference implementation on 1 processor.
 - 15 points: at least 2x faster than the reference implementation on 30 processors.
 - 15 points: at least 2x faster than the reference implementation on 60 processors.

A few recommendations for performance:

- Reducing the number of calls to `tri_area` will help, but may not be enough on its own.
- Sequences with “simpler” element types often have more efficient memory layouts. Try reorganizing the code to avoid storing tuples in sequences when possible.
 - For example, creating sequences of type `int Seq.t` or `real Seq.t` can be significantly faster than `(int * int) Seq.t` or `(real * real) Seq.t`.

- This is because simple primitive types (such as `int`, `real`, `bool`, `char`) are small enough to fit into a single machine word, and are stored “in-place” in an underlying array, with no pointer indirection.
- In contrast, a sequence of type `(int * int) Seq.t` is often represented in memory as an **array of pointers to tuples**. The tuples need to be individual allocated and garbage-collected, leading to higher overheads for garbage collection. Additionally, the memory indirection can lead to additional cache misses.

Test suite

You can test your implementation as follows. This will run your code on the full test suite and measure the number of calls to `tri_area`. To get the full 55 points for the first part, you need to pass all of these tests. We will use the same test suite when grading.

```
$ make test
$ ./test
```

Performance testing

For performance testing, we will use the following commands, replacing `<P>` with the number of processors. We will use at least 20 repetitions, but possibly more for more stable measurements.

```
$ make main
```

```
# run the reference implementation on P processors
$ ./main @mpl procs <P> -- -impl reference -warmup 3 -repeat 20 -n 2000000
...
average XXXXXs    # we will use these measurement as the baseline
...

# run your code on P processors
$ ./main @mpl procs <P> -- -impl mine -warmup 3 -repeat 20 -n 2000000
...
average XXXXXs    # goal: this measurement should be 2x faster than
...               # the reference measurement above.
```

We plan to use `crunchy1.cims.nyu.edu` for performance testing. Here are reference measurements. Your goal is to beat all three of these measurements by at least 2x when using the same number of processors. (Note that there is some performance variability in the reference quickhull implementation due to a high rate of allocation and garbage collection, so we use more repetitions here to get a more consistent measurement.)

```
$ ./main @mpl procs 1 -- -impl reference -warmup 3 -repeat 20 -n 2000000
...
average 2.5720s

$ ./main @mpl procs 30 -- -impl reference -warmup 3 -repeat 50 -n 2000000
...
average 0.2441s

$ ./main @mpl procs 60 -- -impl reference -warmup 3 -repeat 50 -n 2000000
...
average 0.2134s
```

Heatmap of calls to `tri_area`

For analyzing the number of calls to `tri_area`, we have included a “heatmap” script that shows visually where your algorithm improves over the reference implementation. For each call to `tri_area(p,q,r)`, it drops a token in a bucket representing a small region around `r`, and compares the number of tokens in these buckets against the reference implementation doing the same.

In the heatmap, green regions are an improvement over the reference implementation (i.e., fewer calls to `tri_area(p,q,r)` for in the region around `r`). Red is worse than (i.e., more calls than) the reference implementation.

```
$ make heatmap
$ ./heatmap -n 10000 -resolution 1000 -heatmap-block-size 5 -output heatmap.ppm
...
# look at heatmap.ppm to see the output
```

Feel free to fiddle with the parameters! The `resolution` parameter controls how many pixels there are per unit. The `heatmap-block-size` parameter controls the size of the heatmap blocks (in pixels).