Introduction to Programming with Scientific Applications (Spring 2023)

Final project - Geometric Median

Name
Théodore Guy Andreas Doubinsky
Jeppe Tilby
Mattias Nørgaard Agerbo

Contents

1 Introduction / Resume

With drone-technology on the rise, shipping companies are looking into the possibility of using drones to deliver parcels directly to costumers. This project tries to implement a foundation of algorithms that can determine the best place for hubs for these drones. This report describes the code we have created to solve this problem, which includes functions to calculate a geometric median of points in 2d-space, and some functions to visualize this geometric median, with some efficiency-tests on the former. Furthermore, this report discusses potential future implementations and improvements that the code could be upgraded with, as well as whether or not this code can be used for its intended application. The solution chapter is sorted as answers to exercises presented by Gerth Brodal, and can be found at: `https://gsbrodal.github.io/ipsa/project-1-medians`.

2 Solution

This solution uses functions from the packages `scipy` (namely `minimize`) and `matplotlib`, since these packages are very standard for optimizing and plotting graphs. Furthermore, functions from `numpy` and `random` have been used due to their wide functionality, which would be redundant for us to implement.

2.1 Exercise 1.1

For calculating the distance between two points and the sum of all distances from one point to several distinct points, we implemented the simple functions `dist2D` and `dists` respectively, with the latter utilizing the former. We implemented the function `geometric_median`, that utilizes `minimize` to find the geometric median. The function returns the geometric median as a tuple per default, but with the option (`verbose`) of returning the full `scipy.optimize.optimize.OptimizeResult` class that `minimize` returns, which we implemented for later use in the function `two_geometric_medians`. In order to get a decent start guess for the aforementioned function, we implemented the function `average`, that rather naively takes the mean over each coordinate index in the set of points.

2.2 Exercise 1.2

We implemented the function `make_plot`, that plots the calculated geometric median, the points of the given set, and dashed lines to indicate the distance from a point to the geometric median. The function was written in such a fashion that it would be usable later when plotting the function `two_geometric_medians` and in exercise 1.3 in which a contour plot is to be plotted.

We wrote the code in such a manner that it is optional to give the function a precalculated geometric median as a variable. If not given one, the function calls the function `geometric_median` on its own.

When writing the function we payed attention to the order of the objects being plotted so as the layering would be as desired. This resulted in having to run through the set of points twice so the points would be plotted on top of the distance indicator lines. We did not make a plot using `make_plot` alone, as it was not stated explicitly in the problem description and

it seemed redundant as we had to plot it anyway on top of the contour plot in the upcoming exercise 1.3.

2.3 Exercise 1.3

We implemented the function `make_contour_plot`, that similarly to `make_plot` has the option of taking a precalculated geometric median. After plotting the contour plot it calls the function `make_plot` so as to indicate the set of points etc. Many design choices were made and variables implemented, when writing the contour plot function. Some of which that were left for potential use by others.

In order to not have the points of the set be on the edge of the contour plot, we wrote code that automatically based on the width of the given plot adds an increment of 5% to each side of the plot. Several `if`-statements were implemented so as to not allow the width of the contour plot be 0, which can happen should a set of points falling on a line parallel with the x-axis or y-axis be given as an input. Furthermore, three options for this plot have been implemented. `x_margin_scale` and `y_margin_scale` determine how much the margin should be scaled (1 being default-value). This is especially useful when doing subplots. The last option is `resolution`, that determines at what resolution the contour should be created - we found that 100 is more than enough for a nice contour on a computer screen.

When doing the plots described in the problem description exercise 1.3, matplotlib's `subplot` were utilized. This particular plot can be seen on figure 3.

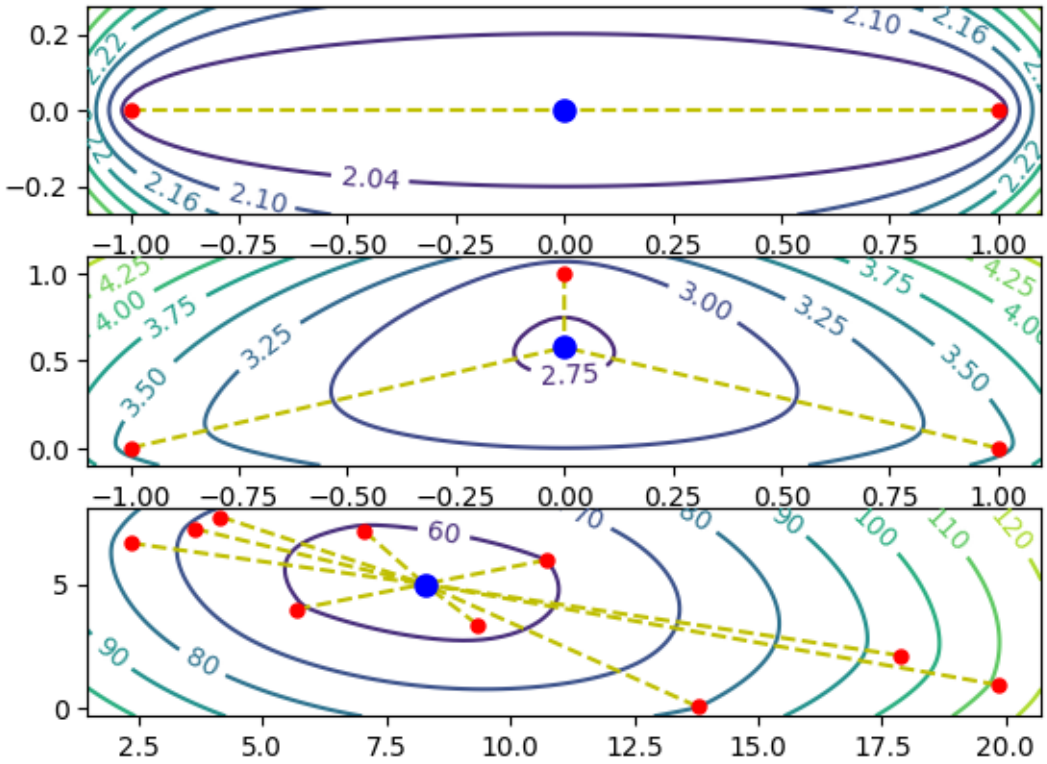


Figure 1: Example of subplot based on exercises 1.2 and 1.3

2.4 Exercise 2.1

We implemented the function `two_geometric_medians` that, given a set of 3 or more points, calculates the best 2-partition of the set, such that sum of `geometric_median` taken on each of the subsets is minimized. In order to check whether a point laid to the right or left of a bisector, we used a function `direction`, that utilizes the properties of the cross product to determine this (the well-known *right-hand-rule*). Initially, we tried implementing `two_geometric_medians` as another `scipy.optimize.minimize` problem, where two points can be interpreted as a singular 4-dimensional point - however, this required us to create weird methods for an initial value, and we therefore resorted to an algorithm that checked all possible bisectors as described in problem description.

2.5 Exercise 2.2

We wrote a crude function `make_bisector_plot` that simply takes the output of `two_geometric_medians` (and later `new_two_geometric_medians`), and makes a plot featuring a bisector.

As mentioned prior we used the already implemented function `make_plot` to plot the two groups of geometric medians.

We also implemented a function `perp_bisector`, that given two distinct points, using basic geometry, returns the midpoint of said points and the perpendicular vector to the line between the points. We used this when plotting the true bisector using `axline` imported from `matplotlib.pyplot`.

An example of this can be seen in figure 2.

2.6 Exercise 2.3 (optional)

To extend `two_geometric_medians` so it does not potentially fail when taking sets containing three or more points on a line we have extended the same algorithm, such that it stores all points falling on the translate bisector, and using the `subset` function finds all the possible 2-partitions along the translate bisector separated into the left and right subsets respectively. The algorithm evaluates the sum of these subsets by using `geometric_median`.

Since this is very heavy computational wise, we considered an alternate algorithm that will be discussed in the Discussion section 4.

When we wrote the function we assumed that the function as in the problem description only takes sets i.e. no duplicates of points as can be present in a list. This made the computation of the complement sets easier. Alternatively we could have written a function that found the complement set given other data types.

We futhermore found a set (S5) consisting of 6 points on a straight line where we correctly predicted that `two_geometric_medians` fails, but `new_two_geometric_medians` finds a correct solution than `two_geometric_medians`. Showing that at least for this set, `new_two_geometric_medians` returns the best solution.

The solutions can be seen on figure 3.

3 Tests

We have done some tests on the function for `new_two_geometric_medians`, that allow for three points on a line. We have tested its runtime compared to the number of input points

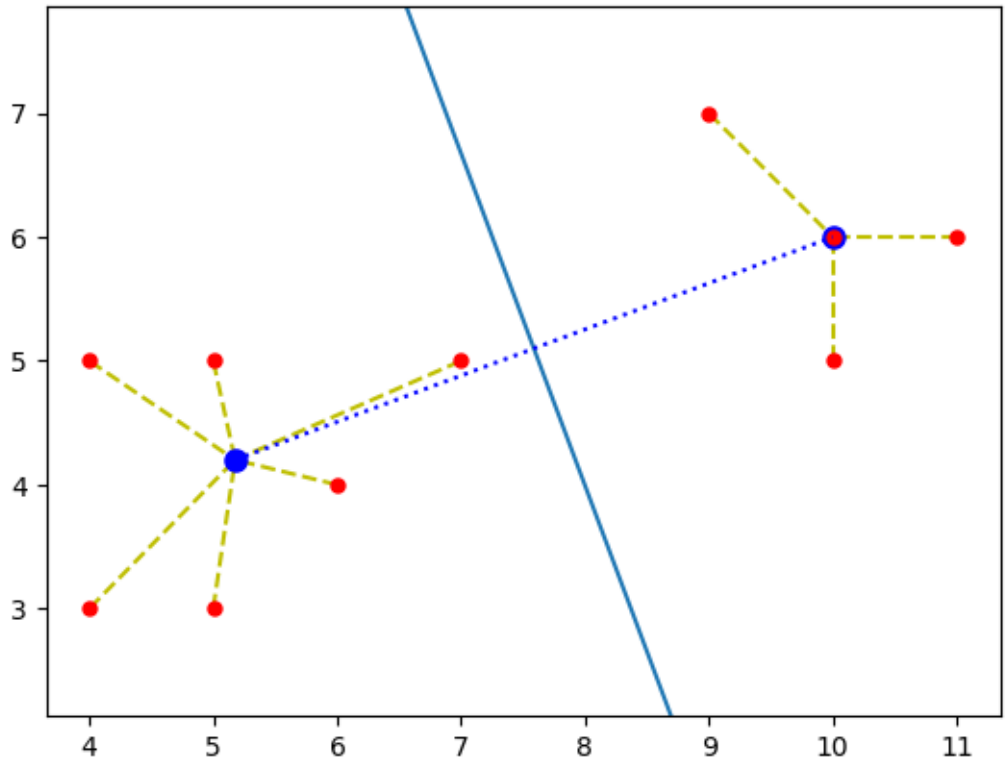


Figure 2: Example of partition of sets of points done by two_geometric_median.

using the python module `timeit`, with the following code:

```
for i in range(10):
    S = {(unif(0, 20), unif(0, 10)) for _ in range(10 * (i + 1))}
    print(f"Testing with n_points = {10 * (i+1)}:")
    print(timeit.timeit("new_two_geometric_medians(S)", number = 1,
    globals = globals()))
```

Listing 1: timeit test

Here we get the following results

Number of points	10	20	30	40	50	60	70	80	90	100
Runtime (seconds)	0.57	2.67	8.61	17.16	29.85	45.36	67.07	99.85	133.92	179.91

Table 1: timeit runtimes

The runtimes are recorded on an older *Lenovo Thinkpad x250*, so one could expect a positive time bias compared to more modern machines. The algorithm runtime seems to be quite slow, especially for its intended purpose, where parcel drones delivering to adresses probably needs this algorithm for $n = 1000$. Furthermore, these numbers are to be taken with a grain of salt, since point-sets can vary a lot even though they have the same number of points, and we only run on one point-set given the number of points. To test how big this grain of salt is, we have done a similar test but running 50 tests on 20 points (with different point sets). Estimating a mean and standard variation of this data, we get a mean

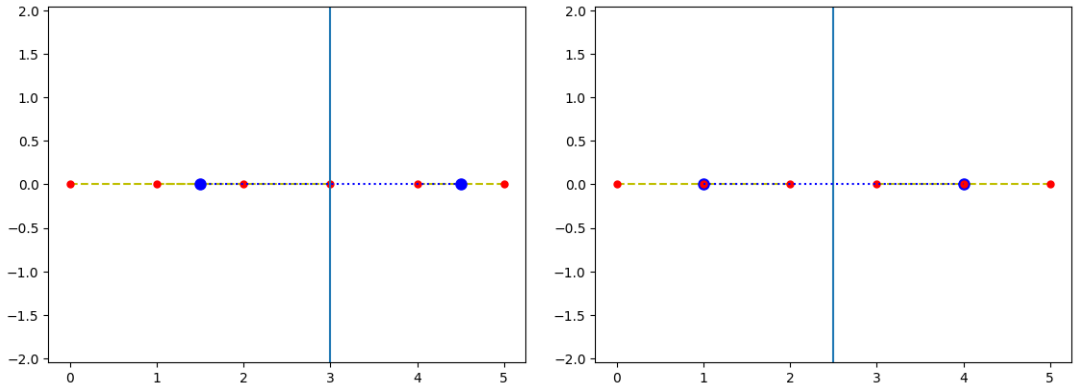


Figure 3: Left plot is showing solution by `two_geometric_median` and right plot is showing solution by `new_two_geometric_median`.

of $\mu = 3.08$ and a standard deviation of $\sigma = 0.23$. Furthermore, the minimal time was 2.71 and the maximal time was 3.78 seconds, which is a large variation, but definitely not an extreme one.

4 Discussion

This code was made to optimize antenna positions to deliver parcels. Antennae, however, have a maximum range that weren't taking into consideration - and a further expansion of the code would then be to implement some kind of `n_geometric_medians`, that could place a minimum number of antennae given a range of antennas, such that all points of interest are in the range of at least one antenna. If this grid becomes to large, one should possibly begin to take the curvature of the earth into consideration.

Furthermore, our implementation of `geometric_median` doesn't care about *consumer equality*, i.e. given two consumers on a straight line, a person would probably choose to put the median point on the midpoint between the two, whereas this algorithm would treat all points on the line between the two consumers as equally good.

To speed up computation of the `new_two_geometric_medians` function we considered implementing the following optimization. Instead of running over every possible 2-partition of the set of points falling on the bisector, one could use the properties of the dot product to partition the 3 or more points falling on the bisector into 2 subsets and thus reducing the number of cases needed to be checked down to 4 from 2^n, where n is the number of points falling on the translate bisector.

A visual proof of this can be seen in the appendix, but here is the mathematical idea. Given 2 points (p_1 and p_2) falling on the translate bisector, we use these to partition the set into two disjoint sets, using the direction of the translate bisector to indicate the order of points (going from p_1 to p_2 with p_j being a random point on the bisector), the subsets would thus be given by those points falling after the base point (p_1) used in the construction of the translate bisector ($\{ p_j \mid \langle \overrightarrow{p_1 p_j}, \overrightarrow{p_1 p_2} \rangle > 0 \}$) and those falling before it including the base point ($\{ p_j \mid \langle , \overrightarrow{p_1 p_j}, \overrightarrow{p_1 p_2} \rangle \leq 0 \}$). This would bring the possible partitions needed to be checked down to 4 i.e. empty set, whole set, before and after base point with there respective

compliments.

This optimization probably won't change the test results in the prior section, since those points were random and thereby with a high probability not on a line, but it will instead be a great on point sets where there are many points on the line (i.e. roads and such). The algorithm `two_geometric_medians` probably runs slow due to the fact that is has to run over $\frac{n(n-1)}{2}$ possibilities, so finding a way to reduce this number would probably greatly decrease the runtime.

5 Conclusion

We have created a foundation to create functions that decide where to put delivery drone hubs, which allows us to put up to two hubs with minimal total distance to consumers, as well as functions that visualize this process. These functions have some obvious optimizations, which have not been implemented yet, and in a true application, further functions are needed, since we have only done a base-case solution. Other than that, our implementation is a solid foundation for the problem this project wished to tackle.

6 Appendix

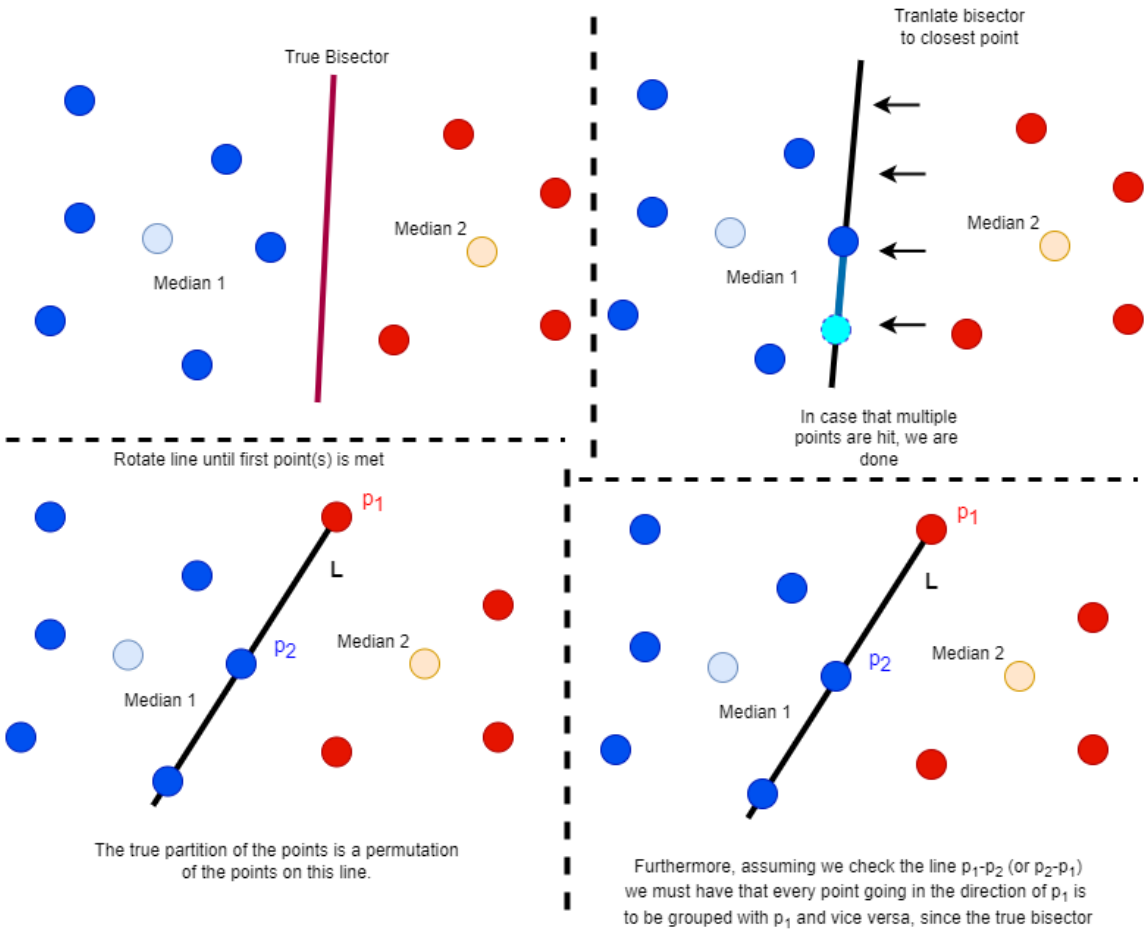
6.1 Test times and code of the repeated 20-point test:

```
    test_list = []
for i in range(50):
    S = {(unif(0, 20), unif(0, 10)) for _ in range(20)}
    test_list.append(timeit.timeit("new_two_geometric_medians(S)", number = 1,
     globals=globals()))
print(test_list)
```

With the result:

[2.9380254, 3.1410148000000007, 3.0841905999999994, 2.9298408999999985,
3.590027299999999, 3.5404613000000005, 3.3325397000000017, 2.9862321999999963,
3.0370232, 3.127695100000004, 3.308152200000002, 2.7230500000000006,
2.8065527999999986, 2.9055305999999987, 3.0217760000000027, 2.9394956000000008,
2.9993351000000033, 3.1518913999999967, 3.016214600000005, 3.0614009000000095,
3.305220200000008, 3.2331688000000014, 3.43919369999999, 3.4432556000000005,
3.3152828999999997, 3.3885106000000036, 3.027803300000002, 3.0221454999999935,
3.013186999999988, 2.7299526999999983, 3.168588099999994, 3.014624900000001,
3.2413860000000057, 3.7818366000000054, 2.9373549000000025, 3.1790670000000034,
2.8836127000000005, 3.0445081999999957, 3.0543667000000028, 3.256215900000001,
3.0188282999999956, 2.953342099999986, 2.862612100000007, 2.8280823999999996,
2.7780672000000095, 2.707984099999976, 3.1557004999999947, 2.872206900000009,
2.800967600000007, 3.0580194000000063]

6.2 Visual proof of the optimization of two_geometric_medians



The last thing to consider is if p_1-p_2 cannot be found in the way above
This means either that all points are contained in one of the partitions, or if all the points are contained on the true bisector. In either case this would be solved by putting all points either in the left or in the right partition.

In conclusion: The permutation of distribution of points on **L** can be reduced to four cases. Letting p_1 and p_2 be the points generating the bisector, the four cases are:

Case 1: Put all points on the bisector in blue section.
Case 2: Put all points on the bisector in red section
Case 3: Put p_1 and all points going in the p_1-direction in blue section
Case 4: Put p_1 and all points going in the p_1-direction in red section