TMA4180 Project: Location analysis

Kristian Holme, Jo Andersson Stokke, and Caroline Jensen

1 Introduction

We consider the following location problems

$$\min_{x \in \mathbf{R}^2} \max_{a \in A} d(a, x) \tag{1}$$

$$\min_{x \in \mathbf{R}^2} \sum_{a \in A} d(a, x) \tag{2}$$

where $A = \{a^1, ..., a^m\}$ is the set of all given locations. We denote $a^i = (a_1^i, a_2^i)$ and the new location $x = (x_1, x_2)^T \in \mathbf{R}^2$. We will consider the following distances:

$$d_1(x, y) = |x_1 - y_1| + |x_2 - y_2| \tag{3}$$

$$d_2(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} \tag{4}$$

$$d_\infty(x, y) = max\{|x_1 - y_1|, |x_2 - y_2|\} \tag{5}$$

2 2.2

2.1

Proof that the distances given in equation (3), (4) and (5) are metrics.

For (3);

Definiteness:

$$d_1(x, y) = 0 \Leftrightarrow |x_1 - y_1| + |x_2 - y_2| = 0$$

Since the absolute values always are equal or larger than zero, we get

$$x_1 - y_1 = 0 \Rightarrow x_1 = y_1 \;\; and \;\; x_2 - y_2 = 0 \Rightarrow x_2 = y_2 \;\; \Rightarrow x = y$$

Symmetry:

$$d_1(x, y) = |x_1 - y_1| + |x_2 - y_2| = |-(y_1 - x_1)| + |-(y_2 - x_2)| = |y_1 - x_1| + |y_2 - x_2| = d_1(y, x)$$

Triangle inequality:

$$\begin{aligned} d_1(x, z) &= |x_1 - z_1| + |x_2 - z_2| \\ &= |x_1 - y_1 + y_1 - z_1| + |x_2 - y_2 + y_2 - z_2| \\ &\leq |x_1 - y_1| + |y_1 - z_1| + |x_2 - y_2| + |y_2 - z_2| = d_1(x, y) + d_1(y, z) \end{aligned}$$

Since $d_1(x, y)$ fullfills definiteness, symmetry and the triangle inequality it is a metric. □

For (4);

Definiteness:

$$d_2(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} = 0 \Leftrightarrow (x_1 - y_1)^2 + (x_2 - y_2)^2 = 0$$

Since a squared number always is equal or larger than zero, we get

$$x_1 - y_1 = 0 \Rightarrow x_1 = y_1 \;\; and \;\; x_2 - y_2 = 0 \Rightarrow x_2 = y_2 \;\; \Rightarrow x = y$$

Symmetry:

$$d_2(x,y) = \sqrt{(x_1-y_1)^2+(x_2-y_2)^2} = \sqrt{(-(y_1-x_1))^2+(-(y_2-x_2))^2} = \sqrt{(y_1-x_1)^2+(y_2-x_2)^2} = d_2(y,x)$$

Triangle inequality:

$$\begin{aligned}
d_2(x,z) &= \sqrt{(x_1-z_1)^2+(x_2-z_2)^2} \\
&= \sqrt{\sum_{i=1}^{2}(x_i-y_i+y_i-z_1)^2} \\
&= \sqrt{\sum_{i=1}^{2}(x_i-y_i)^2+\sum_{i=1}^{2}(y_i-z_i)^2+2\sum_{i=1}^{2}(x_i-y_i)(y_i-z_i)} \\
&\leq \sqrt{\sum_{i=1}^{2}(x_i-y_i)^2+\sum_{i=1}^{2}(y_i-z_i)^2+2((\sum_{i=1}^{2}(x_i-y_i)^2)^{\frac{1}{2}}(\sum_{i=1}^{2}(y_i-z_i)^2)^{\frac{1}{2}})} \\
&= \sqrt{(\sum_{i=1}^{2}(x_i-y_i)^2)^{\frac{1}{2}}+(\sum_{i=1}^{2}(y_i-z_i)^2)^{\frac{1}{2}})^2} \\
&= \sqrt{\sum_{i=1}^{2}(x_i-y_i)^2}+\sqrt{\sum_{i=1}^{2}(y_i-z_i)^2} = d_2(x,y)+d_2(y,z)
\end{aligned}$$

Since $d_2(x,y)$ fullfills definiteness, symmetry and the triangle inequality it is a metric. □
For (5);
Definiteness:

$$d_\infty(x,y) = \max\{|x_1-y_1|,|x_2-y_2|\} = 0 \Rightarrow |x_1-y_1| = 0 \ \textit{and} \ |x_2-y_2| = 0 \Rightarrow x = y$$

Symmetry:

$$d_\infty(x,y) = \max\{|x_1-y_1|,|x_2-y_2|\} = \max\{|-(y_1-x_1)|,|-(y_2-x_2)|\} = \max\{|y_1-x_2|,|y_2-x_2|\} = d_\infty(y,x)$$

Triangle inequality:

$$\begin{aligned}
d_\infty(x,z) &= \max\{|x_1-z_1|,|x_2-z_2|\} \\
&= \max\{|x_1-y_1+y_1-z_1|,|x_2-y_2+y_2-z_2|\} \\
&\leq \max\{|x_1-y_1|+|y_1-z_1|,|x_2-y_2|+|y_2-z_2|\} \\
&= \max\{|x_1-y_1|,|x_2-y_2|\}+\max\{|y_1-z_1|,|y_2-z_2|\} \\
&= d_\infty(x,y)+d_\infty(y,z)
\end{aligned}$$

Since $d_\infty(x,y)$ fullfills definiteness, symmetry and the triangle inequality it is a metric. □

2.2

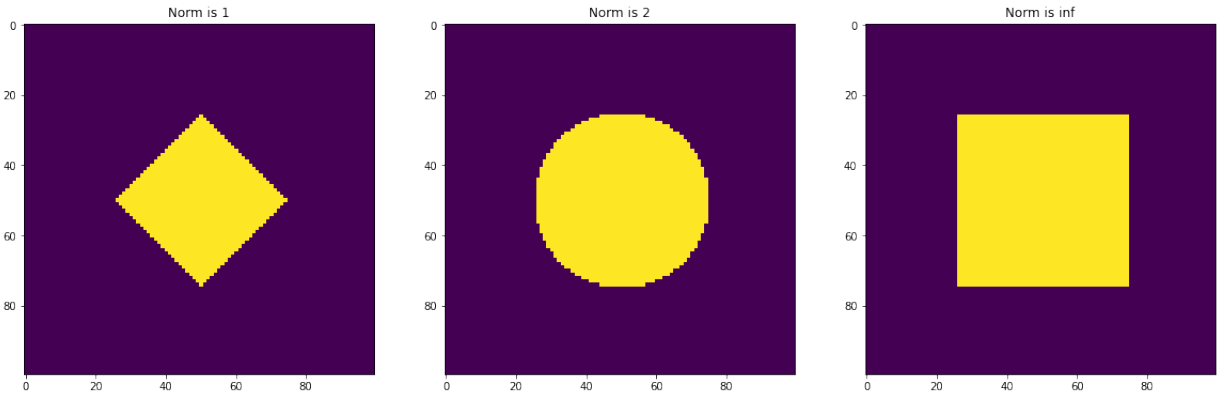

Figure 1: Unit balls with different norms

2.3

Using the triangle inequality for norms we get

$$\begin{aligned} ||xt + (1-t)y|| &\leq ||xt|| + ||(1-t)y|| \\ &= |t| \cdot ||x| + |1-t| \cdot ||y|| \end{aligned}$$

Let $t \in [0, 1]$. Then $|t| = t$ and $|1 - t| = 1 - t$. Then we have

$$||xt + (1-t)y|| \leq t \cdot ||x|| + (1-t|) \cdot ||y||,$$

So every norm is convex. □

2.4

Let

$$\begin{aligned} f_1(x) &= \max_{a \in A} d(a, x) \text{ and} \\ f_2(x) &= \sum_{a \in A} d(a, x), \end{aligned}$$

where $a, x \in \mathbb{R}^2$.

First a result regarding metrics induced by norms. Since all the three metrics we consider in this project are induced norms, the following result holds for all of them.

$$\begin{aligned} d(a, xt + (1-t)y) &= ||a - tx - (1-t)y|| \\ &= ||ta - tx + (1-t)a - (1-t)y|| \\ &\leq t||a - x|| + (1-t)||a - y|| \\ &= td(a, x) + (1-t)d(a, y). \end{aligned} \tag{6}$$

Using this inequality we obtain for f_1:

$$\begin{aligned} f_1(tx+(1-t)y) &= \max_{a\in A} d(a, tx+(1-t)y) \\ &\leq \max_{a\in A} [td(a,x)+(1-t)d(a,y)] \\ &\leq \max_{a\in A} td(a,x) + \max_{a\in A}(1-t)d(a,y) \\ &= t\max_{a\in A} d(a,x) + (1-t)\max_{a\in A} d(a,y) \\ &= tf_1(x)+(1-t)f_1(y). \end{aligned}$$

So f_1 is convex.
Now we examine f_2:

$$\begin{aligned} f_2(tx+(1-t)y) &= \sum_{a\in A} d(a, tx+(1-t)y) \\ &\leq \sum_{a\in A} [td(a,x)+(1-t)d(a,y)] \\ &= t\sum_{a\in A} d(a,x) + (1-t)\sum_{a\in A} d(a,y) \\ &= tf_2(x)+(1-t)f_2(y). \end{aligned}$$

So f_2 is also convex. □

2.5

Let $f_{1,2}(x) = \max_{a\in A} d_2(a,x)$.

Given a point x in $\mathbb{R}^2$, $f_{1,2}(x)$ is the distance to the point in A furthest away. This means that if we draw a circle around x, this circle must contain all $a \in A$. We want to find the point where $f_{1,2}(x)$ is minimized, so the problem is equivalent to finding the smallest circle containing all the points in A. Then the solution to problem (1) is the center of the circle.

2.6

Solution approach for problem (2) with Manhatten distance function (3).

The problem is:

$$\min_{x\in\mathbf{R}^2} \sum_{a\in A} (|a_1 - x_1| + |a_2 - x_2|) = \min_{x\in\mathbf{R}^2} (\sum_{a\in A} |a_1 - x_1| + \sum_{a\in A} |a_2 - x_2|)$$

Our solution approach would be:

1. For $x_1 \in \mathbf{R}$ find $\min_{x_1\in\mathbf{R}} \sum_{a\in A} |a_1 - x_1|$
2. For $x_2 \in \mathbf{R}$ find $\min_{x_2\in\mathbf{R}} \sum_{a\in A} |a_2 - x_2|$

2.7

Let $f(x) = \sum_{a\in A}(d_2(a,x))^2 = \sum_{a\in A}(x_1 - a_1)^2 + (x_2 - a_2)^2$

First we want to show that the composition of a convex and a convex and monotone function is convex.

Let $u(x)$ and $g(x)$ be convex functions, let g be monotone, and let $h(x) = g(u(x))$.

Then we have:

$$\begin{aligned} h(tx+(1-t)y) &= g(u(tx+(1-t)y)) \\ &\leq g(tu(x)+(1-t)u(y)) \\ &\leq tg(u(x))+(1-t)g(u(y)) \\ &= th(x)+(1-t)h(y), \end{aligned}$$

so h is convex.

Now let $g(x) = x^2$ and $u(x) = d_2(a, x)$. Since u now is positive-valued and g is monotone on $\mathbb{R}^+$, we know that $h(x) = (d_2(a, x))^2$ is convex. This follows from (6) since $d_2(a, x)$ is derived from $||\cdot||_2$. Since a sum of convex functions is convex, f must be convex.

To compute the minimizer of f, we compute the gradient:

$$\begin{aligned} \nabla f(x) &= \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial}{\partial x_1}\sum_{a\in A}(x_1-a_1)^2+(x_2-a_2)^2 \\ \frac{\partial}{\partial x_2}\sum_{a\in A}(x_1-a_1)^2+(x_2-a_2)^2 \end{bmatrix} \\ &= \sum_{a\in A}\begin{bmatrix} \frac{\partial}{\partial x_1}[(x_1-a_1)^2+(x_2-a_2)^2] \\ \frac{\partial}{\partial x_2}[(x_1-a_1)^2+(x_2-a_2)^2] \end{bmatrix} \\ &= \sum_{a\in A}\begin{bmatrix} 2(x_1-a_1) \\ 2(x_2-a_2) \end{bmatrix} \\ &= \sum_{a\in A} 2(x-a). \end{aligned}$$

Setting the gradient equal to zero then produces the minima:

$$\begin{gathered} \nabla f(x) = \sum_{a\in A} 2(x-a) = 0 \\ \Leftrightarrow \\ \sum_{a\in A}(x-a) = |A|x - \sum_{a\in A} a = 0 \\ \Leftrightarrow \\ x = \frac{\sum_{a\in A} a}{|A|}. \end{gathered}$$

So the minimizer x is the average of all the points $a \in A$. From the expression for the gradient, we see that the hessian of f is $2|A|I_2$, which is symmetric and positive definite, so x is really a minimizer.

2.8

We will now consider problem (2) with the Euclidean distance function (4)

$$\min_{x\in\mathbf{R}^2}\sum_{a\in A} d_2(a,x) = \min_{x\in\mathbf{R}^2}\sum_{a\in A}\sqrt{(x_1-a_1)^2+(x_2-a_2)^2} \tag{7}$$

The necessary optimality conditions for minimizers, x^*, of this problem are $\nabla\sum_{a\in A} d_2(x^*, a) = 0$ and that the Hessian must be positive semi-definite.

For the first order necessary condition we get:

$$\sum_{a\in A}\nabla d_2(x^*,a) = \begin{bmatrix} \sum_{a\in A}\frac{x_1^*-a_1}{\sqrt{(x_1^*-a_1)^2+(x_2^*-a_2)^2}} \\ \sum_{a\in A}\frac{x_2^*-a_2}{\sqrt{(x_1^*-a_1)^2+(x_2^*-a_2)^2}} \end{bmatrix} = 0$$

$$\Rightarrow \sum_{a\in A}\frac{x_1^*-a_1}{\sqrt{(x_1^*-a_1)^2+(x_2^*-a_2)^2}} = 0 \;\&\; \sum_{a\in A}\frac{x_2^*-a_2}{\sqrt{(x_1^*-a_1)^2+(x_2^*-a_2)^2}} = 0$$

We see that the gradient is not the defined in the cases where $x = a$, which means that a can be a minimizer. This is why in the Weiszfeld algorithm we have to first check if the minimum is attained at an existing location a^i, since the gradient is calculated to find the stopping criterion.

For the second order necessary condition we get:

$$\sum_{a\in A}\nabla^2 d_2(x^*,a) = \begin{bmatrix} \sum_{a\in A}\frac{(x_2^*-a_2)^2}{((x_1^*-a_1)^2+(x_2^*-a_2)^2)^{\frac{3}{2}}} & \sum_{a\in A}\frac{-(x_1^*-a_1)(x_2^*-a_2)^2}{((x_1^*-a_1)^2+(x_2^*-a_2)^2)^{\frac{3}{2}}} \\ \sum_{a\in A}\frac{-(x_1^*-a_1)(x_2^*-a_2)^2}{((x_1^*-a_1)^2+(x_2^*-a_2)^2)^{\frac{3}{2}}} & \sum_{a\in A}\frac{(x_1^*-a_1)^2}{((x_1^*-a_1)^2+(x_2^*-a_2)^2)^{\frac{3}{2}}} \end{bmatrix}$$

For $\sum_{a\in A}\nabla^2 d_2(x^*,a)$ to be positive semi definite we need that all eigenvalues are positive. To show that all the eigenvalues are positive we show that the trace and the determinant are positive, since this is equivalent.

$$tr(\sum_{a\in A}\nabla^2 d_2(x^*,a)) = \sum_{a\in A}\frac{(x_2^*-a_2)^2}{((x_1^*-a_1)^2+(x_2^*-a_2)^2)^{\frac{3}{2}}} + \sum_{a\in A}\frac{(x_1^*-a_1)^2}{((x_1^*-a_1)^2+(x_2^*-a_2)^2)^{\frac{3}{2}}} > 0$$

$$\begin{aligned} det(\sum_{a\in A}\nabla^2 d_2(x^*,a)) &= \sum_{a\in A}\frac{(x_2^*-a_2)^2}{((x_1^*-a_1)^2+(x_2^*-a_2)^2)^{\frac{3}{2}}}\cdot\sum_{a\in A}\frac{(x_1^*-a_1)^2}{((x_1^*-a_1)^2+(x_2^*-a_2)^2)^{\frac{3}{2}}} \\ &\quad -(\sum_{a\in A}\frac{-(x_1^*-a_1)(x_2^*-a_2)^2}{((x_1^*-a_1)^2+(x_2^*-a_2)^2)^{\frac{3}{2}}})^2 \\ &= \sum_{i=1}^{m}\frac{(x_2^*-a_2^i)^2}{((x_1^*-a_1^i)^2+(x_2^*-a_2^i)^2)^{\frac{3}{2}}}\cdot\frac{(x_1^*-a_1^i)^2}{((x_1^*-a_1^i)^2+(x_2^*-a_2^i)^2)^{\frac{3}{2}}} \\ &\quad -\sum_{i=1}^{m}\frac{(x_2^*-a_2^i)^2}{((x_1^*-a_1^i)^2+(x_2^*-a_2^i)^2)^{\frac{3}{2}}}\cdot\frac{(x_1^*-a_1^i)^2}{((x_1^*-a_1^i)^2+(x_2^*-a_2^i)^2)^{\frac{3}{2}}} \\ &\quad +\sum_{i=1}^{m}\sum_{j=1}^{m}\frac{(x_2^*-a_2^i)^2}{((x_1^*-a_1^i)^2+(x_2^*-a_2^i)^2)^{\frac{3}{2}}}\cdot\frac{(x_1^*-a_1^j)^2}{((x_1^*-a_1^j)^2+(x_2^*-a_2^j)^2)^{\frac{3}{2}}}, i\neq j \\ &= \sum_{i=1}^{m}\sum_{j=1}^{m}\frac{(x_2^*-a_2^i)^2}{((x_1^*-a_1^i)^2+(x_2^*-a_2^i)^2)^{\frac{3}{2}}}\cdot\frac{(x_1^*-a_1^j)^2}{((x_1^*-a_1^j)^2+(x_2^*-a_2^j)^2)^{\frac{3}{2}}}, i\neq j \\ &> 0 \end{aligned}$$

So the determinant and the trace are positive for all x^*, which means that all the eigenvalues are positive, so $\sum_{a\in A}\nabla^2 d_2(x^*,a)$ is a positive semi-definite matrix for all x^*. This is as expected since the objective function is convex, as shown in section 2.4.

2.9

Let $A = \{a^1 = (a_1^1, 0), a^2 = (a_1^2, 0)\}$ and $x = (x_1, 0)$

We saw that the solution to the median problem with the squared Euclidean metric was the average of the two points $x^*_{squared} = (\frac{a_1^1 + a_1^2}{2}, 0)$.

If we don't square the metric, the problem is to minimize $f(x) = \sqrt{(x_1 - a_1^1)^2} + \sqrt{(x - a_1^2)^2} = |x_1 - a_1^1| + |x_1 - a_1^2|$. The gradient is $\nabla f(x) = \text{sgn}(x_1 - a_1^1) + \text{sgn}(x_1 - a_1^2)$, which is zero when x_1 is anywhere between a_1^1 and a_1^2.

So in this case the solutions to the two problems are not identical.

3 The Weiszfeld algorithm

3.1

In this problem we want to minimize

$$f(x) = \sum_{i \in \mathcal{M}} v^i d_2(x, a^i) = \sum_{i \in \mathcal{M}} v^i \left[(x_1 - a_1^i)^2 + (x_2 - a_2^i)^2\right]^{1/2}$$

We obtain the partial derivatives ($j \in \{1, 2\}$):

$$\begin{aligned}
\frac{\partial f}{\partial x_j} &= \sum_{i \in \mathcal{M}} v^i \frac{(x_j - a_j^i)}{\left[(x_1 - a_1^i)^2 + (x_2 - a_2^i)^2\right]^{1/2}} \\
&= \sum_{i \in \mathcal{M}} v^i \frac{(x_j - a_j^i)}{d_2(x, a^i)}
\end{aligned}$$

A minimizer must be a critical point (unless it is a location), so we set the partial derivatives to zero and solve for x_j:

$$\begin{aligned}
\sum_{i \in \mathcal{M}} v^i \frac{(x_j - a_j^i)}{d_2(x, a^i)} &= 0 \\
-\sum_{i \in \mathcal{M}} v^i \frac{a_j^i}{d_2(x, a^i)} + \sum_{i \in \mathcal{M}} v^i \frac{x_j}{d_2(x, a^i)} &= 0 \\
\sum_{i \in \mathcal{M}} v^i \frac{x_j}{d_2(x, a^i)} &= \sum_{i \in \mathcal{M}} v^i \frac{a_j^i}{d_2(x, a^i)} \\
x_j \sum_{i \in \mathcal{M}} \frac{v^i}{d_2(x, a^i)} &= \sum_{i \in \mathcal{M}} \frac{v^i a_j^i}{d_2(x, a^i)} \\
x_j &= \frac{\sum_{i \in \mathcal{M}} \frac{v^i a_j^i}{d_2(x, a^i)}}{\sum_{i \in \mathcal{M}} \frac{v^i}{d_2(x, a^i)}}
\end{aligned}$$

This is a fix point equation, so the Weiszfeld algorithm is a fix point iteration algorithm using a reformulation of $\nabla f(x) = 0$ as the fix point equation.

We can reformulate the above equation as

$$
\begin{aligned}
x &= \frac{\sum_{i\in\mathcal{M}} \frac{v^i x}{d_2(x,a^i)}}{\sum_{i\in\mathcal{M}} \frac{v^i}{d_2(x,a^i)}} - \frac{\sum_{i\in\mathcal{M}} \frac{v^i x}{d_2(x,a^i)}}{\sum_{i\in\mathcal{M}} \frac{v^i}{d_2(x,a^i)}} + \frac{\sum_{i\in\mathcal{M}} \frac{v^i a^i}{d_2(x,a^i)}}{\sum_{i\in\mathcal{M}} \frac{v^i}{d_2(x,a^i)}} \\
&= x - \frac{\sum_{i\in\mathcal{M}} \frac{v^i(x-a)}{d_2(x,a^i)}}{\sum_{i\in\mathcal{M}} \frac{v^i}{d_2(x,a^i)}} \\
&= x - \frac{1}{\sum_{i\in\mathcal{M}} \frac{v^i}{d_2(x,a^i)}} \nabla f(x),
\end{aligned}
$$

so the Weiszfeld algorithm is a gradient descent method with step size $\alpha_k = \frac{1}{\sum_{i\in\mathcal{M}} \frac{v^i}{d_2(a^i,x_k)}}$

3.2

The relative accuracy of our current point x is (and notice that it is always non-negative)

$$
r(x) = \frac{f(x) - f(x^*)}{f(x^*)}
$$

From theorem 2 we get that $f(x) - f(x^*) \leq UB(x)$, and from theorem 3 we have $\frac{1}{f(x^*)} \leq \frac{1}{LB(x)}$. Thus it must be the case that $r(x) \leq \frac{UB(x)}{f(x^*)} \leq \frac{UB(x)}{LB(x)}$

Therefore, if $LB(x) > 0$ and $\exists \epsilon > 0 : \frac{UB(x)}{LB(x)} < \epsilon$, we have that $r(x) < \epsilon$, so the relative accuracy of our current iterate x is at most ϵ.

3.3

If want our approximated solution to have a relative accuracy of at most (i.e. equal or better than) ϵ, we can compute

$$
\frac{\|\nabla f(x)\| \cdot \sigma(x)}{f(x) - \|\nabla f(x)\| \cdot \sigma(x)}
$$

and terminate the iteration if this value is less than ϵ. This is easy to do, because we know f, ∇f, and $\sigma(x)$. $\sigma(x)$ is the maximum distance from x to a point in the convex hull of our locations. This maximum must be achieved at the boundary of the convex hull, because for any given point y inside the convex hull the farthest point b on the boundary which lies on the line through x and y must be farther from x than y. And given any point b on the boundary of the convex hull that is not a location, one of the locations that define the line b is on must be farther away from x. This is not true in general, but is true for d_2, because the level sets of $g(y) = d_2(x, y)$ are circles, not polygons.

So $\sigma(x) = \max_{i\in\mathcal{M}} d_2(a^i, x)$, which can easily be calculated for any x.

3.4

The pseudo code for the implemented Weiszfeld algorithm is given in Algorithm 1. Here

$$
Test_k = \left[\left(\sum_{i\in\mathcal{M}\setminus\{k\}} v^i \frac{a_1^k - a_1^i}{d_2\left(a^k, a^i\right)} \right)^2 + \left(\sum_{i\in\mathcal{M}\setminus\{k\}} v^i \frac{a_2^k - a_2^i}{d_2\left(a^k, a^i\right)} \right)^2 \right]^{\frac{1}{2}}
$$

and the function and gradient is given from the expressions in problem 3.1. a^i from the list of points $\mathcal{M}$ which is actually passed as inputs to the algorithm is also called p^k and P respectively.

Algorithm 1: Weiszfeld algorithm for approximating a solution to the Fermat-Weber problem

Input : P array of points in $\mathbb{R}^2$, v list of corresponding weights, tol tolerance for relative error
Output: x list of iterations with last element being the estimated solution, relative error

for p^k *in* P **do**
  Calculate $Test_k$
  if $Test_k \leq v^k$ **then**
    return p^k, 0
  end
end
Create list x
Set x_0 = sum(P) / length of P (The Euclidean Median)
Set relative error = ∞
while *relative error* > *tol* **do**
  Set $x_{numerator,1}, x_{numerator,2}, x_{denominator}$ to 0
  for i *in length of* P **do**
    To $x_{numerator,1}$ add $v^i p_1^i$
    To $x_{numerator,2}$ add $v^i p_1^i$
    To $x_{denominator}$ add $d_2\left(p^i, x\right)$
  end
  Set x_{new} to $[x_{numerator,1}, x_{numerator,2}]$ / $x_{denominator}$
  Append x_{new} to x
  Set fval to $f_{d_2}(x_{new})$
  Set Dfval to $\nabla f_{d_2}(x_{new})$
  Set σ to max $d_2(x_{new}, P)$
  Set relative error to Dfval σ / (fval - Dfval σ)
end
return x, relative error

We then sample 10 points in $\mathbb{R}^2$ from the uniform distribution from the interval $[0, 1]$, and 10 corresponding weight values from the same distribution. We then run the algorithm with the relative tolerance of 10^{-7}. Figure 2 shows one example run from this setup.

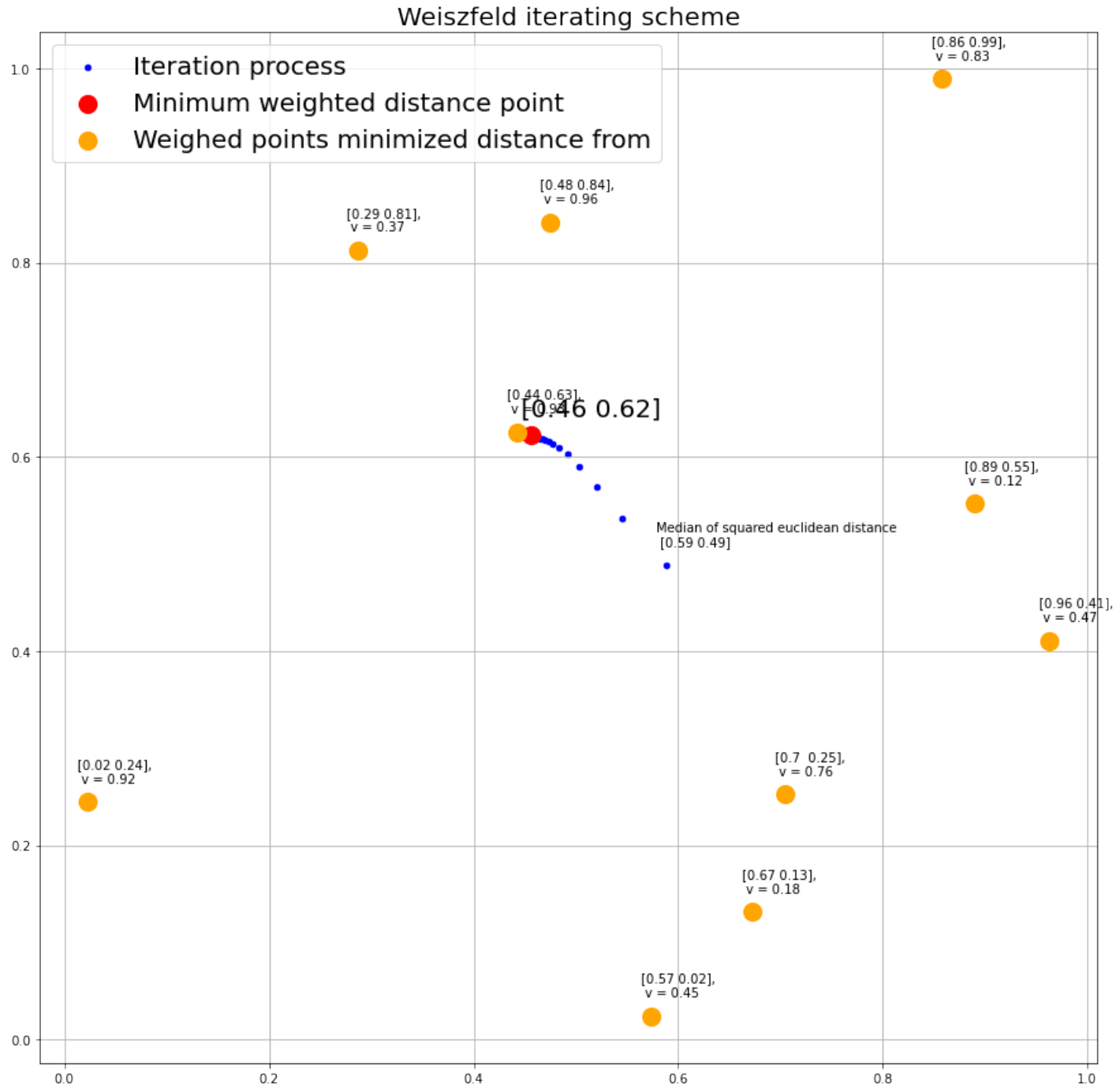

Figure 2: Example test run of the algorithm Weiszfeld

3.5

From the same setup as in 3.2.4, and the same samples, we get the example run in figure 3.

Figure 3: Example test run of the algorithm Backtracking Gradient Descent

Note on implementation of the backtracking gradient descent implementation: As the formula for the relative error in Weiszfelds stopping criteria is proved only using properties of the problem at hand, and not anything specific to the algorithm itself, we implemented the backtracking gradient descent with the same stopping criteria to make the models more commensurable.

How does the algorithms differ? Both algorithms are examples of gradient descent methods. The difference is in how the step size is calculated. In the Weiszfeld algorithm, the step is calculated directly as shown in 3.1, while backtracking does a line search until a step size satisfying the Armijo condition is found. Since the step size in the Weizfeld algorithm is smaller if there are more points, the step size will

get very small when there are many locations to consider, which may be a disadvantage.

How does the two algorithms compare in terms of performance? To answer this we use two metrics. The run time, and the amount of iteration steps. To test this our new setup is as before, but we instead sample 130 points, and run both algorithms on the same inputs 100 times with the relative tolerance of 10^{-7}. Saving the amount of needed iterations and run times for each run, we create the plots of figure 4.

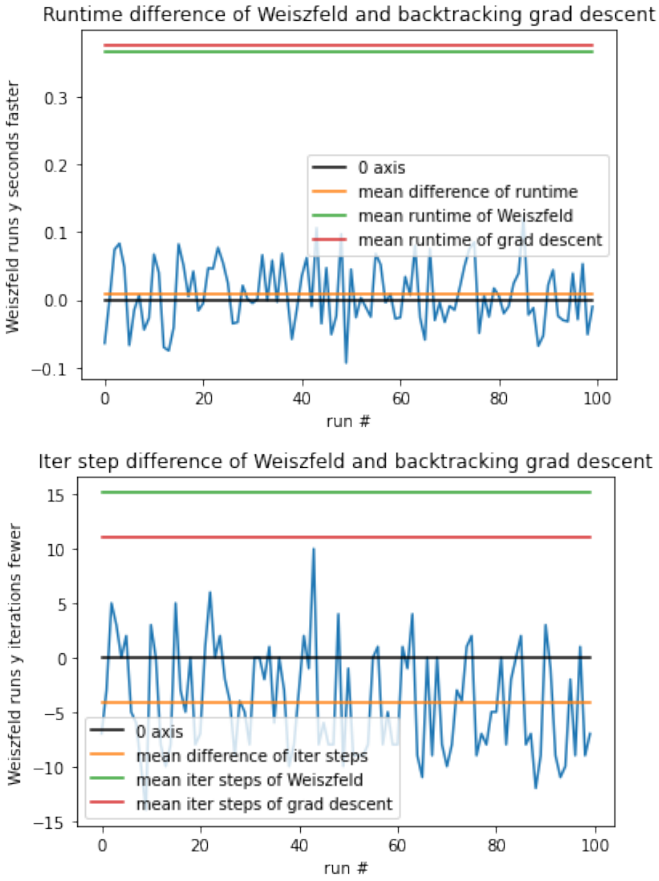

Figure 4: Comparing metrics of performance between Weiszfeld and backtracking gradient descent

From these plots it does seem like the Weiszfeld algorithms on average runs slightly faster than its counterpart, but in reference to the average run time of each algorithm the difference in run time does not seem to differentiate the algorithms too much. In reference to the metric of iteration steps however, the backtracking gradient descent does considerably better. But, as it is often possible to increase computation needed in each iteration for the amount of steps needed, this metric does have serious problems as a measure of performance. Due to this, if an algorithm would have to be preferred based on these metrics, the scheme performing better on run time would probably be preferred of one scoring better on amount of iteration steps. Due to this, Weiszfeld is probably the scheme to prefer in this problem, by a very small margin.

4 Work distribution

All tasks where discussed in the group, but the calculations and implementations where mostly done individually. Project points 2.2.1, 2.2.6 and 2.2.8 were done by Caroline. 2.2.3 - 2.2.5, 2.2.7, 2.2.9 and 3.2.1 were done by Kristian. 2.2.2, 3.2.4 and 3.2.5 were done by Jo. All contributed to 3.2.2 and 3.2.3.

Appendix

4.1 Code

```
import numpy as np
import matplotlib.pyplot as plt
np.set_printoptions(precision=2)
import time

def plotUnitBalls(pnorm, N = 1000):
    grid = np.zeros((N,N))
    r = N/4 #Assume the grid is [-2,2]x[-2,2], then r = N/4 represent the radis of
    for i in range(N):
        for j in range(N):
            if np.linalg.norm(np.array([i - N/2, j - N/2]), ord = pnorm) < r:
                grid[i,j] = 1
    return grid

N = 100
norms = [1,1.5,2,4, np.inf]
M = len(norms)
unitballs = np.zeros((M,N,N))
for count, norm in enumerate(norms):
    unitballs[count] = plotUnitBalls(norm, N)

fig, ax = plt.subplots(1,M, figsize = (20,20))
for i in range(M):
    ax[i].set_title(f"Norm is {norms[i]}")
    ax[i].imshow(unitballs[i])



def Weiszfeld(P, v, tol = 1E-8, iter_limit = 1000):

    '''
    Parameters
    ----------
    P : list of points to minimize distance from
    v : weights of each point
    tol : TYPE, optional
        DESCRIPTION. The default is 1e-8.
    iter_limit : Failsafe iteration limit

    Returns
    -------
    x: approximated optimum
    count/error : iterationsteps if the method converges, error if not

    '''
```

```
M = len(v)
x = []
Test = np.zeros(M)

"""Testing if minimum is attained in P"""
for k in range(M):
    zum1, zum2 = 0, 0
    for i in range(M):
        if i == k:
            continue
        zum1 += v[i]*(P[k,0] - P[i,0])/np.linalg.norm(P[k] - P[i])
        zum2 += v[i]*(P[k,1] - P[i,1])/np.linalg.norm(P[k] - P[i])
    Test[k] = np.sqrt((zum1**2 + zum2**2))

for k, test_k in enumerate(Test):
    if test_k <= v[k]:
        return [P[k]], 0
else:
    euclidMed = P.sum(axis=0)/len(P) #Shown in PDF
    x.append(euclidMed)


"""Begining iteration process"""
rel_error = np.inf
count = 0
while rel_error > tol:
    count += 1
    zum1, zum2, zum3, f, Df = 0, 0, 0, 0, np.zeros(2)
    x.append(np.zeros(2))
    xprev = x[-2]
    for i in range(M):
        zum1 += v[i]*P[i,0]/np.linalg.norm(P[i] - xprev)
        zum2 += v[i]*P[i,1]/np.linalg.norm(P[i] - xprev)
        zum3 += v[i]/np.linalg.norm(P[i] - xprev)
    """Calculating stopping critera by relative error"""
    x[-1] = np.array([zum1/zum3,zum2/zum3])
    x_current = x[-1]
    for i in range(M):
        f += v[i]*np.linalg.norm(P[i] - x_current)
        Df +=  v[i] * (P[i] - x_current)/np.linalg.norm(P[i] - x_current)
    x_vec = np.tile(x[-1], M).transpose()
    sigma = np.zeros(M)
    for i in range(M):
        sigma[i] = np.linalg.norm(x_vec[i] - P[i])
    sigma = np.max(sigma)
    Df_max = max(Df)
    #Df_max = np.linalg.norm(Df)
    rel_error = abs((Df_max*sigma) / (f - Df_max*sigma))
```

```
        """Failsave iter limit"""
        if count > iter_limit:
            print("Weiszfeld iterlimit reached")
            break

    """Return thelist of iteration points"""
    x_final = np.zeros((count+1,2))
    for i, x_i in enumerate(x):
        x_final[i] = np.array(x_i).flatten()
    return x_final, rel_error



#%% Step control
def chooseStep(xk, pk, f, Df, P, v, a=1, rho = 0.5, c1=0.5):
    '''
    Parameters
    ----------
    xk : curent iteration point
    pk : descent direction
    f : objective function
    Df : gradient
    a : initial step size. The default is 1.
    rho : step-adjustment factor. The default is 0.5.
    c1 : Armijo-parameter. The default is 0.5.

    Returns
    -------
    a : step size satisfying te Armijo condtition.
    '''
    count = 0
    while (f(xk + a*pk, P, v) > f(xk, P, v) + c1*a*Df(xk, P, v).dot(pk)):
        a = a*rho
        count += 1
    return a

#%% Gradient descent

def gradDescent(f, Df, P, v,  a=1, rho=0.5, c1 = 0.5, backTrack=True, tol=1e-8, iter
    '''
    Parameters
    ----------
    f : objective function
    Df : gradient
    x0 : initial guess.
    a : default step size. The default is 1.
    rho : step-adjustment factor. The default is 0.5.
    c1 : Armijo-parameter. The default is 0.5.
    backTrack : Bool. True if backtracking is required. The default is True.
```

```
    tol : TYPE, optional
        DESCRIPTION. The default is 1e-8.
    retSteps : Bool. True if number of steps should be returned The default is Fals

    Returns
    -------
    x: approximated optimum
    count/error : iterationsteps if the method converges, error if not

    '''
    xprev = np.inf
    p = 0
    M = len(v)
    x = []
    Test = np.zeros(M)


    """Testing if minimum is attained in P"""
    for k in range(M):
        zum1, zum2 = 0, 0
        for i in range(M):
            if i == k:
                continue
            zum1 += v[i]*(P[k,0] - P[i,0])/np.linalg.norm(P[k] - P[i])
            zum2 += v[i]*(P[k,1] - P[i,1])/np.linalg.norm(P[k] - P[i])
        Test[k] = np.sqrt((zum1**2 + zum2**2))

    for k, test_k in enumerate(Test):
        if test_k <= v[k]:
            return [P[k]], 0
    else:
        euclidMed = P.sum(axis=0)/len(P) #Shown in PDF
        x.append(euclidMed)

    rel_error = np.inf
    descent_error = np.inf
    count = 0
    """Gradient descent with backtracking iteration"""
    while(rel_error > tol): #continue while gradient norm is above tolerance
        x.append(np.zeros(2))
        count += 1
        xprev = x[-2]
        Dfprev = Df(xprev, P, v)
        p = -Dfprev #compute direction (negative gradient)
        if backTrack:
            a = chooseStep(xk = xprev, pk = p, f = f, Df = Df, a = a, P = P, v = v,
        x[-1] = (xprev + a*p)      #iterate
        """Calculating stopping critera by relative error"""
        Dfval =  Df(x[-1], P, v)
```

```
        fval = f(x[-1], P, v)
        descent_error = np.linalg.norm(Dfval)
        """ Here the absolute error intitaly used as stoppingcriteria in the gradie
        and such better be compared."""
        rel_error = descent_error/fval #Both are always positive
        if count > iter_limit:
            print("grad descent iterlimit reached")
            break
    x_final = np.zeros((count+1,2))
    for i, x_i in enumerate(x):
        x_final[i] = np.array(x_i).flatten()
    return x_final, descent_error

def f(x, P, v):
    fval = 0
    for i in range(len(v)):
        fval += v[i]*np.linalg.norm(P[i] - x)
    return fval

def Df(x, P, v):
    Dfval = 0
    for i in range(len(v)):
        numerator = v[i]*(P[i] - x)
        denominator = np.linalg.norm(P[i] - x)
        Dfval += numerator/denominator
    #Dfval = (v*(P - np.tile(x, len(v))/(np.linalg.norm(P - x)))).sum(axis = 1)
    return -Dfval

N = 10
P = np.random.uniform(0,1,N*2)
P = P.reshape((N,2))
v = np.random.uniform(0.1,1,len(P))
tol = 1E-7

x, rel_error_x = Weiszfeld(P, v, tol = tol, iter_limit = 10000)
y, rel_error_y = gradDescent(f, Df, P, v, tol = tol, iter_limit = 10000, c1 = 0.5)

fix, ax = plt.subplots(1, 3,figsize = (15,15))
plot_limit = 50
plot_limit_x = min(plot_limit,len(x))
plot_limit_y = min(plot_limit,len(y))
ax[0].plot(np.array(range(len(x[:plot_limit_x]))), x[:plot_limit_x])
ax[0].set_title("Iteration of the first(blue) and second (orange) coordinate of Wei
ax[1].plot(np.array(range(len(y[:plot_limit_y]))), y[:plot_limit_y])
minlen = min(len(x), len(y))
error = np.abs(x[:minlen] - y[:minlen])
print(x[-1])
print(y[-1])
ax[2].plot(np.array(range(len(error[:plot_limit]))), error[:plot_limit])
```

```
fig, ax = plt.subplots(figsize = (15,15))
ax.set_title("Weiszfeld iterating scheme", size = 20)
for i in range(len(x) + N):
    if i < len(x) - 1:
        if i == 0:
            ax.scatter(x[i,0], x[i,1], s = 20, c = "blue", label = "Iteration proces
            ax.annotate(f"Median of squared euclidean distance \n {x[i]}", x[i], xy
        else:
            ax.scatter(x[i,0], x[i,1], s = 20, c = "blue")
    elif i == len(x) - 1:
        ax.scatter(x[i,0], x[i,1], s = 200, c = "red", label = "Minimum weighted dis
        ax.annotate(f"{x[i]}", x[i],xytext = x[i] + np.array([-0.01,0.02]), size =
    else:
        j = i - len(x)
        if j == 0:
            ax.scatter(P[j,0], P[j,1], s = 200, c = "orange", label = "Weighed poin
            ax.annotate(f"{P[j]}, \n v = {v[j]:.2f}", P[j],xytext = P[j] + np.array
        else:
            ax.scatter(P[j,0], P[j,1], s = 200, c = "orange")
            ax.annotate(f"{P[j]}, \n v = {v[j]:.2f}", P[j],xytext = P[j] + np.array
ax.legend(prop={'size': 20})
ax.grid()

fig, ax = plt.subplots(figsize = (15,15))
ax.set_title("Gradient descent with backtracking iteration scheme", size = 20)
for i in range(len(y) + N):
    if i < len(y) - 1:
        if i == 0:
            ax.scatter(y[i,0], y[i,1], s = 20, c = "blue", label = "Iteration proces
            ax.annotate(f"Median of squared euclidean distance \n {y[i]}", y[i], xy
        else:
            ax.scatter(y[i,0], y[i,1], s = 20, c = "blue")
    elif i == len(y) - 1:
        ax.scatter(y[i,0], y[i,1], s = 200, c = "red", label = "Minimum weighted dis
        ax.annotate(f"{y[i]}", y[i],xytext = y[i] + np.array([-0.01,0.02]), size =
    else:
        j = i - len(y)
        if j == 0:
            ax.scatter(P[j,0], P[j,1], s = 200, c = "orange", label = "Weighed poin
            ax.annotate(f"{P[j]}, \n v = {v[j]:.2f}", P[j],xytext = P[j] + np.array
        else:
            ax.scatter(P[j,0], P[j,1], s = 200, c = "orange")
            ax.annotate(f"{P[j]}, \n v = {v[j]:.2f}", P[j],xytext = P[j] + np.array
ax.legend(prop={'size': 20})
ax.grid()

N = 130
P = np.random.uniform(0,1,N*2)
```

```
P = P.reshape((N,2))
v = np.random.uniform(0.1,1,len(P))
tol = 1E-6

t1W = time.time()
x, rel_error_x = Weiszfeld(P, v, tol)
dtW = time.time()- t1W

t1G = time.time()
y, rel_error_y = gradDescent(f, Df, P, v, tol = tol, iter_limit = 10000, backTrack =
dtG = time.time() - t1G

print(f" Weiszfeld needed {round(dtW, 4)} seconds, {len(x)} iterations to reach tole
print(f" Gradient descent needed {round(dtG,4)} seconds, {len(y)} iterations to reac

M = 100
tol = 1E-5
dtW = np.zeros(M)
iterW = np.zeros(M)
dtG = np.zeros(M)
iterG = np.zeros(M)
for i in range(M):
    N = 130
    P = np.random.uniform(0,1,N*2)
    P = P.reshape((N,2))
    v = np.random.uniform(0.1,1,len(P))
    t1W = time.time()
    x, rel_error_x = Weiszfeld(P, v, tol)
    dtW[i] = time.time()- t1W
    iterW[i] = len(x)

    t1G = time.time()
    y, rel_error_y = gradDescent(f, Df, P, v, tol = tol, iter_limit = 10000, backTra
    dtG[i] = time.time() - t1G
    iterG[i] = len(y)

plt.figure()
plt.title("Runtime difference of Weiszfeld and backtracking grad descent")
plt.xlabel("run #")
plt.ylabel("Weiszfeld runs y seconds faster")
plt.plot(np.array(range(M)), dtG - dtW)
plt.plot(np.array(range(M)), np.zeros(M), label = "0 axis", color = "black")
plt.plot(np.array(range(M)), np.full(M, np.mean(dtG - dtW)), label = "mean differenc
plt.plot(np.array(range(M)), np.full(M, np.mean(dtW)), label = "mean runtime of Wei
plt.plot(np.array(range(M)), np.full(M, np.mean(dtG)), label = "mean runtime of grad
plt.legend()

plt.figure()
plt.title("Iter step difference of Weiszfeld and backtracking grad descent")
```

```
plt.xlabel("run #")
plt.ylabel("Weiszfeld runs y iterations fewer")
plt.plot(np.array(range(M)), iterG - iterW)
plt.plot(np.array(range(M)), np.zeros(M), label = "0 axis", color = "black")
plt.plot(np.array(range(M)), np.full(M, np.mean(iterG - iterW)), label = "mean diffe
plt.plot(np.array(range(M)), np.full(M, np.mean(iterW)), label = "mean iter steps o
plt.plot(np.array(range(M)), np.full(M, np.mean(iterG)), label = "mean iter steps o
plt.legend()
```