Techniques in Operations Research Group Project

Power Optimisation in Wireless Sensor Networks

Chris Swan, Dominic Yew, Greta Di Lorenzo, Mark O'Brien

May 2021

1 Introduction

Wireless sensor networks are autonomous systems that measure the conditions of an environment. These networks contain sensors that each transmit information to a central relay that combines the locally sensed data. It then transmits this data to a base station for processing.

Sensors are often dropped in forestry areas to predict or detect the beginning of a fire. These sensors can measure temperature, humidity, smoke, and other environmental properties to determine whether conditions mean a high risk for a fire. It is common for sensors and relays to be battery operated, hence making it necessary to optimise the amount of power consumed. This highlights how the optimisation of power consumption in wireless networks is an important research problem in both the mathematical and engineering sciences.

In this paper, we analyse a wireless sensor network relay location problem, in which we are given a set X of n sensor locations in the plane $\mathbb{R}^2$. The aim is to determine the optimal location of a relay in this plane such that the power usage of the network is minimised. Each sensor only transmits at the minimum distance required to reach the relay, and the relay only transmits at the minimum distance required to reach all sensors. Ultimately, we are required to find a location $\mathbf{s} \in \mathbb{R}^2$ to position the relay in order to minimise the distance between the relay and all of the sensors. From this, we have the following minimisation problem:

$$\min \; P(\mathbf{s}) = \sum_{i=1}^{n} \|\mathbf{s} - \mathbf{x}_i\|^2 + \max_{x \in X} \|\mathbf{s} - \bar{\mathbf{x}}\|^2 \tag{1}$$

2 Background

2.1 Differentiability, Continuity & Convexity of the Objective Function

In beginning our analysis of the problem, we start by examining the two components of our objective function (1). The first component, $P_1 = \sum_{i=1}^{n} \|\mathbf{s} - \mathbf{x}_i\|^2$, is simply the sum of the square of the Euclidean distance between the relay and each of the sensors. It is easily shown that this function is both convex and continuously differentiable; for proof of the convexity see Section 2.1.2 below. As for being continuously differentiable, continuity is trivial and differentiability can be seen by the following:

Let $\mathbf{s} = (s_1, s_2)^T$ and $\mathbf{x}_i = (x_{i,1}, x_{i,2})^T$, where $x_{i,k}$ denotes the k^{th} element of the i^{th} vector in X.

Let $\Psi = \|\mathbf{s} - \mathbf{x}_i\|^2$. Then we have that:

$$
\begin{aligned}
\Psi &= \left(\sqrt{(s_1 - x_{i,1})^2 + (s_2 - x_{i,2})^2}\right)^2 \\
\frac{\partial \Psi}{\partial s_1} &= 2(s_1 - x_{i,1}) \cdot \frac{1}{2\sqrt{(s_1 - x_{i,1})^2 + (s_2 - x_{i,2})^2}} \cdot 2\sqrt{(s_1 - x_{i,1})^2 + (s_2 - x_{i,2})^2} \\
&= 2(s_1 - x_{i,1}) \\
\frac{\partial \Psi}{\partial s_2} &= 2(s_2 - x_{i,2}) \cdot \frac{1}{2\sqrt{(s_1 - x_{i,1})^2 + (s_2 - x_{i,2})^2}} \cdot 2\sqrt{(s_1 - x_{i,1})^2 + (s_2 - x_{i,2})^2} \\
&= 2(s_2 - x_{i,2}) \\
\frac{\mathrm{d}\Psi}{\mathrm{d}\mathbf{s}} &= 2(\mathbf{s} - \mathbf{x}_i)
\end{aligned}
$$

Therefore we have that

$$\nabla P_1(s) = 2\sum_{i=1}^{n}(\mathbf{s} - \mathbf{x}_i) \tag{2}$$

For the second component, we show that it is not C^1 with a counter example.

Consider a system with two sensors:

Let $X = \{(1,0), (0,1)\}$

Let $\mathbf{s}$ be an arbitrary point $(x, y) \in \mathbb{R}^2$.

$$
\begin{aligned}
\Rightarrow P(s) = P(x,y) &= \begin{cases} (x-1)^2 + (y-0)^2 + 2[(x+1)^2 + (y-0)^2], & x \leq 0 \\ (x+1)^2 + (y-0)^2 + 2[(x-1)^2 + (y-0)^2], & x > 0 \end{cases} \\
&= \begin{cases} 3x^2 + 2x + 2y^2 + 3, & x \leq 0 \\ 3x^2 - 2x + 2y^2 + 3, & x > 0 \end{cases}
\end{aligned}
$$

$$
\Rightarrow \frac{\partial P}{\partial x} = \begin{cases} 6x + 2, & x \leq 0 \\ 6x - 2, & x > 0 \end{cases}, \quad \frac{\partial P}{\partial y} = 4y,\ y \in \mathbb{R}
$$

Now we show that the following limit does not exist:

$$\lim_{(x,y)\to(0,0)} \frac{\partial P}{\partial x}$$

$$
\begin{aligned}
\lim_{x\to 0^-} \frac{\partial P}{\partial x} &= \lim_{x\to 0} (6x + 2) = 2 \\
\lim_{x\to 0^+} \frac{\partial P}{\partial x} &= \lim_{x\to 0} (6x - 2) = -2 \\
\Rightarrow \lim_{x\to 0^-} \frac{\partial P}{\partial x} &\neq \lim_{x\to 0+} \frac{\partial P}{\partial x}
\end{aligned}
$$

Therefore, the following limit does not exist:

$$\lim_{(x,y)\to(0,0)} \frac{\partial P}{\partial x}$$

$\therefore \frac{\partial P}{\partial x}$ is discontinuous at x = 0, $y \in \mathbb{R}$.

$\therefore \frac{\partial P}{\partial x}$ is not differentiable at x = 0.

$\therefore \frac{\partial P}{\partial s}$ is not differentiable at x = 0 for all $y \in \mathbb{R}$.

$\therefore P(s) \notin C^1$.

To prove the convexity of $P(s)$, we rely on two lemmas:

2.1.1 Lemma 1.1 (Sum of convex functions is convex)

Proof: Suppose $h = f + g$, where f and g are convex, then

$$\begin{aligned} h(\lambda x + (1-\lambda)x) &= f(\lambda x + (1-\lambda)x) + g(\lambda x + (1-\lambda x)) \\ &\leq \lambda(f(x) + g(x)) + (1-\lambda)(f(x) + g(x)) \end{aligned}$$

where $0 \leq \lambda \leq 1$

$\therefore$ by the definition of convex functions, h is convex.

2.1.2 Lemma 1.2 (All norms are convex)

Proof: If $x, y \in \mathbb{R}^n$ and $0 \leq \lambda \leq 1$, then

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

where the inequality follows from the triangle inequality.

The convexity of $P(s)$ follows immediately, as we can see that $P(s)$ is a sum of norms, which are all convex (by Lemma 1.2), hence $P(s)$ is convex. □

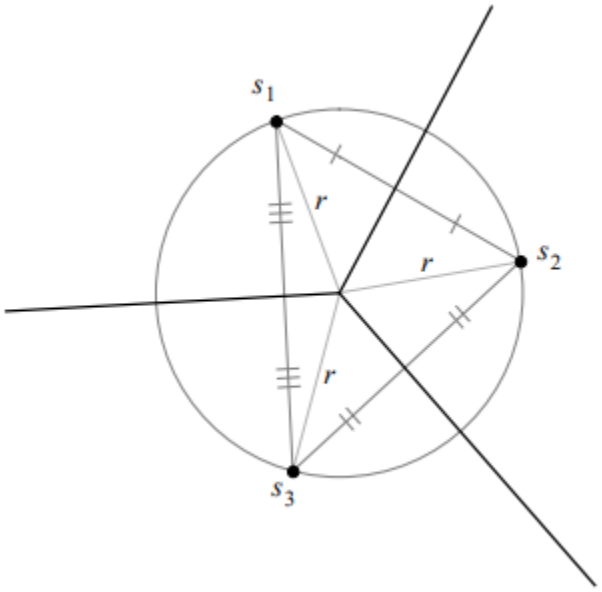

Figure 1: Voronoi diagram with 3 seeds

$|X| = 3;\ \mathbf{s}, X \in \mathbb{R}^{2\times n}$**: Lines consisting of all points at which the problem is not differentiable.**

The conditions above are satisfied by the bold lines in the Voronoi diagram shown in Figure 1 with three sensors placed at the seeds, s_1, s_2 and s_3. Since the NLP is non-differentiable where the max term flips, i.e., when it moves into the region consisting of all points of the plane further from one sensor than to any other. The solution holds for a system with n sensors, with the lines drawn at the boundary of each Voronoi cell.

2.2 A new model with a smooth objective function

Remodelling the original problem in (1) as a constrained l_2-penalty function, we have the following NLP:

$$
\begin{aligned}
P(s) &= \sum_{i=1}^{n} \|s - x_i\|^2 + \beta \\
\text{s.t.} \quad g_1(\mathbf{s}) &= \|\mathbf{s} - \mathbf{x}_1\|^2 \leq \beta \\
&\vdots \\
g_n(\mathbf{s}) &= \|\mathbf{s} - \mathbf{x}_n\|^2 \leq \beta \\
\text{where } \beta &= \max\{\|\mathbf{s} - \mathbf{x}_i\|^2\} \ \ \forall i \in \{1, n\}
\end{aligned}
$$

Restructuring this into an l_2-penalty function, we get:

$$\beta = \max\{\|\mathbf{s} - \mathbf{x}_i\|^2\} \ \ \forall i \in \{1, n\} \tag{3}$$

$$\mathcal{P}(s) = \sum_{i=1}^{n} \|s - x_i\|^2 + \beta + \frac{\alpha}{2} \sum_{i=1}^{n} g_i(\mathbf{s})_+^2 \tag{4}$$

$$\text{where } g_i(\mathbf{s}) = \|\mathbf{s} - \mathbf{x}_i\|^2 - \beta, \quad \forall i \in \{1, n\} \tag{5}$$

$$\text{and } g_i(\mathbf{s})_+ := \begin{cases} g_i(\mathbf{s}) & \text{if } g_i(\mathbf{s}) > 0 \\ 0 & \text{if } g_i(\mathbf{s}) \leq 0 \end{cases} \tag{6}$$

There are certain things that must be elucidated here.

- β (in (3)) acts as a constant for the objective function (4), as shall be seen in Section 3.
- α is the penalty function, which determines how much to penalise (6) when the model does not adhere to a particular constraint $g_i(\mathbf{s})$. This is one of the parameters which we shall tune in Section 4 and the penalty will grow with each iteration of our algorithm.
- One can observe that (6) is indeed continuous and differentiable. $g_i(\mathbf{s}) = 0$ when $\|\mathbf{s} - \mathbf{x}_i\| = \beta$, that is, when $\mathbf{x}_i$ is the furthest point away from $\mathbf{x}$.

2.2.1 Proof of convexity

From here we show that $P(\mathbf{s})$ is now continuous and differentiable for all real $\mathbf{s}$.

We have that

$$\nabla\mathcal{P}(\mathbf{s}) = 2\sum_{i=1}^{n}(\mathbf{s} - \mathbf{x}_i) + 2\alpha \sum_{i=1}^{n}(\mathbf{s} - \mathbf{x}_i)g_i(\mathbf{s})_+ \tag{7}$$

which, expressed as a vector in $\mathbb{R}^2$ is:

$$\nabla\mathcal{P}(\mathbf{s}) = \begin{bmatrix} 2\sum\limits_{i=1}^{n}(\mathbf{s}_1 - \mathbf{x}_{i,1}) + 2\alpha \sum\limits_{i=1}^{n}(\mathbf{s}_1 - \mathbf{x}_{i,1})g_i(\mathbf{s}_1)_+ \\ 2\sum\limits_{i=1}^{n}(\mathbf{s}_2 - \mathbf{x}_{i,2}) + 2\alpha \sum\limits_{i=1}^{n}(\mathbf{s}_2 - \mathbf{x}_{i,2})g_i(\mathbf{s}_2)_+ \end{bmatrix}$$

From here, we derive the Hessian matrix of $P(\mathbf{s})$:

$$\nabla^2\mathcal{P}(\mathbf{s}) = \begin{bmatrix} 2\alpha\sum\limits_{i=1}^{n}\Big(2(\mathbf{s}_1 - \mathbf{x}_{i,1})^2 + g_i(\mathbf{s}_1)_+\Big) + 2 & 0 \\ 0 & 2\alpha\sum\limits_{i=1}^{n}\Big(2(\mathbf{s}_2 - \mathbf{x}_{i,2})^2 + g_i(\mathbf{s}_2)_+\Big) + 2 \end{bmatrix} \tag{8}$$

From the above, we can see that as (8) is a diagonal matrix, its eigenvalues are the diagonal elements. For all inputs of $\mathbf{s} \in \mathbb{R}^2$, the eigenvalues will be positive.

Therefore, (8) is positive definite and by definition (4) is strongly convex. $\square$

2.3 KKT conditions

$$\begin{aligned}
\min\ P(\mathbf{s}) &= \sum_{i=1}^{n} \|\mathbf{s}-\mathbf{x}_i\|^2 + \beta \\
\text{s.t.}\quad g_1(\mathbf{s}) &= \|\mathbf{s}-\mathbf{x}_1\|^2 - \beta \le 0 \\
g_2(\mathbf{s}) &= \|\mathbf{s}-\mathbf{x}_2\|^2 - \beta \le 0 \\
&\vdots \\
g_n(\mathbf{s}) &= \|\mathbf{s}-\mathbf{x}_n\|^2 - \beta \le 0 \\
\text{where } \beta &= \max\{\|\mathbf{s}-\mathbf{x}_i\|^2\}\ \ \forall i \in \{1,n\}
\end{aligned}$$

$$L(x,\lambda) = P(s) + g_1(s)\lambda_1 + g_2(s)\lambda_2 + ... + g_n(s)\lambda_n$$

$$\Rightarrow L(x,\lambda) = L(x_1, x_2, ..., x_n, \lambda_1, \lambda_2, ..., \lambda_n)$$

$$= \sum_{i=1}^{n} \|s-x_i\|^2 + \beta + \lambda_1(\|s-x_1\|^2 - \beta) + \lambda_2(\|s-x_2\|^2 - \beta) + ... + \lambda_n(\|s-x_n\|^2 - \beta)$$

$$= \sum_{i=1}^{n} \|s-x_i\|^2 + \beta + \sum_{i=1}^{n} \lambda_i(\|s-x_i\|^2 - \beta)$$

$$\nabla_x L(x,\lambda) = \begin{bmatrix} 2(s-x_1) + 2\lambda_1(s-x_1) \\ 2(s-x_2) + 2\lambda_2(s-x_2) \\ \vdots \\ 2(s-x_n) + 2\lambda_n(s-x_n) \end{bmatrix}$$

KKTa:

$$\nabla_x L(x^*,\lambda^*) = 0$$

$\Rightarrow$

$$2(s-x_1^*) + 2\lambda_1^*(s-x_1^*) = 0$$

$$2(s-x_2^*) + 2\lambda_2^*(s-x_2^*) = 0$$

$$\vdots$$

$$2(s-x_n^*) + 2\lambda_n^*(s-x_n^*) = 0$$

$\Rightarrow$

$$2(s-x_i^*) + 2\lambda_i^*(s-x_i^*) = 0\ \ \forall_i \in \{1,n\}$$

KKTb:

1)

$$g(x^\star) \leq 0$$

$\Rightarrow$

$$\|s - x_1^\star\|^2 - \beta \leq 0$$
$$\|s - x_2^\star\|^2 - \beta \leq 0$$
$$\vdots$$
$$\|s - x_n^\star\|^2 - \beta \leq 0$$

$\Rightarrow$

$$\|s - x_i^*\|^2 - \beta \leq 0$$
$$\forall_i \in \{1, n\}$$

2)

$$\lambda^\star \geq 0$$

$\Rightarrow$

$$\lambda_1^\star \geq 0$$
$$\lambda_2^\star \geq 0$$
$$\vdots$$
$$\lambda_n^\star \geq 0$$

$\Rightarrow$

$$\lambda_i^\star \geq 0$$
$$\forall_i \in \{1, n\}$$

3)

$$\lambda_i^* g_i(x^*) = 0$$

$\Rightarrow$

$$\lambda_1^\star(\|\mathbf{s} - \mathbf{x}_1^*\|^2 - \beta) = 0$$
$$\lambda_2^\star(\|\mathbf{s} - \mathbf{x}_2^*\|^2 - \beta) = 0$$
$$\vdots$$
$$\lambda_n^\star(\|\mathbf{s} - \mathbf{x}_n^*\|^2 - \beta) = 0$$

$\Rightarrow$

$$\lambda_i^\star(\|\mathbf{s} - \mathbf{x}_i^*\|^2 - \beta) = 0$$
$$\forall_i \in \{1, n\}$$

3 Algorithms

We began our algorithm experimentation by exploring quasi-Newton methods, in particular the method of determining direction outlined in the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm [4] and using a similarly quasi-Newton inspired step size method developed by Jonathan Barzilai and Jonathan Borwein [2]. Whilst this algorithm was robust in its ability to find a solution to the problem, preliminary testing suggested that it was perhaps excessive for the task at hand.

Additionally, whilst the Barzilai-Borwein step size was formidable in the unconstrained setting, it did not fare as well with the penalty function. We should note that this may have been something that could be worked around and consequently improved, however, we elected to simplify our model to something that was more easily implemented.

Algorithm 1 A hybrid of BFGS and Barzilai-Borwein step size

Input: $k \leftarrow 0, \mathbf{s}^0 \in \mathbb{R}^d, X \in \mathbb{R}^{d\times n}, \lambda_0 > 0, H_0 \leftarrow I^d$, **Choose** ϵ

$d^0 \leftarrow -H_0 \nabla \mathcal{P}(\mathbf{s}^0)$

$\mathbf{s}^1 \leftarrow \mathbf{s}^0 + \lambda_0 d^0$

$k \leftarrow k + 1$

while $\|\nabla \mathcal{P}(\mathbf{s}^k)\| > \epsilon$ **do**

$\sigma^k \leftarrow x^k - x^{k-1}$

$g^k \leftarrow \nabla \mathcal{P}(\mathbf{s}^k) - \nabla \mathcal{P}(\mathbf{s}^{k-1})$

$r^k \leftarrow \frac{H_{k-1} g^k}{\langle \sigma^k, g^k \rangle}$

$H_k \leftarrow H_{k-1} + \frac{1+\langle r^k, g^k \rangle}{\langle \sigma^k, g^k \rangle} \sigma^k (\sigma^k)^T - [\sigma^k (r^k)^T - r^k (\sigma^k)^T]$

$d^k \leftarrow -H_k \nabla \mathcal{P}(\mathbf{s}^k)$

$\lambda_k \leftarrow \frac{\langle \mathbf{s}^k - \mathbf{s}^{k-1}, \nabla \mathcal{P}(\mathbf{s}^k) - \nabla \mathcal{P}(\mathbf{s}^{k-1}) \rangle}{\|\nabla \mathcal{P}(\mathbf{s}^k) - \nabla \mathcal{P}(\mathbf{s}^{k-1})\|^2}$

$\mathbf{s}^{k+1} \leftarrow \mathbf{s}^k + \lambda_k d^k$

$k \leftarrow k + 1$

end while

We turned to a recent step size innovation of Malitsky and Mishchenko [3]: a step size that adapts to the local curvature of $\mathcal{P}(\mathbf{s})$. Here, we make use of the fact the $\mathcal{P}(\mathbf{s})$ is convex and L-smooth, satisfying

$$\|\nabla \mathcal{P}(x) - \mathcal{P}(y)\| \leq L\|x - y\|, \quad \forall x, y. \tag{9}$$

We shall expand on the idea further in Section 4, but we note that in keeping the method of calculating the step size relatively simple, we avoided many of the programming pitfalls that we encountered in attempting to implement Algorithm 1 and its varieties. As we shall also elaborate on in Section 5, Algorithm 2 proved inexpensive for each iteration, and much more durable as we expanded the testing scenarios to include different distributions of sensors.

Algorithm 2 Steepest Descent with Adaptive step size

Input: $k \leftarrow 0, \mathbf{s}^0 \in \mathbb{R}^d, X \in \mathbb{R}^{d \times n}, \lambda_0 > 0, \theta_0 \leftarrow \infty$, **Choose** ϵ

$s^1 \leftarrow s^0 - \lambda_0 \nabla \mathcal{P}(\mathbf{s}^0)$

$k \leftarrow 1$

while $\|\nabla \mathcal{P}(\mathbf{s}^k)\| > \epsilon$ **do**

$\lambda_k \leftarrow \min\left\{\sqrt{1+\theta_{k-1}}\lambda_{k-1}, \frac{\|\mathbf{s}^k - \mathbf{s}^{k-1}\|}{2\|\nabla \mathcal{P}(\mathbf{s}^k) - \nabla \mathcal{P}(\mathbf{s}^{k-1})\|}\right\}$

$s^{k+1} \leftarrow s^k - \lambda_k \nabla \mathcal{P}(\mathbf{s}^0)$

$\theta_k \leftarrow \frac{\lambda_k}{\lambda_{k-1}}$

$k \leftarrow k + 1$

end while

4 Experimental Setup

As many of our team were more familiar with Python, we developed our algorithms concurrently in both Python and MATLAB, using GitHub to collaborate on code. This allowed us to ensure that dependencies on built-in functions and similar mechanisms did not bias or otherwise effect our model's performance. Hardware used varied amongst the group, but was generally a multi-core processor with $>$ 16GB RAM, although no parallel computing was used.

We elected to use several different distributions to generate data to test with our model. These included Uniform, Normal and Poisson distribution. We also experimented with different starting points so as to gauge performance. We created some custom distributions with generating functions, such as circular distributions and "blobs in a circle".

As we were with a convex function in $\mathcal{P}(\mathbf{s})$, we were able to utilise some features of convexity for choosing our initial parameters. In our preliminary testing, we found that we were choosing too large a step size, costing a significant amount of time for the algorithm to adjust. In this case, we had underestimated the magnitude of $\nabla \mathcal{P}(\mathbf{s})$, which was only amplified in larger scalings of the problem. Eventually, we followed a clue of Malitsky and Mishchenko [3] in approximating the Lipchitz constant L in (9). To do this, we used singular value decomposition of X and took the inverse of the largest singular value to be $\lambda = \frac{1}{L}$.

We can see this idea in how the step size of Algorithm 2 is calculated. Taking (9) and rearranging it, we get

$$\frac{1}{L} \leq \frac{\|x - y\|}{\|\nabla \mathcal{P}(x) - \mathcal{P}(y)\|}, \quad \forall x, y. \tag{10}$$

It is not stretch to see how Malitsky and Mishchenko's step size is used to approximate a 'local Lipschitz' constant.

Concerning (3), we had no end of trouble with β. We initially began with the idea of updating it at each iteration, however the folly of this soon became apparent. Eventually, we concluded that β could be updated at certain iterations, but only when $\max_{\mathbf{x} in X}\{\|\mathbf{s}_k - \mathbf{x}\|\} < \max_{\mathbf{x} in X}\{\|\mathbf{s}_k - \mathbf{x}\|\}$. This slight update turned out to be quite nuanced and had to occur within the $\nabla \mathcal{P}(\mathbf{s})$ function as parsing it caused some strange convergence results. Regarding the penalty function, α, after deliberation and experimental testing, we elected to use $\alpha = 2^k$. $\alpha = k$ and $\alpha = k^2$ were considered, however the fastest convergence resulted from the exponential option.

Aside from plotting the results and visually observing convergence, we had as a stopping condition $\|\nabla \mathcal{P}(\mathbf{s}^k)\| \leq$

ϵ, choosing ϵ to be 10^{-6} after some experimentation. Observing the decrease in $\|\nabla\mathcal{P}(\mathbf{s}^k)\|$ ensured that some minimum was being achieved by definition, however, we took measures such as manually testing $\max_{\mathbf{x} in X}\{\|\mathbf{s}_k - \mathbf{x}\|\}$ to ensure that this was indeed happening.

As a baseline for our testing, we compared our model against MATLAB's NLP unconstrained optimiser 'fminsearch', which uses the 'Nelder-Mead simplex direct search'. This is a purely functional method, requiring no gradient information.

5 Experimental Results

We found that with different initialisation positions of $\mathbf{s}$, the model quickly located the neighbourhood of the minimum. Depending upon the stated precision, it found $\|\nabla\mathcal{P}(\mathbf{s})\| \leq \epsilon$ in what seemed a reasonable number of iterations. Although this was always slightly less than 'fminsearch', the time taken was comparable, and the minimum found by 'fminsearch' did not have the same accuracy, however, it appears that this can be augmented with functional refinement.

Different distributions did not overly affect the model's performance. As we were using purely gradient methods, once working properly the model moved to a better neighbourhood at each iteration; the penalty function was rarely called upon until the model drew very close to the global minimum ($\|\nabla\mathcal{P}(\mathbf{s})\| < 10^{-3}$).

In general, our baseline 'fminsearch' found the neighbourhood of the minimum in around 2 to 3 less iterations than our algorithm did, and in around the same time. Comparing this will the results from Algorithm 2, we can see that whilst 'fminsearch' is certainly a useful inbuilt function, our model and Algorithm 2 are not bound to any particular language or software, and can be easily implemented in an open-source environment.

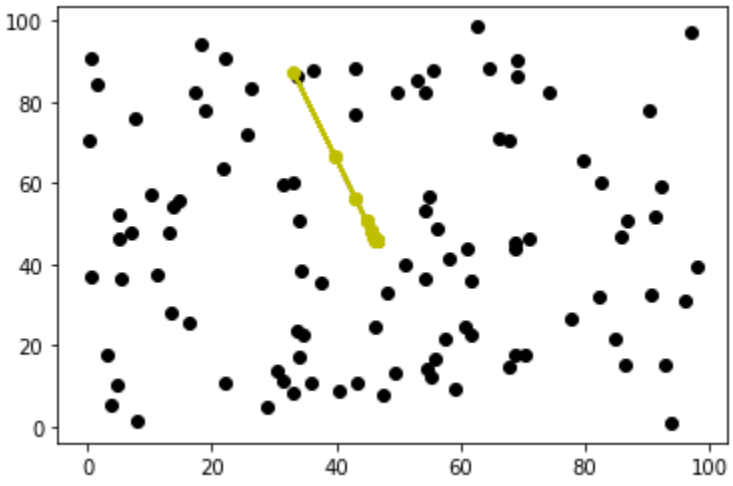

Figure 2: Algorithm 2 on a Uniform Distribution. Yellow line is the convergence of $\mathbf{s}$ to $\mathbf{s}^\star$ of the model, yellow points are iterations of $\mathbf{s}^k$, black points are $\mathbf{x} \in X$

Iteration (k)	$\mathbf{s}$ coordinates	$\|\nabla\mathcal{P}(\mathbf{s})\|$	Step Size (λ)	Time per iteration (seconds)
0	(33.095, 87.448)	8777.296	0.0	0.05252
1	(39.803, 66.537)	4388.6478	0.0025	0.03101
2	(43.155, 56.089)	2194.3239	0.0025	0.02501
3	(44.83, 50.866)	1097.162	0.0025	0.02501
4	(45.668, 48.254)	548.581	0.0025	0.02201
5	(46.087, 46.948)	274.2905	0.0025	0.02401
6	(46.297, 46.295)	137.1452	0.0025	0.02
7	(46.401, 45.969)	68.5726	0.0025	0.018
8	(46.454, 45.805)	34.2863	0.0025	0.017
9	(46.48, 45.724)	17.1432	0.0025	0.018
10	(46.493, 45.683)	8.5716	0.0025	0.018
11	(46.5, 45.662)	4.2858	0.0025	0.01801
12	(46.503, 45.652)	2.1429	0.0025	0.01801
13	(46.504, 45.647)	1.0714	0.0025	0.018
14	(46.505, 45.645)	0.5357	0.0025	0.018
15	(46.506, 45.643)	0.2679	0.0025	0.017
16	(46.506, 45.643)	0.1339	0.0025	0.017
17	(46.506, 45.642)	0.067	0.0025	0.017
18	(46.506, 45.642)	0.0335	0.0025	0.017
19	(46.506, 45.642)	0.0167	0.0025	0.017
20	(46.506, 45.642)	0.0084	0.0025	0.018
21	(46.506, 45.642)	0.0042	0.0025	0.018
22	(46.506, 45.642)	0.0021	0.0025	0.018
23	(46.506, 45.642)	0.001	0.0025	0.018
24	(46.506, 45.642)	0.0005	0.0025	0.01801
25	(46.506, 45.642)	0.0003	0.0025	0.018
26	(46.506, 45.642)	0.0001	0.0025	0.018
27	(46.506, 45.642)	0.0001	0.0025	0.018
28	(46.506, 45.642)	0.0	0.0025	0.01901

Table 1: Algorithm 2 on a Uniform Distribution

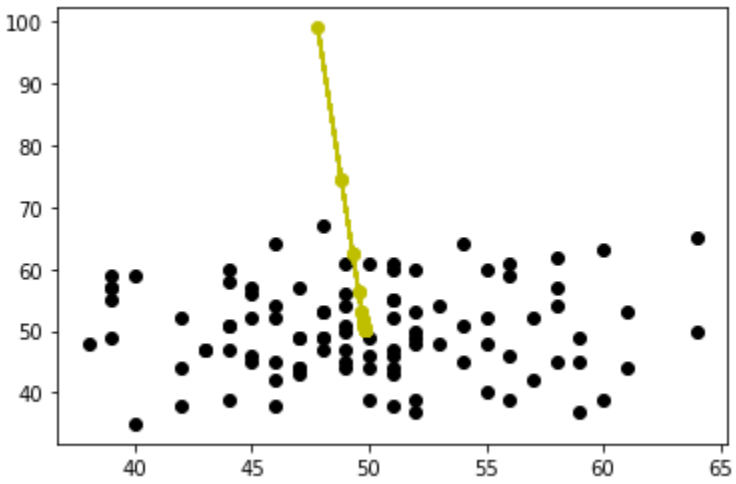

Figure 3: Algorithm 2 on a Poisson Distribution with Expected Rate = 50. Yellow line is the convergence of $\mathbf{s}$ to $\mathbf{s}^\star$ of the model, yellow points are iterations of $\mathbf{s}^k$, black points are $\mathbf{x} \in X$

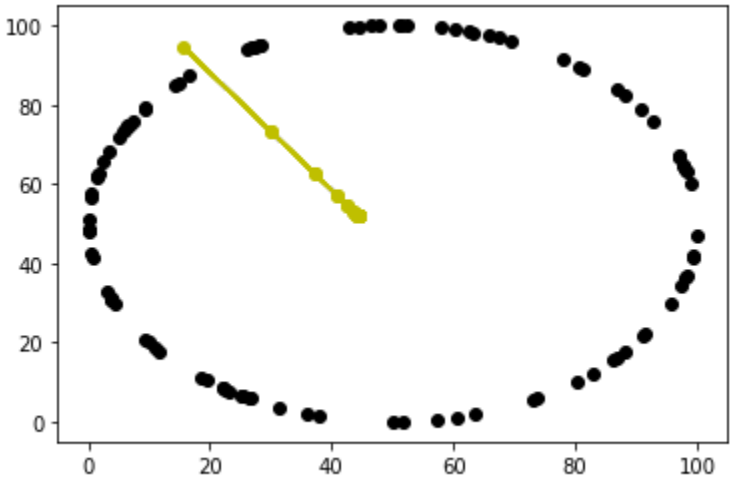

Figure 4: Algorithm 2 on a Circle Distribution. Yellow line is the convergence of $\mathbf{s}$ to $\mathbf{s}^\star$ of the model, yellow points are iterations of $\mathbf{s}^k$, black points are $\mathbf{x} \in X$

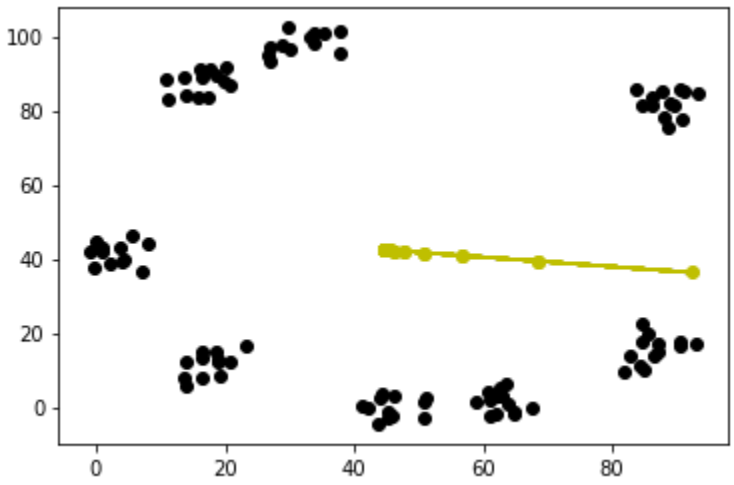

Figure 5: Algorithm 2 on a 'Blob Circle' Distribution. Yellow line is the convergence of $\mathbf{s}$ to $\mathbf{s}^\star$ of the model, yellow points are iterations of $\mathbf{s}^k$, black points are $\mathbf{x} \in X$

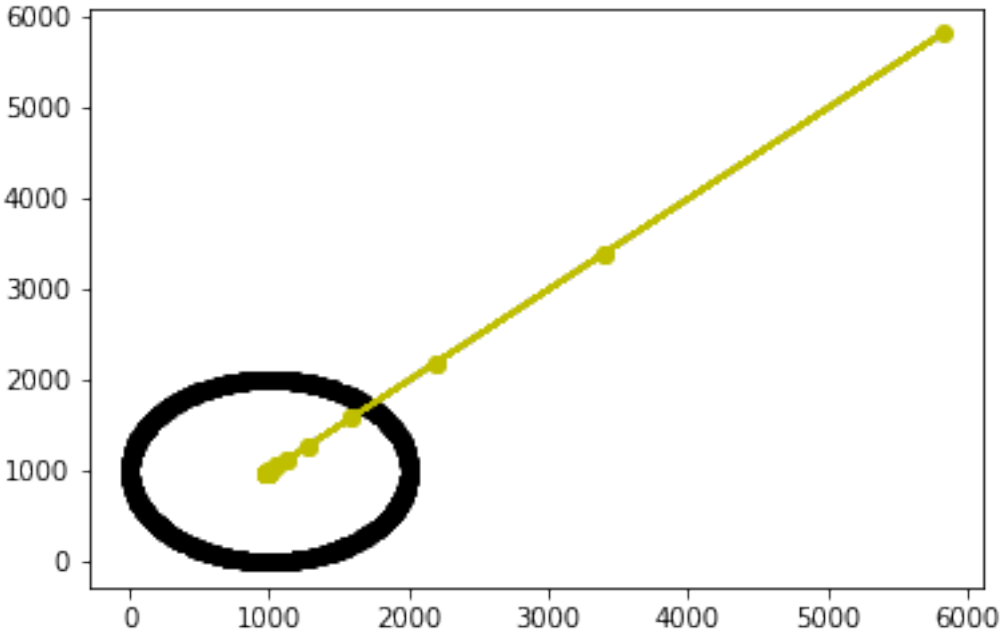

Figure 6: Algorithm 2 on a Circle Distribution. Yellow line is the convergence of $\mathbf{s}$ to $\mathbf{s}^\star$ of the model, yellow points are iterations of $\mathbf{s}^k$, black points are $\mathbf{x} \in X$

In examining our figures, we can can see in in all four that the convergence is visually similar for each type of distribution. $\mathbf{s}$ initialises on the boundary of the distribution and traverses in, following the direction of $-\|\nabla \mathcal{P}(\mathbf{s})\|$. We can see in each case the usefulness of our step size used in Algorithm 2, which adapts to the local gradient

without undue computational cost. This contrasts with a backtracking line search, for which the cost is significant and, as the scale of the distribution increases, impractical.[1]

Each of the Figures 2 to 5 are on distributions of 100 sensors, and each took the same amount of time and iterations to converge. When we scaled up the problem, as seen in Figure 6, to 1000 sensors, with $\mathbf{s}$ scaled up to a domain of $(0, 10000)$, we see that convergence occurs fairly quickly (in this case in 38 iterations). Scaling and field-size increase make for interesting visualisation, but show no change in convergence; the algorithm continues to converge a solution and we have that $\mathcal{P}(\mathbf{s}^k) - \mathcal{P}^\star \leq \mathcal{O}(\frac{1}{k})$, [3]

6 Discussion

Through extensive testing and experimentation, we have concluded that Algorithm 2 is a sturdy and durable algorithm, able to acclimatise itself to differing topology without excessive computation. The l_2-penalty method for converting a constrained non-linear program to an unconstrained one has distinct advantages in that we need not worry about the location of $\mathbf{s}^0$; this is in contrast to the log-barrier penalty method for example.

Furthermore, in our duel testing of the algorithm, we note that whilst MATLAB was significantly slower in running our model than Python, inbuilt functions and solvers as well as data visualisation made the user experience much easier.

In regard to the scope of the project, it is clear that optimisation of wireless sensor networks is certainly a problem that can be expanded to other scenarios: it bears a great deal of similarity to Linear Regression and Support Vector Machines in the machine learning world, albeit with the caveat of constraint programming.

[1] We did initially experiment with using a line search, using Armijo-Goldstein [1] and Wolfe conditions, however this quickly became impractical.

Iteration (k)	$\mathbf{s}$ coordinates	$\|\nabla\mathcal{P}(\mathbf{s})\|$	Step Size (λ)	Time per iteration (seconds)
0	(47.808, 99.018)	9775.823	0.0	0.015
1	(48.81, 74.579)	4887.9116	0.0025	0.02601
2	(49.31, 62.37)	2443.9558	0.0025	0.025
3	(49.56, 56.265)	1221.9779	0.0025	0.02501
4	(49.685, 53.212)	610.989	0.0025	0.02201
5	(49.747, 51.686)	305.4945	0.0025	0.023
6	(49.779, 50.923)	152.7472	0.0025	0.024
7	(49.794, 50.542)	76.3736	0.0025	0.01901
8	(49.802, 50.351)	38.1868	0.0025	0.019
9	(49.806, 50.255)	19.0934	0.0025	0.019
10	(49.808, 50.208)	9.5467	0.0025	0.019
11	(49.809, 50.184)	4.7734	0.0025	0.01901
12	(49.81, 50.172)	2.3867	0.0025	0.019
13	(49.81, 50.166)	1.1933	0.0025	0.021
14	(49.81, 50.163)	0.5967	0.0025	0.022
15	(49.81, 50.161)	0.2983	0.0025	0.022
16	(49.81, 50.161)	0.1492	0.0025	0.019
17	(49.81, 50.16)	0.0746	0.0025	0.02101
18	(49.81, 50.16)	0.0373	0.0025	0.019
19	(49.81, 50.16)	0.0186	0.0025	0.02601
20	(49.81, 50.16)	0.0093	0.0025	0.02601
21	(49.81, 50.16)	0.0047	0.0025	0.02601
22	(49.81, 50.16)	0.0023	0.0025	0.024
23	(49.81, 50.16)	0.0012	0.0025	0.02901
24	(49.81, 50.16)	0.0006	0.0025	0.03101
25	(49.81, 50.16)	0.0003	0.0025	0.024
26	(49.81, 50.16)	0.0001	0.0025	0.021
27	(49.81, 50.16)	0.0001	0.0025	0.02
28	(49.81, 50.16)	0.0	0.0025	0.019

Table 2: Algorithm 2 on a Poisson Distribution

References

[1] Larry Armijo, *Minimization of functions having Lipschitz continuous first partial derivatives*, Pacific Journal of Mathematics **16** (1966), no. 1, 1–3.

[2] Jonathan Barzilai and Jonathan M. Borwein, *Two-Point Step Size Gradient Methods*, IMA Journal of Numerical Analysis **8** (1988), no. 1, 141–148.

[3] Yura Malitsky and Konstantin Mishchenko, *Adaptive gradient descent without descent*, Proceedings of the 37th International Conference on Machine Learning (ICML) **119** (2020), 1–4.

[4] Jorge Nocedal and Stephen J. Wright, *Numerical optimization*, Springer, New York, 1999.