Methodology for solving constrained non-differentiable non-convex NLP

Matthew Calapai

October 13, 2023

Contents

1 Introduction

1.1 Optimisation

Optimisation is a sub-category of Operations Research that utilises a mathematical approach to solving problems. The idea behind optimisation problems is simply using algorithms to find minimum or maximum values to objective functions that contain at least one variable and a set of constraints. This process often involves implementing different algorithms to test the program. Such programs can be linear and non-linear, triggering different types of algorithms to be used. The motivation behind optimisation simply comes from the desire to optimise the output of certain systems. Optimisation is used in many sectors of work, such as logistics, production, finance and labour. Optimisation is a vital part of decision making as it can minimise the use of resources such as money, time and materials to consequently maximise efficiency and effectiveness.

1.2 Project Description

An online video sharing platform has asked for their their revenue generated through advertisements to be maximised by finding an optimal time allocation for each of their m ads over their total display time, T. The traffic visiting the platform is constant throughout the period T and the revenue generated by each ad, R_i, is proportional to the clicking rate of each ad, $V_i = kx_i^2$, where x_i for $i = 0, 1, ..., m$ represents the display time for advertisement i and $k > 0$. However the revenue for each ad cannot be greater than some maximum, b_i. Therefore $R_i = \min\{a_i V_i, b_i\}$, where $a_i > 0$. The display time for each ad also has to be at least some minimum, c_i.

2 Background

2.1 Mathematical Preliminaries

Optimisation can be categorised into two types: unconstrained and constrained optimisation. Before outlining these optimisation methods, some basic concepts necessary to understanding the following content are reviewed.

Definition 1 (Local and Global Minimum). *A global minimum of f is a point $x^* \in \mathbb{R}^n$ such that for every $x \in \mathbb{R}^n, f(x^*) \leq f(x)$ A local minimum of f is a point $x^* \in \mathbb{R}^n$ for which there exists ϵ such that for each x with $\|x - x^*\| < \epsilon$, we have $f(x^*) \leq f(x)$. Thus for each x "near" x^*, we have $f(x^*) \leq f(x)$. [1, p. 137].*

The *First-Order Necessary Optimality Condition* refers to the criteria for x^* to be a stationary point.

Definition 2 (Stationarity). *Let f be a C^1 function. We say that x^* is stationary if $\nabla f(x^*) = 0$. Stationary points include local minima, local maxima and saddle points. [1, p. 138].*

Throughout the solution, the local minima is of main interest. Minimality may be concluded by the *Second Order Condition for Minimality.*

Proposition 1 (Second Order Condition for Minimality). *Let f be C^2. If $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is positive definite then x^* is a strict local minimum of f. [1, p. 148].*

Minimality may also be concluded the through convexity of the objective function. First, note the definite of positive semi-definiteness.

Definition 3 (Positive Semi-Definiteness). *A matrix M is positive semi-definite when each of its eigenvalues are nonnegative, or alternatively, if $z^T Mz$ is nonnegative for every real vector $z \neq 0$. [6].*

Proposition 2 (Convexity). *Suppose f is C^2. Then f is convex if and only if $\nabla^2 f(x)$ is positive semi-definite for every $x \in \mathbb{R}^n$. [1, p. 144].*

Proposition 3 (Minima for Convex Functions). *If a C^1 function $f : \mathbb{R}^n \to \mathbb{R}$ is convex then*

- *Any local minimum is also a global minimum.*
- *$\nabla f(x^*) = 0$ if and only if x^* is a global minimum of f.*

[1, p. 145].

2.1.1 Constrained Optimisation

Constrained optimisation problems involve minimising or maximising an "objective function" that is subject to at least one constraint. The constraints may be equality and/or inequality constraints. Subjecting an objective function to such constraints creates a non-linear program (NLP). A general NLP looks like:

$$\begin{array}{ll} \min & f(\mathbf{x}) \\ \text{s.t.} & g_i(\mathbf{x}) \leq 0, \quad \text{for } i \in I \\ & h_j(\mathbf{x}) = 0, \quad \text{for } j \in J \end{array}$$

where $\mathbf{x} \in \mathbb{R}^n$ is a vector of decision variables, $I = \{1, \dots, p\}$, $J = \{1, \dots, q\}$, $f(\mathbf{x})$ is the objective function, $g(\mathbf{x})$ represents inequality constraints, $h(\mathbf{x})$ represents inequality constraints, and p and q represent the respective quantities of inequality and equality constraints.

When solving such constrained problems, the Lagrange function is used in finding any minima or maxima which is defined as:

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\eta}) = f(x) + \sum_{i=1}^{p} \lambda_i g_i(\mathbf{x}) + \sum_{j=i}^{q} \eta_j h_j(\mathbf{x})$$

As shown above, the Lagrange function features the objective function with each constraint multiplied by a Lagrange multiplier (λ for inequality constraints, η for equality constraints). Each optimal solution will correspond to a Lagrange multiplier. Upon change of the level of a constraint, the Lagrange multiplier will change its corresponding solution in proportion to the level of change within the NLP.

Constrained non-linear programs that only contain equality constraints are called equality-constrained NLP's. Hence the Lagrangian of these programs looks like:

$$L(\mathbf{x}, \boldsymbol{\eta}) = f(\mathbf{x}) + \sum_{j=1}^{q} \eta_j h_j(\mathbf{x})$$

To find local minima in equality-constrained NLP's, we have the following three cases:

Case 1: When $f(\mathbf{x})$ and $h(\mathbf{x})$ are C^1 functions, $\mathbf{x}^*$ is a local minimum when $\nabla_x L(\mathbf{x}^*, \boldsymbol{\eta}^*) = 0$, $h(\mathbf{x}^*) = 0$ and one of the constraint qualifications are satisfied. The constraint qualifications are: h is affine and the gradients of each h_j are linearly independent.

Case 2: When f is C^1 and convex, then x^* is a global minimum if h is affine and $\nabla_x L(\mathbf{x}^*, \boldsymbol{\eta}^*) = 0$.

Case 3: When f is C^2, then $\mathbf{x}^*$ is a local minimum if $\nabla^2_{xx}L(\mathbf{x}^*, \boldsymbol{\eta}^*)$ is positive definite on the critical cone, $\mathcal{C}(\mathbf{x}^*)$. This is known as the second-order sufficient condition. Conversely, if $\nabla^2_{xx}L(\mathbf{x}^*, \boldsymbol{\eta}^*)$ is negative definite on the critical cone, then $\mathbf{x}^*$ is a local maximum.

When the unconstrained problem contains both equality and inequality constraint, both λ and η terms exist in the Lagrangian. In such problems, λ_i and η_j will be referred to as as KKT multipliers. Now to find the optimal solutions to such NLP's, the KKT (Karush-Kuhn-Tucker) conditions are applied to the Lagrange function. Any point that satisfies the KKT conditions is a stationary point of the NLP by the following Theorem.

Theorem 1 (Global minima for convex programs). *Suppose (NLP) is a convex program. If x^* is a stationary point of (NLP), i.e., satisfies the KKT conditions, then x^* is a global minimiser. Conversely, if x^* is a local or global minimiser of (NLP) and a constraint qualification holds then x^* is also stationary. [1, p. 486]*

Furthermore, the second order sufficient conditions can be applied to determine the nature of each of these stationary points. The KKT conditions are as follows:

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

KKTb.
For $i = 1, ..., p$:

1. $g_i(x^*) \leq 0$
2. $\lambda_i \geq 0$
3. $\lambda_i^* g_i(x^*) = 0$

KKTc. $h(x^*) = 0$.

Just like Lagrange multipliers, KKT multipliers adjust the optimal solution by an amount that is proportional to the change in constraint. Furthermore, if an inequality constraint is inactive (the constraint is not equal to zero), the KKT multiplier for that constraint will be equal to zero. Conversely, if an inequality constraint is active (the constraint is equal to zero), it's KKT multiplier will be greater than zero.

2.1.2 Unconstrained Optimisation

Unconstrained optimisation problems involve minimising or maximising a continuously differentiable (denoted C^1) function that isn't subject to any constraints. The general formulation when $\mathbf{x} \in \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$ is a C^1 function, is

$$\min_x f(\mathbf{x}).$$

These types of problems can be solved by several methods that include the calculation of the gradient vector. The gradient vector of f with respect to x is defined by

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix}$$

Such methods that require the calculation of the gradient vector include the Steepest Descent Method and the Broyden–Fletcher–Goldfarb–Shanno (BFGS) method. There also exists methods that exploit second-order information of the objective function through the Hessian matrix, such as Newton's method. The Hessian matrix of a function $f(x)$ is defined by

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

While more computationally expensive, these second-order methods arrive at a solution faster. An outline of these methods is given below, although the focus is on the BFGS method for its use with the solution. Firstly, some mathematical preliminaries are reviewed.

Descent algorithms use first-order information to find a *descent direction*, and then proceed in that direction by a variable *step size* to find a lower point than the previous one, proceeding in this fashion until a minimum is found to a certain predetermined tolerance ϵ.

Definition 4 (Descent Direction). *Given $x \in \mathbb{R}^n$, a vector $d \in \mathbb{R}^n$ is a descent direction for f at x if*

$$\nabla f(x)^T d < 0$$

[1, p. 161].

Proposition 4 (Step Size). *If $x, d \in \mathbb{R}^n$ and d is a descent direction for f at x, then for sufficiently small $t > 0$,*

$$f(x + td) < f(x)$$

[1, p. 171].

Intuitively, the objective function is "sliced" at x in the descent direction d; we then minimise the single-variable optimisation problem that results.

The BFGS method and its "update" function are now outlined as the main unconstrained optimisation method used in this paper.

<u>BFGS Minimisation Method.</u> Reference: [1, p. 242].

Set $k = 0$, set $x^0 \in \mathbb{R}^n$, set $H_0 = I_n$
while *true* **do**
  Calculate $\nabla f(x^k)$ and set $d^k = -H_k \nabla f(x^k)$
  if $\|\nabla f(x^k)\| < \epsilon$ **then**
    Minimiser is found, so break the loop
  else
    Define a new function $q(t) = f(x^k + td^k)$
    Find a step length by:
    Solving $\arg\min_t q(t)$ or
    Using the Armijo-Goldstein and Wolfe conditions
    Finally, set $x^{k+1} = x^k + t_k d^k$ and calculate H_{k+1} via BFGS update
    Set $k = k + 1$
  end
end

Algorithm 1: BFGS Algorithm for Unconstrained Optimisation

<u>BFGS Update Method.</u> Reference: [1, p. 243].

Function *update(x: vector, f: function) : matrix* **H**

Calculate the following:

$s^k = x^{k+1} - x^k$

$g^k = \nabla f\left(x^{k+1}\right) - \nabla f\left(x^k\right)$

$r^k = H_k g^k / \left\langle s^k, g^k \right\rangle$

Set $H_{k+1} = H_k + \frac{1+\left\langle r^k, g^k \right\rangle}{\left\langle s^k, g^k \right\rangle} s^k \left(s^k\right)^T - \left[s^k \left(r^k\right)^T + r^k \left(s^k\right)^T\right]$

return H_{k+1}

end

Algorithm 2: BFGS Update Function

Descriptions of other unconstrained descent methods are omitted for brevity.

Comparison Of Methods

Other descent methods considered were Steepest Descent and Newton's Method. The main issues considered when choosing these methods was the rate of convergence and the computational efficiency of the algorithms. The method of Steepest Descent converges *linearly* to a minimiser [7], whereas Newton's Method converges *quadratically* to a minimiser [1, p. 213]. This is because the Steepest Descent method only requires first order information i.e $\nabla f(\mathbf{x})$, whereas Newton's method exploits second order information of the function, i.e $\nabla^2 f(\mathbf{x})$, the Hessian matrix.

This means that Newton's method is more computationally inefficient. Paired with the analytical solutions obtained by MATLAB's Symbolic Toolbox, such an implementation would be too computationally costly (see § 5.2) hence slower. Conversely, the method of Steepest Descent is more computationally efficient, but the slower rate of convergence implies that this method is, again, too slow to find a minimiser.

The BFGS method was used because it is a Quasi-Newton method: it approximates the Hessian matrix therefore reducing computational cost, but still converges superlinearly, only slightly slower than quadratic convergence. [1, p. 234].

2.1.3 Penalty Methods

As seen in § 2.1.1, finding the minimiser analytically using the KKT conditions requires one to consider all possible combinations of active and inactive constraints. For an NLP with n constraints, this is 2^n possible combinations that need to be considered, which becomes very computationally difficult and expensive. This is discussed in § 6.

Penalty methods address this issue. They "convert" constrained problems into unconstrained problems, to which descent methods discussed in § 2.1.2 can be applied. In general, the constrained NLP

$$\begin{array}{rll} \min & f(\mathbf{x}) & \\ \text{s.t.} & g_i(\mathbf{x}) \le 0, & \text{for } i \in I = 1, ..., p \\ & h_j(\mathbf{x}) = 0, & \text{for } j \in J = 1, ..., q \end{array}$$

is rewritten as the unconstrained NLP

$$\min_x P_\alpha(\mathbf{x}) = f(\mathbf{x}) + \alpha Q(\mathbf{x})$$

where $\alpha > 0$ is termed the *penalty parameter* and $Q(\mathbf{x})$ is the *penalty function*. Several penalty functions exist, though they adopt the general form

$$Q(\mathbf{x}) = \sum_{i \in I} \|\max\{g_i(\mathbf{x}), 0\}\|^q + \sum_{j \in J} \|h_i(\mathbf{x})\|^q \tag{1}$$

where q refers to the order of the norm. Since in the given problem $J \equiv \varnothing$, we only consider the sum involving $g_i(\mathbf{x})$.

Theorem 2 (Convergence of Minima for Penalty Functions). *Let f, g and h be C^1 functions. Assume a constraint qualification on h and g holds at x^*. Suppose x^k minimises P_{α_k} for each k, where $\alpha_k \to \infty$. If $\{x^k\}$ has a cluster point x^* and a constraint qualification holds at x^*, then x^* is (feasible and) stationary for (NLP). [1, p. 434].*

The above theorem says that as $\alpha \to \infty$, the optimal solutions found by minimising the penalty function converge to the optimal solutions found by minimising the original NLP.

Exact Penalty Method.

The *exact penalty method* lets $q = 1$ in Equation 1. In other words, our constrained NLP collapses to the problem

$$\min_{x} P_\alpha(\mathbf{x}) = f(\mathbf{x}) + \alpha \left(\sum_{i \in I} |\max\{g_i(\mathbf{x}), 0\}| \right)$$

The main issue surrounding the exact penalty method is that it is not C^1, however, we propose a smooth approximation to the penalty function that allows infinite differentiabilty in § 3.1. Such an approximation allows us to convert our nonconvex, nondifferentiable, constrained NLP into a continuous, differentiable, unconstrained NLP which is much simpler to solve.

Other Penalty Methods.

Other penalty methods include the l_1 penalty method which lets $q = 1$ in Equation 1, and the log-barrier penalty method. Formal definitions of these methods are omitted for brevity.

The Exact Penalty Method was chosen because the penalty parameter α must not diverge to infinity, but instead it must only be sufficiently large. This is advantageous in a numerical setting because it may become difficult/inefficient for MATLAB to work with very large numbers. Instead, our method allows discretionary selection of a smaller α, which provides the optimal solution. This is explored in § 5.

2.2 Project Formulation

2.2.1 Initial NLP

The initial NLP described by the project description in § 1.2 is

$$\begin{array}{ll} \max & f(\mathbf{x}) = \sum_{i=1}^{m} \min\{a_i k x_i^2, b_i\} \\ \text{s.t.} & g_1(\mathbf{x}) = c_1 - x_1 \leq 0 \\ & \vdots \\ & g_m(\mathbf{x}) = c_m - x_m \leq 0 \\ & g_{m+1}(\mathbf{x}) = \sum_{i=1}^{m} x_i - T \leq 0 \end{array}$$

Which can be written as the minimisation problem

$$\begin{array}{ll} \min & f(\mathbf{x}) = -\sum_{i=1}^{m} \min\{a_i k x_i^2, b_i\} \\ & = \sum_{i=1}^{m} \max\{-a_i k x_i^2, -b_i\} \\ \text{s.t.} & g_1(\mathbf{x}) = c_1 - x_1 \leq 0 \\ & \vdots \\ & g_m(\mathbf{x}) = c_m - x_m \leq 0 \\ & g_{m+1}(\mathbf{x}) = \sum_{i=1}^{m} x_i - T \leq 0 \end{array}$$

Firstly, we took the negative value of the objective function to convert the maximisation problem into a minimisation problem. The constraints g_1 to g_m explain that the display time of ad i must be at least a minimum ad time $c_i > 0$. The last constraint explains that the sum of all the display times from each ad must be no more than a total display time, T.

However due to the minimum function being undifferentiable, we chose to take the minimum function out of the objective function and convert it into several constraints that allowed the NLP to be differentiable. We then got the subsequent NLP in the next section, § 2.2.2.

2.2.2 Initial Thoughts

$$
\begin{array}{ll}
\min & f(\mathbf{x}) = -\sum_{i=1}^{m} a_i k x_i^2 \\
\text{s.t.} & g_1(\mathbf{x}) = c_1 - x_1 \leq 0 \\
& \vdots \\
& g_m(\mathbf{x}) = c_m - x_m \leq 0 \\
& g_{m+1}(\mathbf{x}) = a_1 k x_1^2 - b_1 \leq 0 \\
& \vdots \\
& g_{2m}(\mathbf{x}) = a_m k x_m^2 - b_m \leq 0 \\
& g_{2m+1}(\mathbf{x}) = \sum_{i=1}^{m} x_i - T \leq 0 \\
& \\
& a_i, c_i, k, T, m > 0
\end{array}
$$

So for the this final NLP, the inequality constraints g_1 to g_m represent the requirement that each of the display times for ad i have to be at least a minimum value of $c_i > 0$. Then the constraints from g_{m+1} to g_{2m} explain that the revenue of each ad can be no greater than a maximum ad revenue, $b_i > 0$. Lastly, the final constraint, g_{2m+1} represents that the sum of the all the display times for each ad can't be more than a maximum time, T. This NLP was used then used to study the optimisation problem and derive the KKT conditions in § 2.2.4.

2.2.3 Convexity

This non-linear program is not convex. This is because not all constraints are affine, and furthermore, the objective function is not convex, as shown below.

The objective function f is indeed C^2. The gradient of $f(\mathbf{x})$ from the NLP in § 2.2.2 is

$$
\nabla f(\mathbf{x}) = \begin{bmatrix} -2a_1 k x_1 \\ \vdots \\ -2a_m k x_m \end{bmatrix}
$$

Therefore the Hessian is

$$
\nabla^2 f(\mathbf{x}) = \begin{bmatrix} -2a_1 k & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & -2a_m k \end{bmatrix}
$$

This is a diagonal matrix and so the eigenvalues are accordingly $-2a_1k, \ldots, -2a_mk$. Since, for all i, $a_i > 0$ and $k > 0$, these eigenvalues are negative and so the Hessian of $f(\mathbf{x})$ is negative definite. In other words, it is a concave function. Therefore the program from § 2.2.2 is not convex.

2.2.4 KKT Conditions.

We now derive the KKT conditions that are necessary for identifying minima.

KKTa. Let $p = 2m+1$ be the number of inequality constraints.

$$\begin{aligned}
L(\mathbf{x}, \boldsymbol{\lambda}) &= f(\mathbf{x}) + \sum_{i=1}^{p} \lambda_i g_i(\mathbf{x}) \\
&= -\sum_{i=1}^{m} a_i k x_i^2 + \lambda_1(c_1 - x_1) + \cdots + \lambda_m(c_m - x_m) \\
&+ \lambda_{m+1}(a_1 k x_1^2 - b_1) + \cdots + \lambda_{2m}(a_m k x_m^2 - b_m) \\
&+ \lambda_{2m+1}\left(\sum_{i=1}^{m} x_i - T\right)
\end{aligned}$$

Then

$$\nabla_x L\left(\mathbf{x}, \boldsymbol{\lambda}\right) = \nabla f\left(\mathbf{x}\right) + \sum_{j=1}^{p} \lambda_j \nabla g_j\left(\mathbf{x}\right) = 0$$

This gives equations

$$\begin{aligned}
\Rightarrow \quad -2a_1 k x_1 - \lambda_1 + 2\lambda_{m+1} a_1 k x_1 + \lambda_{2m+1} &= 0 \\
-2a_2 k x_2 - \lambda_2 + 2\lambda_{m+2} a_2 k x_2 + \lambda_{2m+1} &= 0 \\
&\vdots \\
-2a_m k x_m - \lambda_m + 2\lambda_{2m} a_m k x_m + \lambda_{2m+1} &= 0
\end{aligned}$$

KKTb.

1.

$$\begin{gathered}
c_1 - x_1 \leq 0 \\
c_2 - x_2 \leq 0 \\
\vdots \\
c_m - x_m \leq 0 \\
a_1 k x_1^2 - b_1 \leq 0 \\
a_2 k x_2^2 - b_2 \leq 0 \\
\vdots \\
a_m k x_m^2 - b_m \leq 0 \\
\textstyle\sum_{i=1}^{m} x_i - T \leq 0
\end{gathered}$$

2.

$$\begin{gathered}
\lambda_1 \geq 0 \\
\lambda_2 \geq 0 \\
\vdots \\
\lambda_m \geq 0 \\
\lambda_{m+1} \geq 0 \\
\lambda_{m+2} \geq 0 \\
\vdots \\
\lambda_{2m} \geq 0 \\
\lambda_{2m+1} \geq 0
\end{gathered}$$

3.

$$\begin{aligned}
\lambda_1 (c_1 - x_1) &= 0 \\
\lambda_2 (c_2 - x_2) &= 0 \\
&\vdots \\
\lambda_m (c_m - x_m) &= 0 \\
\lambda_{m+1} \left(a_1 k x_1^2 - b_1\right) &= 0 \\
\lambda_{m+2} \left(a_2 k x_2^2 - b_2\right) &= 0 \\
&\vdots \\
\lambda_{2m} \left(a_m k x_m^2 - b_m\right) &= 0 \\
\lambda_{2m+1} \left(\sum_{i=1}^{m} x_i - T\right) &= 0
\end{aligned}$$

KKTc. There are no equality constraints.

3 Algorithms

3.1 Proposed Algorithm

3.1.1 Challenges

The main challenges in constructing and implementing an algorithm to solve the problem outlined in § 1.2 were that the NLP is nonconvex, nondifferentiable and constrained. We addressed this by using the exact penalty method to simplify the problem and address non-convexity. A new challenge was presented in the fact that both the objective function and the l_1 penalty function were not differentiable, however we overcame this by replacing all nondifferentiable functions with smooth approximations.

This allowed the nonconvex, nondifferentiable and constrained NLP to be converted into a convex, differentiable and unconstrained problem, which is much simpler to solve.

3.1.2 Mathematical Preliminaries

We begin by discussing some mathematical preliminaries to the algorithm; in particular, the smooth approximations. The exact penalty method involves taking the absolute value of the max function, two non-differentiable functions. We noted that

$$\max\{a, b\} = \frac{a + b + |a - b|}{2}$$

and so the only remaining issue was smoothly approximating the l_1 norm function $|\cdot|$. We approximate this with the *error function* denoted $\operatorname{erf}(x)$. The error function is defined as

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

This is clearly differentiable with

$$\frac{d}{dx} \operatorname{erf} x = \frac{2}{\sqrt{\pi}} e^{-x^2} \tag{2}$$

by definition. This differentiability allows us to use MATLAB's symbolic toolbox in computing analytical solutions to the gradient $\nabla f(x)$. This is explored further in § 3.1.5. Also, note that the error function is a transformation of the cumulative distribution function of the standard normal random variable. See the proof of Proposition 5 for more details.

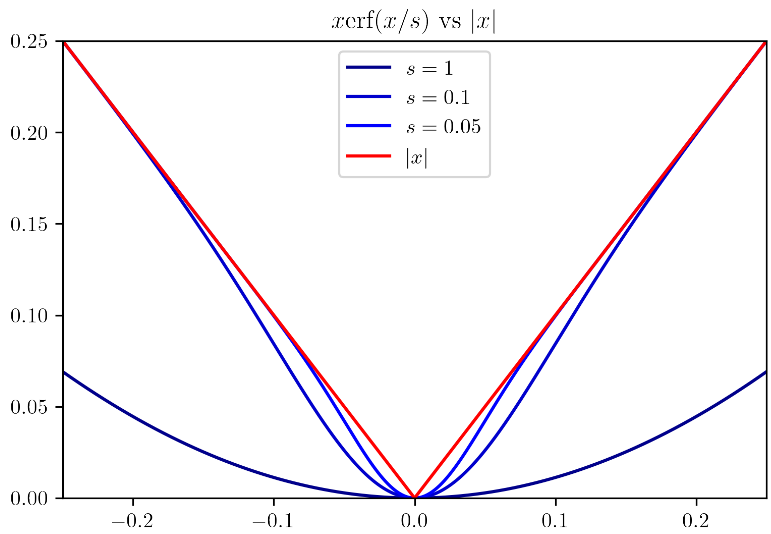

Figure 1: Approximation of l_1 norm

Proposition 5 (Approximation of l_1 norm). *The l_1 norm $|\cdot|$ is approximated by the smooth function $x\,\text{erf}(x/s)$ as $s \to 0$. This is illustrated in Figure 1.*

Proof. Let $Z \sim N(0,1)$ be a standard normal random variable and $\mathbf{P}$ its accompanying probability measure. Then its cumulative distribution function is defined by

$$\Phi(z) := F_Z(z) = \mathbf{P}(Z \le z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{-t^2/2}\, dt \tag{3}$$

This function is equipped with the properties of cumulative distribution functions. In particular, the properties $\lim_{z\to\infty} F_Z(z) = 1$ and $\lim_{z\to-\infty} F_Z(z) = 0$. Furthermore, $F_Z(t)$ is continuous and differentiable since Z is a continuous random variable.

Notice that can rewrite (3) as

$$\Phi(x) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$

It follows that

$$\text{erf}(x) = 2\Phi(x\sqrt{2}) - 1.$$

Since $\Phi(x)$ is continuous,

$$\lim_{s\to 0^+} \text{erf}\left(\frac{x}{s}\right) = \lim_{s\to 0^+} 2\Phi\left(\frac{x}{s}\sqrt{2}\right) - 1 = 1$$

and

$$\lim_{s\to 0^-} \text{erf}\left(\frac{x}{s}\right) = \lim_{s\to 0^-} 2\Phi\left(\frac{x}{s}\sqrt{2}\right) - 1 = -1$$

Therefore, as $s \to 0$,

$$\text{erf}\left(\frac{x}{s}\right) \to \frac{|x|}{x} = \begin{cases} 1 & x \ge 0 \\ -1 & x < 0 \end{cases}$$

and so we conclude that $x\,\text{erf}(x/s) \to |x|$ as $s \to 0$. $\square$

The result is a continuously differentiable function since x is continuously differentiable and $\text{erf}(x/s)$ is continuously differentiable.

This allows us to approximate the $\max\{a,b\}$ function with the following:

$$\max\{a,b\} = \frac{a+b+|a-b|}{2} \approx \frac{a+b+(a-b)\,\mathrm{erf}((a-b)/s)}{2} \text{ as } s \to 0$$

Hence allowing us to rewrite our previously nondifferentiable NLP as a differentiable one.

3.1.3 Algorithm Outline

The discussion above allows us to approximate the original, nondifferentiable NLP (§ 2.2.1) and penalty function (Equation 1) with differentiable functions. Let

$$\xi(x) := x\,\mathrm{erf}(x/s)$$

and

$$\zeta(x,b) := \frac{x+b+\xi(x)}{2}$$

Then the original NLP becomes

$$\begin{array}{rl} \min & f(\mathbf{x}) = \sum_{i=1}^{m} \xi(-a_i k x_i^2, -b_i) \\ \text{s.t.} & g_1(\mathbf{x}) = c_1 - x_1 \leq 0 \\ & \vdots \\ & g_m(\mathbf{x}) = c_m - x_m \leq 0 \\ & g_{m+1}(\mathbf{x}) = \sum_{j=1}^{m} x_i - T \leq 0 \end{array}$$

Let $\varphi(x,b) = (\xi \circ \zeta)(x,b)$. Then the penalty function $P_\alpha(\mathbf{x})$ becomes

$$P_\alpha(\mathbf{x}) = \sum_{i=1}^{m} \zeta(-a_i k x_i^2, -b_i) + \alpha \left[\sum_{i=1}^{m} \varphi(c_i - x_i, 0) + \varphi\left(\sum_{i=1}^{m} x_i - T, 0 \right) \right] \tag{4}$$

This penalty function is continuous and differentiable, and hence can be minimised via BFGS. The minimisation problem is now

$$\min_x P_\alpha(\mathbf{x})$$

3.1.4 Simple Example

This problem is easily visualised in the univariate case when $m = 1$. The NLP is then

$$\begin{array}{rl} \min & f(x) = \xi(-a_i k x^2, -b) \\ \text{s.t.} & g_1(x) = c - x \leq 0 \\ & g_2(x) = x - T \leq 0 \end{array}$$

For illustration purposes, let $c = 0.5$, $a = k = 1$, $b = 2.5$ and $T = 1.2$. Then the penalty function $P_\alpha(\mathbf{x})$ is

$$P_\alpha(\mathbf{x}) = \zeta(-x^2, -b) + \alpha\left[\varphi(0.5 - x, 0) + \varphi(x - 1.2, 0)\right]$$

and so the converted NLP is

$$\min_x P_\alpha(\mathbf{x})$$

This program is visualised in Figure 2.

The original problem is clearly nonconvex, with the feasible region being the overlap of light blue and light red regions in the figure. The penalty function P (in red) clearly has a global minimum at $x = 1.2$, and we can therefore use descent algorithms such as BFGS to find the minimiser.

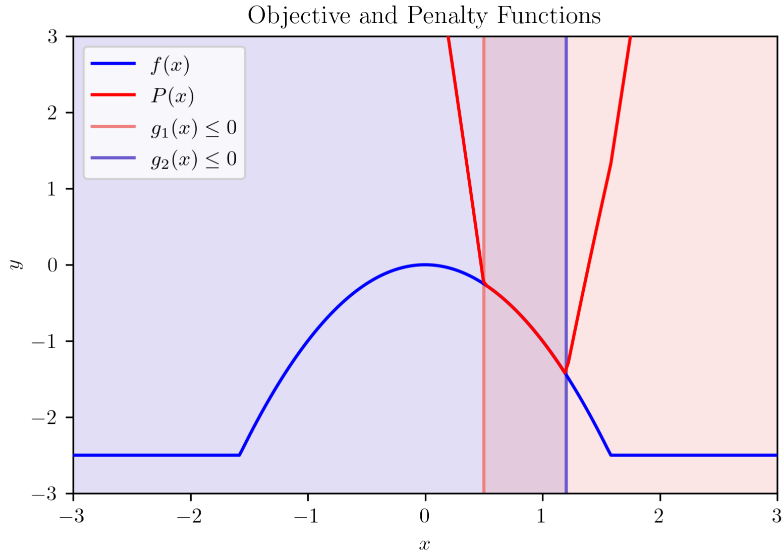

Figure 2: Simple Example of Penalty Function

3.1.5 Implementation in MATLAB

We used MATLAB's symbolic toolbox to assist in calculating the gradient of the penalty function. Since the error function is differentiable by Equation 2, we can use MATLAB's `gradient` function to analytically evaluate the gradient of the penalty function at each step, therefore obtaining accurate solutions.

We implemented our algorithm in MATLAB by first approximating all expressions involving the max and absolute value functions, forming the penalty function via MATLAB's Symbolic Toolbox, and finally implementing a custom BFGS algorithm that finds minimisers given a symbolic function. For a high level overview of this implementation, see Figure 3.

3.2 Alternatives

There are several pre-existing optimisation functions that can be found in MATLAB's Optimisation Toolbox. Such algorithms that are used to solve minimisation problems include the `fminunc` algorithm and the `fmincon` algorithms. The types of `fmincon` algorithms that would prove useful to our NLP include the active-set, SQP and the interior-point algorithms.

The default `fminunc` algorithm uses a variation of the Quasi-Newton method that involves taking various approximations to find the minimum of an unconstrained problem. The algorithm can be used in two ways. The function to be minimised and a starting point can be input, or, the minimisation problem can be input. Either will return the minimum value for the unconstrained problem. [2].

The interior-point algorithm is the default algorithm for the `fmincon` function. It solves constrained minimisation problems by solving a sequence of approximate minimisation problems. It can handle small to large scale problems and satisfies all constraints at each iteration. It can also take steps that fail. In this situation, the algorithm back-tracks and takes a smaller step, until it reaches a step that is successful. [3].

The SQP (sequential quadratic programming) algorithm is used for small to medium sized problems. In most cases, this algorithm is faster than the interior-point algorithm. It works by taking approximations of the Hessian of the Lagrangian at each iteration to create quadratic programming sub-problems which are then used to give a search direction for a line search

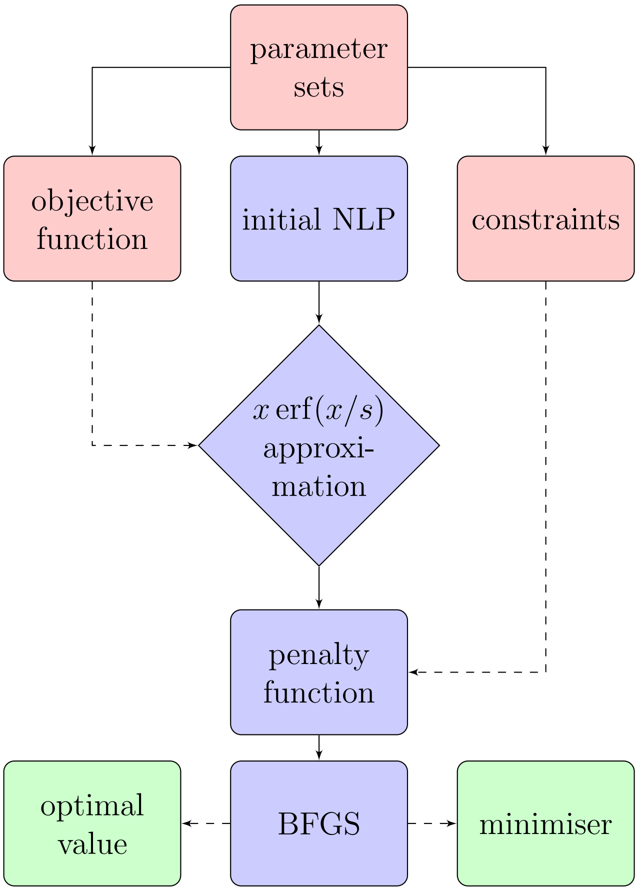

Figure 3: High Level Overview of Algorithm

method [3] that will ultimately find the solution to the minimisation problem. Also, just like the interior-point method, the SQP algorithm can take steps that fail. [3].

The active-set algorithm is very similar to the SQP algorithm. It is also used for small to medium scale problems. However there are a few key differences. Firstly, the active-set algorithm is slightly slower in most cases than the SQP algorithm. This is because SQP uses a more efficient set of linear algebra operations. Also, unlike the SQP algorithm, the active-set algorithm can't take steps that fail and also takes larger steps. Lastly, the SQP takes every iterative step in the feasible region, whereas the active-set algorithm takes steps that may not be strictly in the feasible region. [3].

4 Experimental Setup

A computational study was conducted to explore the time complexity and efficacy of our algorithm as compared to MATLAB's built-in optimisation functions. This was done via a "comparative" study and a speed at scale study.

Comparative Study.

Firstly, we ran all algorithms on randomly generated sets of initial parameters $\{a_i\}$, $\{b_i\}$, $\{c_i\}$ where $i = 1, \ldots, m$, and randomly generated values of T and k. We also randomly generated initial points for each of the search algorithms. This was done `nruns` times for a particular m value, and the results were averaged. This general setup was repeated for several values of m.

We compare our algorithm with several alternatives. Firstly, we solve the NLP by minimising our approximated penalty function (Equation 1) with our custom implementation of the BFGS algorithm, designed to work with MATLAB's Symbolic Toolbox. We then solve the NLP by minimising our approximated penalty function, however finding the minimiser with MATLAB's `fminunc` function. Finally, we compare these results with that of MATLAB's `fmincon`, which is able to solve the original NLP outlined in § 2.2.

It should be noted that, due to the min function in the original NLP and the non-binding nature of the last constraint, there can be several minimisers to the NLP given the same set of parameters, depending on T. In particular, the algorithms should find the same minimiser if T is small (since in this case, the last constraint is more likely to be active) and different minimisers if T is larger. Furthermore, since we used an approximation to solve the NLP, we may achieve different optimal solutions. This is explored in § 5.3.

In any case, since the original problem is a maximisation of revenue, we should just take those solutions $\mathbf{x}$ that lead to the largest value of $-f(\mathbf{x})$. The general outline of the computational setup is given below.

Speed at Scale Study.

To explore how well our algorithm works with large m, as compared to MATLAB's solutions, we run a "speed at scale" study by fixing m and taking several randomly generated parameters sets $\{a_i\}$, $\{b_i\}$, $\{c_i\}$ where $i = 1, \ldots, m$ and several randomly generated initial $\mathbf{x}$, recording the time taken to reach the solution, and averaging these times. We repeat this for increasing values of m.

4.1 Computational Study Setup

We begin the computational study by reviewing the results of five different algorithms (including our own, denoted `PBFGS`) on one single randomly generated parameter set. This allows us to more translucently investigate the performance of our algorithm when compared to MATLAB's own solutions.

We then review our algorithm's time complexity by solving the NLP over several randomly generated parameter sets and initial points x_0, and averaging the time taken and the number of iterations.

Computational Study Algorithm

Set m, $nruns$, $nparamsets$, T and k
For a, b and c, generate array of size $nparamsets \times m$ and populate with random points
For x0, generate array of size $nruns \times m$ and populate with random points
for *i = 1:5* **do**
    for *j = 1:nparamsets* **do**
        Select parameter set
        Set unique minimiser count = 0
        Create empty array for minimiser points, *minimisers* **for** *j = 1:nruns* **do**
            Set initial x point
            switch *i* **do**
                case *i=1* **do**
                    Run our algorithm: approximated penalty function optimised with our BFGS implementation
                end
                case *i=2* **do**
                    Run: approximated penalty function optimised with MATLAB's `fminunc`
                end
                case *i=3* **do**
                    Run: MATLAB's fmincon with `active-set` algorithm
                end
                case *i=4* **do**
                    Run: MATLAB's fmincon with `sqp` algorithm
                end
                case *i=5* **do**
                    Run: MATLAB's fmincon with `interior-point` algorithm
                end
                For each of the above cases, record x^*, $f(x^*)$, and the number of iterations
                End timer
                if *Minimiser already exists* **then**
                    Increase unique minimiser count by 1
                else
                    Append to unique minimiser list *minimisers*
                end
            end
            Average the timer recordings and the number of iterations for each algorithm
        end
    end
end

Algorithm 3: Computational Study Setup

4.1.1 Hardware

This computational study was run on a 16-inch 2019 Apple MacBook Pro with 2.6GHz 6-Core Intel Core i7 and 16GB DDR4 RAM.

5 Experimental Results

5.1 Single Parameter Set

By testing a single, randomly generated parameter set on many different initial x points (and averaging the results) we found the following for our algorithm and each of the algorithms discussed above.

```
Testing values m = 6, k = 1 and T = 10 for 1 parameter sets with 100 random initial points each set.
parameters:
a = [2.035491, 6.481599, 2.200519, 3.165307, 3.944252, 8.732237, ]
b = [6.994812, 5.870460, 1.261124, 7.603735, 4.554550, 8.218424, ]
c = [0.254421, 0.056885, 0.866649, 0.221029, 0.404989, 0.316096, ]
PBGFS
F max 34.503105 at x = [1.894299, 3.624483, 0.886495, 1.550006, 1.074585, 0.970134, ];
found for 99 times, avg iterations of 1.171717 and avg time of 2.786816
F max 33.255536 at x = [2.769018, 0.844531, 1.045617, 1.549907, 2.983454, 1.083953, ];
found for 1 times, avg iterations of 5.000000 and avg time of 1.652431
----------------------------------------------------------------------------------------------------
fminunc
parameters:
F max 34.503105 at x = [1.887923, 1.156283, 1.018741, 1.612227, 1.340563, 1.291358, ];
found for 91 times, avg iterations of 1.120879 and avg time of 0.005492
F max 32.691368 at x = [2.338264, 1.185321, 2.841104, 1.549908, 1.472284, 0.856553, ];
found for 1 times, avg iterations of 10.000000 and avg time of 0.009102
F max 29.618086 at x = [2.317607, 1.886762, 1.482609, 1.549907, 1.674105, 0.617847, ];
found for 1 times, avg iterations of 8.000000 and avg time of 0.005706
F max 27.638445 at x = [3.462882, 1.773674, 0.867317, 0.483210, 1.633948, 1.778972, ];
found for 1 times, avg iterations of 2.000000 and avg time of 0.005870
F max 32.173845 at x = [1.932020, 0.973869, 1.131499, 1.290868, 2.758940, 1.912807, ];
found for 1 times, avg iterations of 4.000000 and avg time of 0.007454
F max 31.420112 at x = [1.853760, 1.352658, 1.579416, 1.195080, 1.994650, 1.660118, ];
found for 1 times, avg iterations of 6.000000 and avg time of 0.007209
F max 34.044247 at x = [1.853760, 0.923842, 1.149457, 1.989036, 1.205092, 0.963004, ];
found for 1 times, avg iterations of 6.000000 and avg time of 0.006854
F max 34.173934 at x = [2.396306, 0.951689, 0.776093, 3.220006, 1.035024, 1.395707, ];
found for 1 times, avg iterations of 5.000000 and avg time of 0.007523
F max 33.473023 at x = [1.853760, 0.864169, 1.229622, 1.900905, 2.939452, 1.229556, ];
found for 1 times, avg iterations of 13.000000 and avg time of 0.011265
F max 30.082253 at x = [1.124518, 1.157168, 1.556039, 2.024884, 2.769440, 1.367954, ];
found for 1 times, avg iterations of 6.000000 and avg time of 0.007475
----------------------------------------------------------------------------------------------------
fmincon-activeset
parameters:
F max 34.503105 at x = [2.218018, 1.144787, 1.368657, 1.666724, 1.974986, 1.626828, ];
found for 96 times, avg iterations of 1.031250 and avg time of 0.010255
F max 33.677290 at x = [2.513591, 0.894154, 1.405174, 1.535833, 2.392935, 1.258313, ];
found for 1 times, avg iterations of 4.000000 and avg time of 0.047063
F max 20.623502 at x = [0.729789, 3.675625, 2.386398, 0.933070, 1.511078, 0.764040, ];
found for 1 times, avg iterations of 1.000000 and avg time of 0.011595
F max 22.849780 at x = [0.764921, 0.056885, 1.513121, 3.158349, 2.374406, 2.132318, ];
found for 1 times, avg iterations of 1.000000 and avg time of 0.004469
F max 32.165767 at x = [3.197164, 0.996116, 1.793963, 1.289879, 1.687653, 1.035225, ];
found for 1 times, avg iterations of 3.000000 and avg time of 0.005031
----------------------------------------------------------------------------------------------------
fmincon-sqp
parameters:
F max 34.503105 at x = [2.257302, 1.522752, 1.131772, 2.309819, 1.770021, 1.008334, ];
found for 100 times, avg iterations of 1.050000 and avg time of 0.004756
----------------------------------------------------------------------------------------------------
fmincon-interiorpoint
parameters:
F max 34.503105 at x = [2.549346, 1.079676, 1.482059, 1.718248, 1.486321, 1.145225, ];
found for 100 times, avg iterations of 1.110000 and avg time of 0.013139
```

The above terminal output is for § 8.2.6 with `disp_comparison` set to `true`.

Preliminary results and observations tell us that our `PBFGS` algorithm performs well in finding the function's optimal value, 34.503. Notice also that our algorithm found this unique solution 99 times out of 100 iterations of different initial $\mathbf{x}_0$ points. This performance is on par with MATLAB's `fmincon-sqp` and `fmincon-interiorpoint` algorithms, and much better than MATLAB's `fminunc` and `fmincon-activeset` algorithms. It may be that `fminunc` is "compounding" approximations, since the penalty function being minimised is approximated, and `fminunc` approximates the objective function to find the minimum (as outlined in § 3.2), therefore increasing the rate of false solutions. Further testing in § 8.1 shows that our `PBFGS` algorithm is highly effective at finding the correct minimiser.

5.2 Speed

We explore the efficiency of our algorithm, as compared to the other discussed algorithms, by taking many randomly generated parameter set $\{a_i\}, \{b_i\}, \{c_i\}$ over many initial points x_0 for descent, and averaging the time taken to find a minimiser.

One major drawback of our algorithm was its slow speed. As discussed in § 6, this is because our algorithm used MATLAB's Symbolic Toolbox to analytically find the derivative of the penalty function(Equation 4), whereas MATLAB's implementations use approximations. This is further discussed in § 3.2 and § 6. Table 1 summarises the performance of our algorithm, compared to the alternatives, for increasing m. We tested 5 randomly generated parameter sets, and for each of these parameter sets, 10 randomly generated initial $\mathbf{x}$ points (to ease computation times). The table entries represent the average time to reach a solution, in seconds.

m	`PBFGS`	`fminunc`	`fmincon-actvset`	`fmincon-sqp`	`fmincon-intpnt`
1	0.08127	0.0014252	0.0040226	0.0023068	0.0051165
2	0.15639	0.0020614	0.0055606	0.0040805	0.0053649
3	0.24685	0.0013552	0.0033699	0.0027409	0.0042453
4	0.63553	0.001857	0.0052084	0.003093	0.0064255
5	1.5405	0.0031013	0.0063956	0.0051819	0.010703
6	1.9587	0.0037001	0.0075659	0.0043083	0.008712
7	5.239	0.0064016	0.013618	0.0061286	0.017911
8	13.6881	0.010688	0.021546	0.010649	0.051049
9	18.6636	0.014678	0.024875	0.013785	0.054965

Table 1: Speed test for $m = 1, \ldots, 7$ with $T = 10$ and $k = 1$.

The above table represents the output of § 8.2.6 with `disp_comparison` set to `false`. These results are graphed in Figure 4.

Figure 4 shows that the time complexity of our algorithm is similar to that of `fmincon-inputpoint`. Clearly, `fminunc`, `fmincon-activeset` and `fmincon-sqp` perform much better at scale, showing what seems to be a $\mathcal{O}(n \log n)$ time complexity (although we should run more tests to confirm this). Our algorithm shows what seems to be $\mathcal{O}(2^n)$ complexity. Again, more testing should be done to confirm this.

5.3 Results of Approximation

The ability for `PBFGS` to find the correct minimum (for the majority of times) should improve as the approximation parameter s (see Proposition 5) increases. We tested this through experimentation.

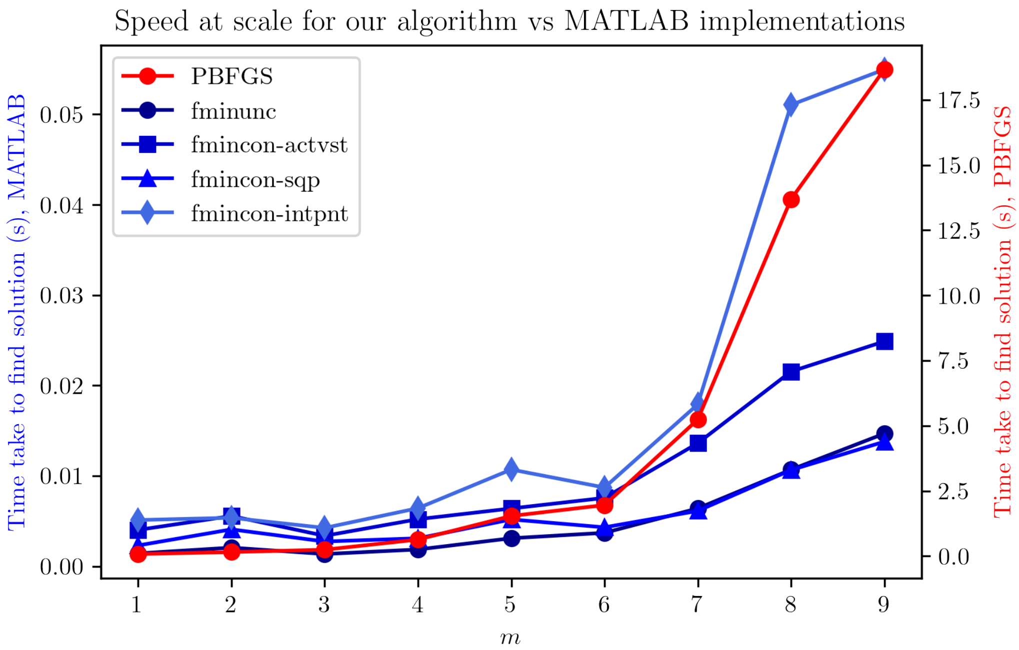

Figure 4: Speeds with increasing m

Using the comparative computational study from § 5.1, we testing our `PBFGS` algorithm on different values of s. This was a controlled experiment, of course, as we kept the randomly generated parameter sets and x_0's constant with MATLAB's `rng` function. We achieved the following results and compared them with MATLAB's `fmincon-sqp` algorithm.

MATLAB Terminal Output

```
Testing values m = 2, k = 1 and T = 2 for 1 parameter sets with 50 random initial points each
PBGFS, s = 0.1
parameters: a = [2.0355      6.4816], b = [2.2005      3.1653], c = [0.32714      0.85914]
F max 5.0928 at [0.99141     0.85843] found for 22 times, avg iterations of 1.2727 and avg time of 0.90684
F max 3.3833 at [0.32738       1.389] found for 17 times, avg iterations of 1.6471 and avg time of 0.82686
F max 4.4615 at [0.83285      1.0131] found for 8 times, avg iterations of 1.875 and avg time of 0.82906
F max 5.3658 at [1.1409     0.85914] found for 3 times, avg iterations of 4.3333 and avg time of 1.0097
---------------------------------------------------------------------------------------------------
PBGFS, s = 0.01
parameters: a = [2.0355      6.4816], b = [2.2005      3.1653], c = [0.32714      0.85914]
F max 5.367 at [1.041     0.93316] found for 45 times, avg iterations of 1.2444 and avg time of 0.83702
F max 3.3832 at [0.32716      1.6728] found for 5 times, avg iterations of 2.2 and avg time of 0.73754
---------------------------------------------------------------------------------------------------
PBGFS, s = 0.001
parameters: a = [2.0355      6.4816], b = [2.2005      3.1653], c = [0.32714      0.85914]
F max 5.3659 at [1.0399     0.95785] found for 49 times, avg iterations of 1.1633 and avg time of 0.8725
F max 3.3831 at [0.32714       1.251] found for 1 times, avg iterations of 24 and avg time of 2.7296
---------------------------------------------------------------------------------------------------
PBGFS, s = 0.0001
parameters: a = [2.0355      6.4816], b = [2.2005      3.1653], c = [0.32714      0.85914]
F max 5.3658 at [1.0398     0.96003] found for 49 times, avg iterations of 1.1633 and avg time of 0.80719
F max 3.3831 at [0.32714       1.263] found for 1 times, avg iterations of 24 and avg time of 2.8242
```

We see that `PBFGS` finds several solutions for large s. As the approximated penalty function approaches the true penalty, fewer unique solutions are found, and perhaps more importantly, the optimal solution does indeed converge to the optimal solution (confirmed theoretically by Theorem 2) found by MATLAB's `fmincon-sqp` algorithm as seen in the below terminal output.

MATLAB Terminal Output

```
fmincon-sqp
parameters: a = [2.0355     6.4816], b = [2.2005      3.1653], c = [0.32714     0.85914]
F max 5.3658 at [1.1409     0.85914] found for 50 times, avg iterations of 1.02 and avg time of 0.003842
```

6 Discussion

Firstly, as the equation given for R_i incorporates the minimiser function, any non-linear program using exactly what was given in § 1.2 would be nonconvex and nondifferentiable as outlined in § 2.2.3. Hence writing an algorithm to solve this NLP would be difficult. Instead of using the nondifferentiable min function in the NLP, a series of constraints were created, in particular, constraints g_{m+1} to g_{2m}, to satisfy the conditions that involved the minimiser function. This gave the NLP shown in § 2.2.2.

Using those KKT conditions outlined in § 2.2.4, we hypothesised an algorithm to simultaneously solve all KKT conditions. To test feasibility of such an algorithm, we initially implemented MATLAB's `solve` function to simultaneously solve the KKT conditions. However, as such a program involves 2^{2m+1} inequality constraints, the algorithm would have to consider 2^{2m+1} possible combinations of active and inactive constraints, an inefficiency that would result in poor performance. Hence we considered an approximated NLP.

The NLP was then converted into a form that was continuous, differentiable and unconstrained. This was done by converting the min function used in R_i to a max function, then approximating the max function through use of the error function, as described in § 3.1.2. This resulted in a new NLP shown in § 3.1.3. The penalty function was then used to convert the NLP into an unconstrained function, shown in § 3.1.3. Lastly, an algorithm was designed that implemented the BFGS method, using MATLAB's Symbolic Toolbox to analytically solve the gradient of the proposed penalty function (Equation 4) at each step. This algorithm was used to test the NLP given in the form of the current approximated penalty function. This algorithm proved to be successful and faster than the previous algorithms which attempted to simultaneously solve the KKT conditions outlined in § 2.2.4. The results are shown above in § 5.

The final `PBFGS` algorithm was compared to several pre-existing optimisation algorithms in MATLAB by running each algorithm for 100 different starting points with $m = 6$, $k = 1$ and $T = 10$ and a constant set of randomised initialised parameters. The algorithms that were tested against ours include the `fminunc` algorithm and the `fmincon` algorithms mentioned in § 3.2. As shown in § 5.1, each algorithm found the same largest optimal value, specifically $-f(\mathbf{x}^*) = 34.503$. The SQP and interior-point algorithms found this solution on every one of the 100 runs, hence SQP and the interior-point algorithms were the most most effective and precise algorithms. Our BFGS algorithm found this solution 99 times out of the 100 runs, and one other optimal solution for one of the other runs, that was less than $-f(\mathbf{x}^*) = 34.503$.

However, as the original problem is revenue maximisation, we would obviously take the solution that gives the largest value of $-f(\mathbf{x})$.

The `fminunc` function found 10 different optimal solutions. As briefly mentioned in § 5.1, this is perhaps because the algorithm minimises an approximated penalty function through further approximations outlined in § 3.2, and therefore this approximating effect is "compounded", misleading the algorithm into several alternative minimisers. The active-set algorithm found 5 different optimal solutions. These other optimal solutions for each of these algorithms were also less than the most occurring optimal solution, $-f(\mathbf{x}^*) = 34.503$. The `fmincon-activeset` algorithm may have found multiple optimal solutions due to the nature of the algorithm. `fmincon-sqp` and `fmincon-interiorpoint` can indeed take steps that fail to descend, then retrace and take smaller steps that lead to a smaller objective function value. Conversely, `fmincon-activeset` cannot do this, and so this is a potential reason for the multiple solutions achieved for different starting points $\mathbf{x}_0$.

The MATLAB algorithms were shown to be significantly faster than our algorithm. This is due to how the `fminunc` and `fmincon` algorithm use more approximations in their steps, hence reaching an optimal solution faster than our BFGS algorithm, which analytically solves the derivative of the penalty function instead of approximating. The fastest pre-existing algorithm used was the SQP algorithm, which found the solution $-f(\mathbf{x}^*) = 34.503$ an average 585 times faster than the algorithm we implemented.

7 Conclusion

It was found that the BFGS algorithm was the most successful out of the algorithms that we tried to implemented in terms of speed and effectiveness. However when compared to other pre-existing MATLAB optimisation algorithms, the `fmincon` SQP algorithm was the most successful and efficient in determining the best allocation of ad times that maximised total revenue.

In future, when implementing an algorithm from scratch, the use of more MATLAB toolboxes should be explored, as well as the potential use of other software. When solving future optimisation problems, the pre-existing MATLAB functions are also sufficient to use.

8 Appendix

8.1 Example Computational Studies

8.1.1 Study 1

MATLAB Terminal Output

```
Testing values m = 4, k = 1 and T = 10 for 2 parameter sets with 100 random initial points each set.
PBGFS
parameters:
a = [6.795164, 7.444393, 9.036574, 3.877268, ]
b = [9.386928, 9.872439, 6.113394, 4.324713, ]
c = [0.959864, 0.806119, 0.564277, 0.679158, ]
F max 29.697475 at x = [1.609762, 4.777100, 1.608633, 1.303968, ];
found for 100 times, avg iterations of 1.010000 and avg time of 0.426537
----------------------------------------------------------------------------------------------------
PBGFS
parameters:
a = [8.521266, 3.645745, 2.079376, 5.678643, ]
b = [5.027213, 2.746220, 4.649021, 5.506953, ]
c = [0.491326, 0.556660, 0.506226, 0.828511, ]
F max 17.929407 at x = [2.376730, 4.284376, 1.495337, 1.843557, ];
found for 100 times, avg iterations of 1.050000 and avg time of 0.446249
----------------------------------------------------------------------------------------------------
fminunc
parameters:
a = [6.795164, 7.444393, 9.036574, 3.877268, ]
b = [9.386928, 9.872439, 6.113394, 4.324713, ]
c = [0.959864, 0.806119, 0.564277, 0.679158, ]
F max 29.697475 at x = [1.421261, 4.777100, 1.464017, 1.946546, ];
found for 100 times, avg iterations of 1.010000 and avg time of 0.001578
----------------------------------------------------------------------------------------------------
fminunc
parameters:
a = [8.521266, 3.645745, 2.079376, 5.678643, ]
b = [5.027213, 2.746220, 4.649021, 5.506953, ]
c = [0.491326, 0.556660, 0.506226, 0.828511, ]
F max 17.929407 at x = [1.415834, 4.777100, 1.955216, 1.132288, ];
found for 100 times, avg iterations of 1.010000 and avg time of 0.001625
----------------------------------------------------------------------------------------------------
fmincon-activeset
parameters:
a = [6.795164, 7.444393, 9.036574, 3.877268, ]
b = [9.386928, 9.872439, 6.113394, 4.324713, ]
c = [0.959864, 0.806119, 0.564277, 0.679158, ]
F max 29.697475 at x = [1.728051, 3.183676, 2.456915, 2.631358, ];
found for 95 times, avg iterations of 1.031579 and avg time of 0.003708
F max 26.461406 at x = [3.211007, 2.604929, 0.564277, 3.619787, ];
found for 1 times, avg iterations of 1.000000 and avg time of 0.003014
F max 26.571199 at x = [0.959864, 2.880517, 3.180420, 2.979198, ];
found for 1 times, avg iterations of 1.000000 and avg time of 0.003434
F max 27.004380 at x = [0.992516, 1.970299, 4.017083, 3.020103, ];
found for 1 times, avg iterations of 1.000000 and avg time of 0.002795
F max 25.939871 at x = [3.716801, 0.906312, 2.485755, 2.891132, ];
found for 1 times, avg iterations of 1.000000 and avg time of 0.002940
F max 22.867157 at x = [4.091379, 0.884900, 0.606697, 4.417024, ];
found for 1 times, avg iterations of 1.000000 and avg time of 0.002543
----------------------------------------------------------------------------------------------------
fmincon-activeset
```

```
parameters:
a = [8.521266, 3.645745, 2.079376, 5.678643, ]
b = [5.027213, 2.746220, 4.649021, 5.506953, ]
c = [0.491326, 0.556660, 0.506226, 0.828511, ]
F max 17.929407 at x = [3.616588, 3.572229, 1.780264, 1.030920, ];
found for 98 times, avg iterations of 1.010204 and avg time of 0.003509
F max 15.663440 at x = [3.575350, 0.638607, 1.323523, 4.462520, ];
found for 1 times, avg iterations of 1.000000 and avg time of 0.002609
F max 16.781919 at x = [1.869161, 3.615581, 1.297665, 3.217592, ];
found for 1 times, avg iterations of 1.000000 and avg time of 0.002613
---------------------------------------------------------------------------------------------------
fmincon-sqp
parameters:
a = [6.795164, 7.444393, 9.036574, 3.877268, ]
b = [9.386928, 9.872439, 6.113394, 4.324713, ]
c = [0.959864, 0.806119, 0.564277, 0.679158, ]
F max 29.697475 at x = [1.618189, 3.269712, 1.699861, 3.412238, ];
found for 100 times, avg iterations of 1.030000 and avg time of 0.002787
---------------------------------------------------------------------------------------------------
fmincon-sqp
parameters:
a = [8.521266, 3.645745, 2.079376, 5.678643, ]
b = [5.027213, 2.746220, 4.649021, 5.506953, ]
c = [0.491326, 0.556660, 0.506226, 0.828511, ]
F max 17.929407 at x = [3.616585, 3.572227, 1.780261, 1.030917, ];
found for 100 times, avg iterations of 1.010000 and avg time of 0.002373
---------------------------------------------------------------------------------------------------
fmincon-interiorpoint
parameters:
a = [6.795164, 7.444393, 9.036574, 3.877268, ]
b = [9.386928, 9.872439, 6.113394, 4.324713, ]
c = [0.959864, 0.806119, 0.564277, 0.679158, ]
F max 29.697475 at x = [1.955793, 3.526829, 1.816401, 1.622878, ];
found for 100 times, avg iterations of 1.080000 and avg time of 0.005362
---------------------------------------------------------------------------------------------------
fmincon-interiorpoint
parameters:
a = [8.521266, 3.645745, 2.079376, 5.678643, ]
b = [5.027213, 2.746220, 4.649021, 5.506953, ]
c = [0.491326, 0.556660, 0.506226, 0.828511, ]
F max 17.929407 at x = [1.579892, 4.205799, 1.719434, 1.804636, ];
found for 100 times, avg iterations of 1.070000 and avg time of 0.005004
```

8.1.2 Study 2

MATLAB Terminal Output

```
Testing values m = 2, k = 2.2 and T = 5.6 for 2 parameter sets with 100 random initial points each set.
PBGFS
parameters:
a = [5.536181, 7.733379, ]
b = [2.715669, 1.300311, ]
c = [0.242389, 0.859308, ]
F max 4.015980 at x = [2.413944, 2.151683, ];
found for 100 times, avg iterations of 2.198000 and avg time of 0.197101
---------------------------------------------------------------------------------------------------
PBGFS
parameters:
a = [2.189763, 3.437992, ]
```

```
b = [1.604583, 5.774306, ]
c = [0.447244, 0.780757, ]
F max 7.378888 at x = [1.326775, 3.474697, ];
found for 100 times, avg iterations of 2.198000 and avg time of 0.186150
-----------------------------------------------------------------------------------------
fminunc
parameters:
a = [5.536181, 7.733379, ]
b = [2.715669, 1.300311, ]
c = [0.242389, 0.859308, ]
F max 4.015980 at x = [1.449057, 1.269595, ];
found for 100 times, avg iterations of 2.188000 and avg time of 0.002091
-----------------------------------------------------------------------------------------
fminunc
parameters:
a = [2.189763, 3.437992, ]
b = [1.604583, 5.774306, ]
c = [0.447244, 0.780757, ]
F max 7.378888 at x = [0.730441, 1.355408, ];
found for 100 times, avg iterations of 2.188000 and avg time of 0.001467
-----------------------------------------------------------------------------------------
fmincon-activeset
parameters:
a = [5.536181, 7.733379, ]
b = [2.715669, 1.300311, ]
c = [0.242389, 0.859308, ]
F max 4.015980 at x = [4.740692, 0.859308, ];
found for 100 times, avg iterations of 2.198000 and avg time of 0.006315
-----------------------------------------------------------------------------------------
fmincon-activeset
parameters:
a = [2.189763, 3.437992, ]
b = [1.604583, 5.774306, ]
c = [0.447244, 0.780757, ]
F max 7.378888 at x = [2.321993, 3.278007, ];
found for 100 times, avg iterations of 2.198000 and avg time of 0.003623
-----------------------------------------------------------------------------------------
fmincon-sqp
parameters:
a = [5.536181, 7.733379, ]
b = [2.715669, 1.300311, ]
c = [0.242389, 0.859308, ]
F max 4.015980 at x = [4.740682, 0.859318, ];
found for 100 times, avg iterations of 2.198000 and avg time of 0.002507
-----------------------------------------------------------------------------------------
fmincon-sqp
parameters:
a = [2.189763, 3.437992, ]
b = [1.604583, 5.774306, ]
c = [0.447244, 0.780757, ]
F max 7.378888 at x = [2.321993, 3.278007, ];
found for 100 times, avg iterations of 2.198000 and avg time of 0.002252
-----------------------------------------------------------------------------------------
fmincon-interiorpoint
parameters:
a = [5.536181, 7.733379, ]
b = [2.715669, 1.300311, ]
c = [0.242389, 0.859308, ]
F max 4.015980 at x = [3.041284, 1.906786, ];
found for 100 times, avg iterations of 2.338000 and avg time of 0.006327
-----------------------------------------------------------------------------------------
```

```
fmincon-interiorpoint
parameters:
a = [2.189763, 3.437992, ]
b = [1.604583, 5.774306, ]
c = [0.447244, 0.780757, ]
F max 7.378888 at x = [2.491218, 2.005320, ];
found for 100 times, avg iterations of 2.268000 and avg time of 0.005370
```

8.2 MATLAB Code

8.2.1 Symbolic Toolbox BFGS Implementation

```matlab
function [xstar,fstar,iter] = BFGS(f,n,x0)
%BFGS uses the  B r o y d e n FletcherGoldfarbShanno  unconstrained optimisation
%   method to find the local minimum of a function.
%
%   Parameters
%   ----------
%   f : symbolic function
%       (for example, syms x; f = x^2)
%   n : int
%       size of symbolic x (or in the context of our project,
%                           number of ads)
%   x0 : vector of size 1xn
%       initial point for descent search
%
%   Returns
%   -------
%   xstar : vector of size 1xn
%       this is the minimiser
%   fstar : double
%       this is the value of f at the minimiser
%   iter : int
%       this is the number of iterations that the search algorithm took to
%       find the minimiser

    x = sym('x', [1, n]);
    syms t

    % BFGS Step 1
    H = eye(n); % initial BFGS H matrix
    tol = 0.01; % stopping tolerance
    gradf = gradient(f); % exact gradient of f using symbolic toolbox
    solfound = false;
    iter = 1; % starting iteration number

    xs = zeros(n,10); % keep track of x at each step of descent
    xs(:,iter) = x0; % set initial x value

    while true
        % BFGS Step 2
        initial_f_grad_value = double(subs(gradf,x,xs(:,iter).'));
        initial_abs_f_grad_value = norm(initial_f_grad_value);

        if initial_abs_f_grad_value < tol
            solfound = true;
            break;
        end
```

```
        d = -H*initial_f_grad_value;

        % if d is NaN then something went wrong; return last x
        if isnan(d)
           solfound = false;
           break
        end

        % BFGS Step 3
        q = subs(f, x, (xs(:,iter) + t*d).');
        qfun = matlabFunction(q);
        [tval,fval,exitflag,output] = fminunc(qfun,0,optimoptions('fminunc','Display','none'));

        % BFGS Step 3
        xs(:,iter+1) = xs(:,iter) + tval*d;

        %%% BFGS Update
        s = xs(:,iter+1) - xs(:,iter);
        g = double(subs(gradf, x, xs(:,iter+1).')) - double(subs(gradf, x, xs(:,iter).'));
        r = (H*g) / (dot(s,g));
        H = H + ( (1 + dot(r,g))/(dot(s,g)) ) * (s*s.') - (s*r.' + r*s.');
        iter = iter+1;
    end

    % returns
    if solfound == true
        xstar = xs(:,iter);
        fstar = 1*double(subs(f,x,xs(:,iter).'));
    else
       xstar = xs(:,iter-1);
       fstar = 1*double(subs(f,x,xs(:,iter-1).'));
    end
end
```

8.2.2 l1 norm approximation

```
function h = modap(x)
%modap approximates the absolute value of x, |x|.
%
%   Parameters
%   ----------
%   x : int/double or symbol
%
%   Returns
%   -------
%   h : double (if x is double); symbolic function (if x is symbolic)

    s = 0.0001; % set approximation parameter s (proposition 5)
    h = x.*erf(x./s); % use error function to approximate |x|
end
```

8.2.3 Max function approximation

```
function m = maxap(x,a)
%maxap uses approximation of the l1 norm in modap to approximate the
%   max(x,b) function
%
```

```
%   Parameters
%   ----------
%   x : int/double or symbol
%   a : int/double
%
%   Returns
%   -------
%   m : double (if x is double); symbolic function (if x is symbolic)
    m = (x + a + modap(x - a))./2;
end
```

8.2.4 Symbolic penalty function

```
function f = objfun(x,n,a,b,c,k,T)
%objfun is the approximated penalty function described in the report
%   (section 3.1.3). This MATLAB function specifically returns a symbolic
%   function for use with MATLAB's Symbolic Toolbox.
%
%   Parameters
%   ----------
%   x : 1xn symbol vector
%   n : int
%       size of symbolic x (or in the context of our project,
%                               number of ads)
%   a : 1xn vector
%       parameter set for a values, a > 0
%   b : 1xn vector
%       parameter set for b values
%   c : 1xn vector
%       parameter set for c values
%   k : double
%       must have k > 0
%   T : double
%       total ad time allocation constraint
%
%   Returns
%   -------
%   f : symbolic function
%       this is the penalty function from 3.1.3

    m = n;
    of = sym('of', [1, m]);
    cons = sym('cons', [1, m+1]);
    f = sym('f', [1, m]);

    % Penalty Function
    alpha = 10;
    for i = 1:m
        of(i) = maxap(-a(i)*k*x(i).^2,-b(i)); % objective function
        cons(i) = modap(maxap(c(i)-x(i),0)); % constraints
    end
    cons(end) = modap(maxap(sum(x) - T,0)); % final constraint

    f = sum(of) + alpha*sum(cons); % penalty function
end
```

8.2.5 Scalar objective function(s)

<u>Scalar Objective Function for `fminunc`</u>

```
function f = sobjfun(x,m,a,b,c,k,T)
```

```
%objfun is the approximated penalty function described in the report
%   (section 3.1.3). This MATLAB function specifically returns a scalar
%   value of f. Therefore all the inputs are the same as objfun (note m =
%   n) except that x is a scalar 1xn vector.

    n_cons = m+1;
    alpha = 10; % penalty parameter
    of = zeros(m,1);
    cons = zeros(n_cons,1);
    for i = 1:m
        of(i) = maxap(-a(i)*k*x(i).^2,-b(i));
        cons(i) = modap(maxap(c(i)-x(i),0));
    end
    cons(end) = modap(maxap(sum(x) - T,0));

    f = sum(of) + alpha*sum(cons);
end
```

Scalar Objective Function for fmincon

```
function f = conobjfun(x,a,b,k)
%objfun is the approximated objective function for the initial NLP
%   described in section 2.2. Inputs are the same as sobjfun.

    f = sum(max(-a.*k.*x.^2, -b));
end
```

8.2.6 Computational Study Algorithm

```
n_param_sets = 5; % number of randomly generated parameter sets
n_runs = 10; % number of randomly generated initial points for each parameter
    set
n = 5; % same as m
m = n; % number of ads to be displayed over time T
T = 10; % time T
k = 1; % constant k > 0
x = sym('x', [1, m]); % symbolic x for PBFGS and MATLAB fmin
tol = 0.1; % tolerance for classifying a found minimiser as existing vs new
disp_comparison = true; % display setting:
% if true then will display the algorithm comparison (section 5.1)
% else, display scaling speed test results (section 5.2)

%rng(43); % seed for controlled experimentation (comment out)

% initialise random point matrices
as = zeros(n_param_sets,n);
bs = zeros(n_param_sets,n);
cs = zeros(n_param_sets,n);
x0s = zeros(n_runs,n);

% populate random point matrices
for i = 1:n_param_sets
    as(i,:) = unifrnd(1,10,1,m);
    bs(i,:) = unifrnd(1,10,1,m);
    cs(i,:) = unifrnd(0,1,1,m);
end
for i = 1:n_runs
    x0s(i,:) = unifrnd(0,5,1,m);
end

```

```
disp("Testing values m = " + m + ", k = " + k + " and T = " + T + " for " + n_param_sets + ...
     " parameter sets with " + n_runs + " random initial points each set.");
% array for recording times when disp_comparison = false
timer_array = zeros(n_param_sets*n_runs, 5);
for z = 1:5
    current_iteration = 1; % record total iterations
    for i = 1:n_param_sets % run over each parameter set
        a = as(i,:);
        b = bs(i,:);
        c = cs(i,:);
        n_min = 0; % number of minimums found for particular parameter set and x0
        solutions = zeros(10, m+4); % xstar (takes m values), fmin, n_min, n_iter, tElapsed
        for j = 1:n_runs % test several x0 values for given set of parameters
            tStart = tic; % start recording time
            x0 = x0s(j,:);
            switch z
                case 1
                    % penalty function custom BFGS
                    [xstar, fstar, n_iter] = BFGS(objfun(x,n,a,b,c,k,T),n,x0);
                    xstar = xstar.';
                    fstar = conobjfun(xstar,a,b,k);
                case 2
                    % penalty function MATLAB fminunc
                    [xstar,fstar,exitflag,output] = fminunc(@(x)sobjfun(x,m,a,b,c,k,T),x0,optimoptions('fminunc','Display','none'));
                    n_iter = output.iterations;
                    fstar = conobjfun(xstar,a,b,k);
                case 3
                    % fmincon-activeset with ineq constraints in 2.2.1
                    A = [-1*eye(n);ones(1,n);-1*eye(n)];
                    B = [-1.*c, T, zeros(1,n)];
                    options = optimoptions(@fmincon,'Algorithm','active-set','Display','none');
                    [xstar,fstar,exitflag,output] = fmincon(@(x)conobjfun(x,a,b,k),x0,A,B,[],[],[],[],[],options);
                    n_iter = output.iterations;
                case 4
                    % fmincon-sqp with ineq constraints in 2.2.1
                    A = [-1*eye(n);ones(1,n);-1*eye(n)];
                    B = [-1.*c, T, zeros(1,n)];
                    options = optimoptions(@fmincon,'Algorithm','sqp','Display','none');
                    [xstar,fstar,exitflag,output] = fmincon(@(x)conobjfun(x,a,b,k),x0,A,B,[],[],[],[],[],options);
                    n_iter = output.iterations;
                case 5
                    % fmincon-interiorpoint with ineq constraints in 2.2.1
                    A = [-1*eye(n);ones(1,n);-1*eye(n)];
                    B = [-1.*c, T, zeros(1,n)];
                    options = optimoptions(@fmincon,'Algorithm','interior-point','Display','none');
                    [xstar,fstar,exitflag,output] = fmincon(@(x)conobjfun(x,a,b,k),x0,A,B,[],[],[],[],[],options);
                    n_iter = output.iterations;
            end
            tElapsed = toc(tStart); % finish recording time
            timer_array(current_iteration, z) = tElapsed; % append time to timer_array
```

```
            fstar = -fstar; % take negative to get maximum

            % handle for new/old min finding
            oldmin = false;
            for p = 1:n_min % iterate over each min already found
                if abs(fstar - solutions(p,m+1)) < tol % if new min is within tolerance of found min
                    % then append to solutions found for that min
                    solutions(p,m+2) = solutions(p,m+2) + 1;
                    solutions(p,m+3) = solutions(p,m+3) + k;
                    solutions(p,m+4) = solutions(p,m+4) + tElapsed;
                    oldmin = true; % found min already exists
                    break;
                end
            end
            if oldmin == false % if a new min is found
                n_min = n_min + 1; % increase number of minds
                solutions(n_min,:) = [xstar, [fstar, 1, n_iter, tElapsed]]; % record new min
            end
            current_iteration = current_iteration + 1; % increase iteration
        end
        if disp_comparison
            switch z
                case 1
                    % penalty function BFGS
                    fprintf("PBGFS \n")
                    fprintf("parameters: \na = [%s] \nb = [%s] \nc = [%s]\n", ...
                        sprintf('%f, ', a), sprintf('%f, ', b), sprintf('%f, ', c));
                case 2
                    % penalty function MATLAB fminunc
                    fprintf("fminunc \n")
                    fprintf("parameters: \na = [%s] \nb = [%s] \nc = [%s]\n", ...
                        sprintf('%f, ', a), sprintf('%f, ', b), sprintf('%f, ', c));
                case 3
                    % penalty function MATLAB fminunc
                    fprintf("fmincon-activeset \n")
                    fprintf("parameters: \na = [%s] \nb = [%s] \nc = [%s]\n", ...
                        sprintf('%f, ', a), sprintf('%f, ', b), sprintf('%f, ', c));
                case 4
                    % penalty function MATLAB fminunc
                    fprintf("fmincon-sqp \n")
                    fprintf("parameters: \na = [%s] \nb = [%s] \nc = [%s]\n", ...
                        sprintf('%f, ', a), sprintf('%f, ', b), sprintf('%f, ', c));
                case 5
                    % penalty function MATLAB fminunc
                    fprintf("fmincon-interiorpoint\n")
                    fprintf("parameters: \na = [%s] \nb = [%s] \nc = [%s]\n", ...
                        sprintf('%f, ', a), sprintf('%f, ', b), sprintf('%f, ', c));

            end
```

```
            for u = 1:n_min
                avg_iter = solutions(u, m + 3)/solutions(u,m+2);
                avg_time = solutions(u, m + 4)/solutions(u,m+2);
                to_display = sprintf('%f, ', solutions(u,1:m));
                fprintf("F max %f at x = [%s]; \nfound for %i times, avg iterations of %f and avg time of %f \n", ...
                    solutions(u,m+1), to_display, solutions(u, m + 2), avg_iter, avg_time)
            end
            disp("-------------------------------------------------------------------------------------
        end
    end
end

algorithm_times = mean(timer_array,1); % average tElapsed for each algorithm over all parameter sets AND x0

if ~disp_comparison % display for speed test
    for i = 1:5
        switch i
            case 1
                disp("PBFGS took an average of " + algorithm_times(i) + " seconds.")
            case 2
                disp("fminunc took an average of " + algorithm_times(i) + " seconds.")
            case 3
                disp("fmincon-activeset took an average of " + algorithm_times(i) + " seconds.")
            case 4
                disp("fmincon-sqp took an average of " + algorithm_times(i) + " seconds.")
            case 5
                disp("fmincon-interiorpoint took an average of " + algorithm_times(i) + " seconds.")
        end
    end
end
```

References

[1] Zhang, J. (2022). *Techniques in Operations Research MAST30013* [Lecture notes]. https://canvas.lms.unimelb.edu.au/courses/128801/files/10384742?wrap=1

[2] The MathWorks, Inc. (2022). *fminunc*. https://au.mathworks.com/help/optim/ug/fminunc.html

[3] The MathWorks, Inc. (2022). *Choosing the Algorithm*. https://au.mathworks.com/help/optim/ug/choosing-the-algorithm.html

[4] The MathWorks, Inc. (2022). *Constrained Nonlinear Optimization Algorithms*. https://au.mathworks.com/help/optim/ug/constrained-nonlinear-optimization-algorithms.html#bsgppl4

[5] Northwestern University. (2022). *fminunc (Optimization Toolbox)*. http://www.ece.northwestern.edu/local-apps/matlabhelp/toolbox/optim/fminunc.html

[6] Wikipedia (2022). *Definite Matrix*. https://en.wikipedia.org/wiki/Definite_matrix

[7] Science Direct (2022). *Method of Steepest Descent*.
https://www.sciencedirect.com/topics/mathematics/steepest-descent-method