Non-smooth optimisation applied to improving steam turbine designs

Jonathan Grant-Peters

Wolfson College

University of Oxford

DPhil

Michaelmas 2020

Acknowledgements

I would like to thank my supervisor Raphael Hauser, without whose guidance and encouragement I would have been second guessing and abandoning my direction of research every second month.

I also want to acknowledge may industrial supervisor Tobias Ruehle for his tireless support in answering all of my Steam Turbine related question, not to mention his hospitality during my visits to Siemens.

I'd like to thank my wife Melissa for her support and encouragement not to mention her tolerating 8 months of "only a few more months to go".

Thanks to my parents whose teaching, encouragement, and prayer support have been instrumental in me getting to this point.

Thanks to all of Cohort 3, whose community has without a doubt been the highlight of my time in Oxford. I also want to thank Chris Breward and Colin Please for running the amazing CDT that InFoMM is.

Finally I want to thank Anca Mustata for expanding my horizons and waking up my slumbering mind.

This project is motivated by Siemens' desire to improve the efficiency of their steam turbines. Siemens has developed a software called 3dv1d which tests a theoretical steam turbine design both for efficiency and practicality. By using 3dv1d to define black box objective and constraint functions, we can find optimal steam turbine designs by applying mathematical optimisation software.

We begin by investigating the optimisation algorithm currently used by Siemens, which is known to yield good steam turbine designs for Siemens even though it does not converge to a point which satisfies optimality conditions. This investigation reveals that this algorithm fails because the efficiency function is not continuously differentiable. Since the algorithm currently used is a smooth optimisation algorithm, this means its failure follows from the fact that it is not suitable to the problem.

Next we explore the literature for existing non-smooth optimisation (NSO) algorithms which may be suitable for solving the problem. We identify one of these as being most applicable, and test its effectiveness only to find that it runs too slowly to be useful in practice.

Our continued exploration of the topic of non-smooth optimisation led us to the particular topic of exact line search for non-smooth optimisation, and more specifically non-smooth 1D optimisation. Here we have noted that many of the objections to using ELS do not apply in the context of NSO. Moreover, while the topic of 1D optimisation has been well developed, there is a surprising absence of any algorithm capable of robust super-linear convergence when applied to 1D NS functions. Our main contribution to mathematics through this project has been to construct a new algorithm to fill this void.

In addition to this, we have also addressed the larger problem motivated by Siemens: to robustly locate local optima for functions like that implied by 3dv1d. While not being as rigorous as in 1D optimisation, we have constructed a heuristic for the purpose of solving problems like Siemens' which we demonstrate both to perform robustly and often find the best known solution.

Contents

List of Figures

Acronyms

BASM blind active set method.

BFGS Broyden–Fletcher–Goldfarb–Shanno.

BLS backtracking line search.

DGM discrete gradient method.

DUPM dynamic underestimating polynomial method.

ELS exact line search.

EUPM extremal underestimating polynomial method.

GS gradient sampling.

HLS hybrid line search.

ILS inexact line search.

KKT Karush-Kuhn-Tucker.

LICQ linear independence constraint qualification.

LS line search.

LSM line search method.

MS Mifflin-Strodiot.

MSA Mifflin-Strodiot algorithm.

NAG Numerical Algorithms Group.

NC non-convex.

NP nonlinear programming.

NS non-smooth.

NSO Non-smooth optimisation.

NSO non-smooth optimisation.

PS piecewise smooth.

QN Quasi-Newton.

RHS right hand side.

SO smooth optimisation.

SQP sequential quadratic programming.

SQP-GS sequential quadratic programming with gradient sampling.

SUPM static underestimating polynomial method.

UPM underestimating polynomial method.

Glossary

$\omega(\boldsymbol{v}, \boldsymbol{w})$ The angle between the vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ (see **??**).

c_1 The parameter from the Wolfe Conditions (see Definition 2.8) which ensures a sufficient reduction of the merit function, also known as the Armijo constant.

$\mathcal{X}$ Both brackets (see Definition 5.1) and extended brackets (see Definition 5.6).

x^L The point on the left of a bracket (see Definition 5.1), or the set of points on the left of an extended bracket (see Definition 5.6).

x^M The central point of a bracket or extended bracket (see Definitions 5.1 and 5.6).

x^R The point on the right of a bracket (see Definition 5.1), or the set of points on the right of an extended bracket (see Definition 5.6).

σ The function which determines the next point to evaluate within a bracketing method (see Algorithm 12).

U The function which updates the bracket within a bracketing method (see Algorithm 12).

$\boldsymbol{c}$ The constraint function (see Problem 2.1).

$\mathcal{B}_k$ The bundle of subgradients, used at the k-th iteration of a bundle method to approximate the Goldstein ϵ-subdifferential.

$\boldsymbol{H}_k$ The $n \times n$ matrix, used at the k-th iteration of a QN Line Search Method to approximate the hessian.

$\boldsymbol{x}_k$ The k-th iterate of a Line Search Method (see Algorithm 1).

$\boldsymbol{p}_k$ The search direction at the k-th iterate of a Line Search Method (see Algorithm 1).

α_k The step length at the k-th iterate of a Line Search Method (see Algorithm 1).

$\mathcal{I}_n$ The set of all length n sequences of EUPM iterations (see Definitions 5.10 and 5.11).

$\mathcal{S}(I)$ The set of sub-strings of the sequence I (see Definition 5.14).

Ω The feasible set (see Definition 2.1).

f The objective function (see Problem 2.1).

$Q_k(f, \mathcal{X}_0)$ The length k sequence of EUPM iterations generated by running the EUPM to the function f with the starting bracket $\mathcal{X}_0$ (see Definition 5.12).

$\mathcal{A}_f(\boldsymbol{x})$ The set of indices $\{i|\ f_i(\boldsymbol{x}) = f(\boldsymbol{x})\}$ (see Definition 4.3).

$\tilde{\mathcal{B}}_I$ The set of brackets such that there exists a function f with $Q_n(f, \mathcal{X}) = I$. (see Definition 5.13).

$\boldsymbol{v}$ Usually denotes a subgradient.

$\boldsymbol{x}^*$ A true local or global optimiser (see Definitions 2.2 and 2.3).

c_2 The parameter from the Wolfe Conditions (see Definition 2.8) which ensures that the step length is suitably large.

3dv1d The software package developed by Siemens to numerically compute the efficiency of a steam turbine with a given design.

e04ucf The name of the sequential quadratic programming algorithm implemented by NAG and used by Siemens.

Line Search The second phase of each iteration of a LSM: where the step length is calculated, is called a Line Search algorithm.

Line Search Method The class of optimisation algorithms which involve two steps per iteration: first selecting a search direction, and second determining how far to go that that direction.

Chapter 1

Introduction

1.1 Introduction

As of the year 2015, more than 80% of the worlds electricity was generated using fossil fuels [3, p. 4]. In order to harvest this electricity, the fuel is burnt to create thermal energy, which is then converted into electricity by use of a steam turbine. While turbines are used mainly in the context of fossil fuels, they are also applicable to geothermal and nuclear power, both of which also convert thermal energy to electricity.

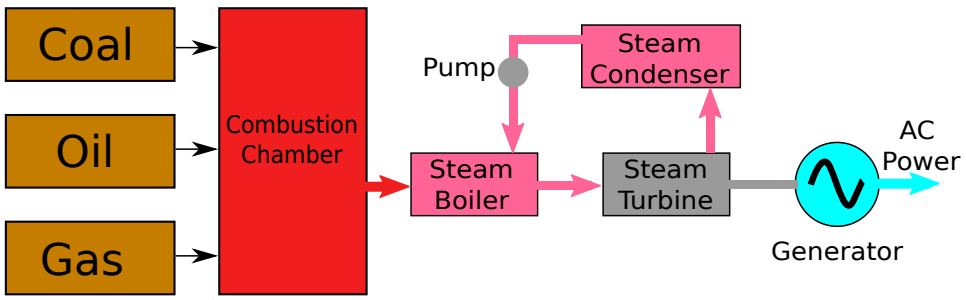

Figure 1.1.1 – Schematic showing the process by which chemical energy is converted first to thermal and then to electric energy.

The energy conversion process involves channelling the thermal energy, produced by burning the fuel, into water contained in a boiler, thus converting the water into steam which then is directed into the steam turbine. As the steam passes through the turbine, it expands and causes the turbine to rotate, thus driving the generator and inducing

electricity. Once the steam passes through the turbine, it is cooled and pumped back into the boiler.

As one of the worlds leading manufacturers of steam turbines, Siemens aim to make their designs as efficient as possible. For this purpose, they have constructed a software called 3dv1d which approximates the performance of a turbine given a particular design. Using this software, we can test the efficiency and practicality of a steam turbine without needing to build it. By writing the output of this software in the form of an objective function (the efficiency of the steam turbine) and a constraint function (which assesses how practical the design is), Siemens can utilise mathematical software in order to find optimal turbine designs.

The motivation of this project, which has been funded by Siemens, is to determine how best to apply mathematical optimisation to compute optimal steam turbine designs. In this chapter, we begin in Section 1.2 by looking at Siemens' practical problem and describe how Siemens' shape optimisation problem is converted to the problem of optimising efficiency with respect to a list of discrete and continuous parameters. We continue in Section 1.3 by discussing the software package which Siemens have built to model their steam turbines and explain how the software affects our problem formulation before setting the scene for later chapters in Section 1.4.

In Chapter 2, we provide a detailed description of this projects starting point. Specifically, we examine the optimisation algorithm which Siemens have been in the practice of using: a sequential quadratic programming (SQP) called e04ucf from the Numerical Algorithms Group (NAG) library [4], and identify the conditions which are required to guarantee convergence. We do this in order to understand why the algorithm terminates at a point which does not satisfy optimality conditions. Here we discover that the optimisation problem which we are attempting to solve is non-smooth (NS), meaning that it lies outside of the domain for which e04ucf was intended to be used. As a consequence of this, we redirected this project towards the topic of non-smooth optimisation (NSO) in order to address the motivating problem.

Therefore in Chapter 3 we explore the literature of optimisers suitable for non-

smooth functions and select the sequential quadratic programming with gradient sampling (SQP-GS) method of Curtis and Overton [5] from the list of algorithms suitable for this type of problem. However, when we trial SQP-GS on 3dv1d we find that SQP-GS actually yields worse results than the original smooth optimiser, while being far more expensive computationally. In the end, we find the best solutions to Siemens' problem by combining e04ucf (NAG's sequential quadratic programming algorithm) and SQP-GS to produce a heuristic which is practical for Siemens' purposes.

From this point on, we focus less on Siemens' specific problem and more on the general optimisation problem which shares the properties which we observe in Siemens' problem. We specifically interest ourselves in the line search subroutine which is found in a class of commonly used optimisation algorithms called line search methods (LSMs).

In Chapter 4 we argue that the subset of line search algorithms called exact line search may be effective for non-smooth optimisation problems despite not being competitive for smooth optimisation. In the process of doing so, we construct a new line search algorithm which makes use of ideas from exact line search while itself not being always exact. In Chapter 4, we demonstrate the effectiveness of this new line search algorithm by studying the performance of a greater line search method of which this line search is a subroutine, and not so much on the new line search itself. We display the full details of our new line search algorithm in Chapter 5 where we construct a new 1D optimiser which is designed to be used for this line search algorithm. It is within these two final chapters where what we consider to be our contributes to mathematics may be found, with Chapter 5 containing the more complete and thorough work of the two.

1.2 Designing Steam Turbines

The problem of designing an optimal steam turbine involves determining what is the ideal shape for each of its components, meaning that this is a shape optimisation problem. However, Siemens have simplified this problem through years of experience by

constructing a parametrisation of the different shapes involved. Therefore, instead of solving a shape optimisation problem, we have the much simpler option of optimising over a vector of parameters. In this section, we briefly describe some of these parameters which go into defining steam turbine design.

A steam turbine consists of a rotor shaft which passes through a number of compartments called drums as illustrated in Figure 1.2.1. This shaft is connected to a generator such that when it rotates, an electric current is induced. Fixed to the shaft are a number of blade rows, as illustrated in Figure 1.2.2. When steam is forced through these blades, it forces the shaft to rotate.

The shape of a turbine's blades depends on the drum in which they are found. Each turbine drum is designed with a different steam pressure in mind. A high pressure drum will contain short and thick blades while a low pressure drum has blades which are far longer and require strong roots to handle the stresses caused by the mass of the blades themselves.

There are two types of blade rows which are found within a drum. First of all, the rotor blade rows are fixed to the rotor shaft as previously described, and are what cause the shaft to rotate. Next, we have stator blade rows which are fixed to the casing which holds the rotor shaft in place. These blades redirect the steam as it passes through the drum, ensuring that it hits the rotor blades at an ideal angle.

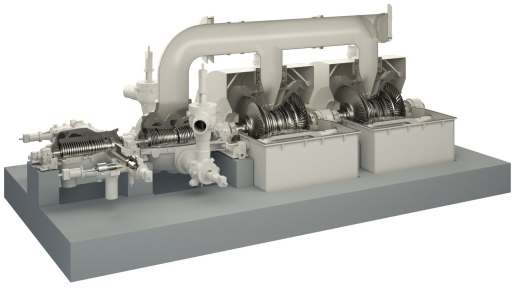

Figure 1.2.1 – Graphic of a steam turbine which has a high pressure drum, a medium pressure drum, and two low pressure drums. (Image taken from [1].)

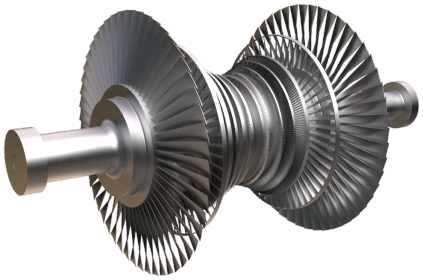

Figure 1.2.2 – We show above a rotor shaft with the rotor blade rows for a low pressure drum attached. This image has been provided by and is owned by Siemens.

Given a set of variable values, Siemens visualises the corresponding steam turbine with a schematic called a blade path (such as the one shown in Figure 1.2.3). We present one here in order to draw attention to the different types of variables which are visualised in the schematic. We now give a non-exhaustive list of the variables involved.

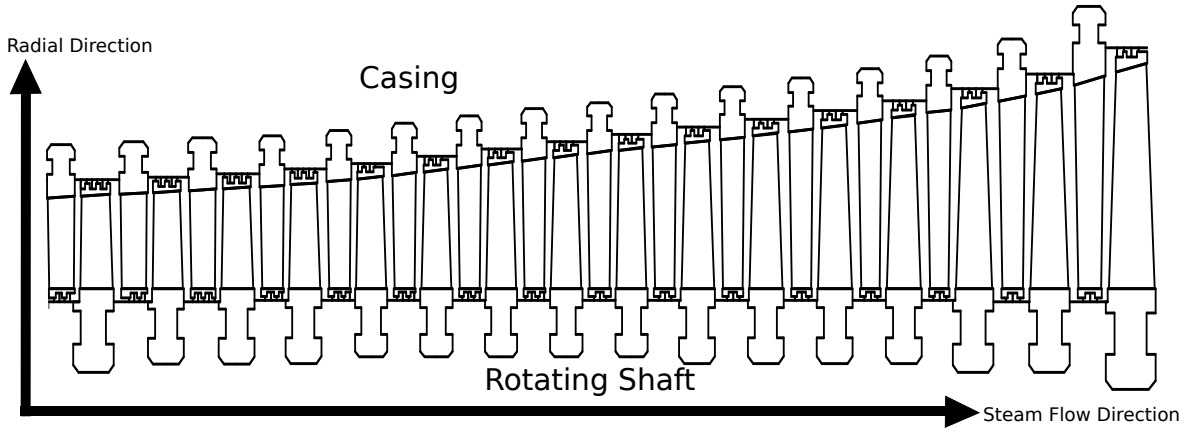

Figure 1.2.3 – Blade path of a high pressure turbine drum with 16 stages. This image has been provided by and is owned by Siemens.

Number of stages - Dimensional

Blade rows come in pairs. For each row of blades fastened to the rotor shaft, there is also a stator blade row fastened to the turbine casing which precedes it. We call this combination of a rotor and a stator blade row a stage. The value of this variable determines the number of each type of variable which follows.

Inner flow path diameter description - Continuous

Within each drum, the shape of the rotor shaft is not constant. In general, the rotor shaft will be thinner at the entrance to the drum and thicker at the exit. The inner flow path diameter description specifies the radial thickness of the rotor shaft at the position of every blade row within the drum (see Figure 1.2.4).

Blade heights - Continuous

Each row of blades within the turbine is rooted either to the rotor shaft (if it is a rotor blade row), or to the main casing (if it is a stator blade row). Regardless of which, the

length of each blade row is such that there does not exist any leftover space between the end of the blade row and the interior of the steam turbine. Therefore, the length of the blade row refers directly to the shape of the steam turbine (see Figure 1.2.5). By combining the blade height variables with the inner flow path diameter variables, we obtain the shape of the drum.

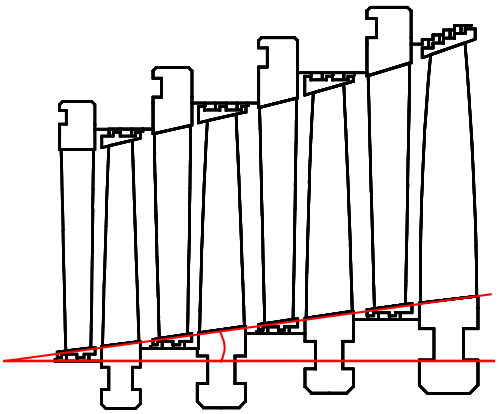

Figure 1.2.4 – We mark above the angle which can be used to uniquely specify the inner flow path diameter description. This image has been provided by and is owned by Siemens.

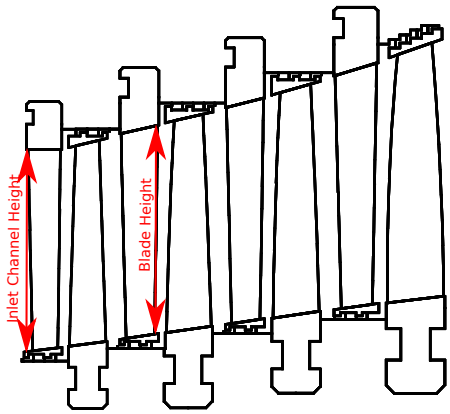

Figure 1.2.5 – We mark above the distance which is measured by the 'inlet channel height' and 'blade height' variables respectively. This image has been provided by and is owned by Siemens.

Blade root nominal size - Continuous

Every blade row is anchored into the rotating shaft (rotor blades) or casing (stator) with a root. The size of the root determines how thick the blade row can be, but is also related to the number of blade rows allowed, in that the thicker a blade root is the fewer of them there is space for. We illustrate the meaning of root size in Figure 1.2.6.

Stator blade stagger angle - Continuous

The stator blade stagger angle controls the angle between the direction of steam flow and the direction which the blade itself faces (see Figure 1.2.7). This angle will be constant for all blades within a single blade row, this implying that there will be one of these variable types for each blade row. Note that this particular variable is not

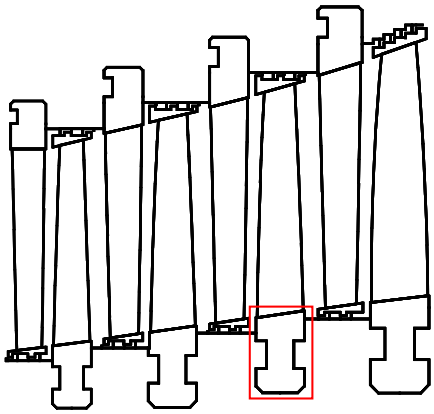

Figure 1.2.6 – We identify a blade root in this blade path with a red box. Note that the roots corresponding to the 1st and 3rd stage are of different sizes. This image has been provided by and is owned by Siemens.

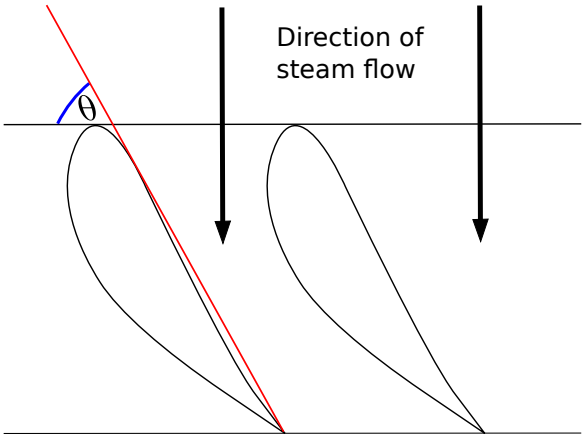

Figure 1.2.7 – We illustrate the stagger angle by the angle marked in blue between the direction of the steam flow, and the direction in which the blade is pointing. This image has been provided by and is owned by Siemens.

visualised within the blade path schematic.

Airfoil Scaling - Continuous

The airfoil scaling variable refers to the thickness of a blade row (not to be confused with the thickness of the root which anchors the blade row). While we might fix this value to equal the thickness of the corresponding root, allowing these values to be different offers additional flexibility when designing steam turbines. In particular, separating these variables is useful for cases where we might need a large root to handle stresses but would prefer a thinner blade row for other reasons.

Stage load distribution - Continuous

The stage load distribution variables differ from the other variables in that we have no direct control over these. While the physics of steam turbines lies outside the domain of this project, we understand these variables to be dependent variables which refer to the pressure within the steam turbine at each blade row. Siemens implement these variables as independent variables which interact with the others through specific constraints.

Blade materials - Discrete Ordinal

There are five types of metals which Siemens uses for its blades. We refer to them here by $\{m_1, m_2, \ldots, m_5\}$. The metals can be divided into two categories: those suitable for 'hot' temperatures ($> 520°$C) and those suitable for 'cold' temperatures ($< 510°$C). Aside from heat suitability, there are two factors which influence our choice of metal: strength and cost. These two are related to each other in that stronger metals cost more.

The order of metals suitable for 'hot' temperatures from weakest to strongest is $\{m_1, m_2, m_3\}$, while the order of metals suitable for 'cold' temperatures ordered from weakest to strongest is $\{m_4, m_5\}$. In practice, the cost variation between these metals is sufficiently large that given a realistic budget often only one metal is valid for any particular blade row.

Root type - Discrete Categorical

Every blade row is anchored into the shaft (rotor) or the casing (stator) by use of a root. In addition to being able to specify the size of a root, one can also pick the type of root used. The three basic root types are illustrated in Figure 1.2.8 (taken from [2]).

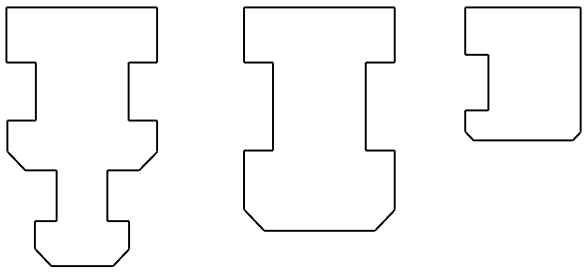

Figure 1.2.8 – We show three basic root types above. From left to right, these are called the 'Double T root', the 'T root', and the 'L root'. (Image taken from [2, Figure 6].)

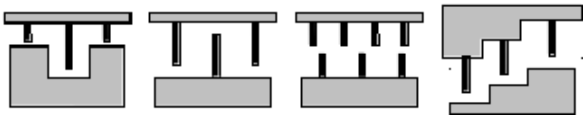

Figure 1.2.9 – We show above four different sealing types that can be used. The sealing type on the left is called a labyrinth seal, the central types are called fin seals, and the sealing type on the right is called a step seal. (Image taken from [2, Figure 6].)

Sealing type - Discrete Categorical

As steam is driven through the turbine, we wish for it to push at the rotor blades and thus cause the shaft to rotate. Since the rotor blade rows must be free to rotate, it means that there must be some space left between the end of the blade row and the casing. However, if too much steam passes through this region, then we do not harvest any energy from it. Therefore, we use one of several sealing types to reduce the steam that can pass the blade row in this way. In practice, each sealing type is designed for a specific situation, and therefore in most cases the choice of sealing type is automatic. We illustrate four different options for the sealing type in Figure 1.2.9 (taken from [2]).

Blade profiles - Discrete Categorical

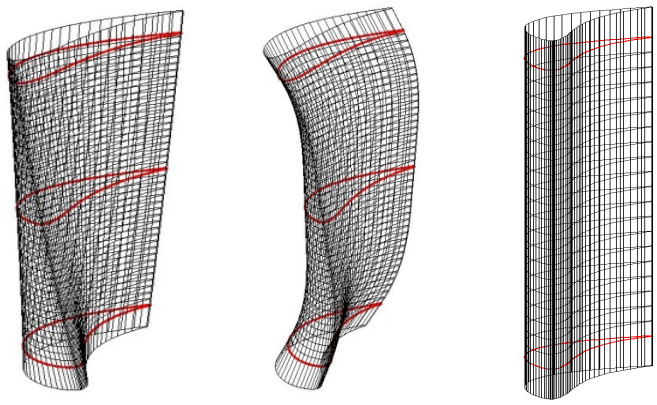

Figure 1.2.10 – Three commonly used blade profiles. (Image taken from [2, Figure 6].)

The ideal shape of the stator blade row is considered known by Siemens. As such, the only way that these blade rows are modified is by changing the stagger angle. For the rotor blades however, we must consider what the ideal blade shape is. The problem here is that shape optimisation is a hard problem in and of itself, and it would be costly to carry out such a procedure for each individual blade row.

Instead, Siemens approximates a blade profile by the shape of the blade row at three positions on the blade as illustrated in red in Figure 1.2.10 (taken from [2]). The shape at each of these three positions is taken from a finite set of options which have been

determined over the years. By default, the shapes used depend on the choice of independent continuous variables. This dependency is based on Siemens' prior experience at constructing profiles. Hence the blade profile is a dependent discrete variable.

1.3 Formulating an optimisation problem

In the previous section, we described how Siemens parametrise their shape optimisation problem, thus converting it into the problem of optimising efficiency with respect to a number of continuous and discrete variables. Writing $\boldsymbol{x} \in \Omega_1 \subseteq \mathbb{R}^n$ to be a vector of continuous parameters, $\boldsymbol{y} \in \Omega_2 \subseteq \mathbb{Z}^m$ to be a vector of discrete parameters, $\boldsymbol{\alpha}$ to be a vector of customer specified constants, and f to be the efficiency function, we express Siemens' problem in the following form:

$$\underset{\substack{\boldsymbol{x}\in\Omega_1(\boldsymbol{\alpha})\subseteq\mathbb{R}^n,\\ \boldsymbol{y}\in\Omega_2(\boldsymbol{\alpha})\subseteq\mathbb{Z}^m}}{\text{minimize}} \quad f(\boldsymbol{x},\boldsymbol{y};\boldsymbol{\alpha}) \tag{1.1}$$

In this section, we describe 3dv1d, the software tool which Siemens have provided us with to compute a steam turbines efficiency given specified discrete and continuous variables, and refine our formulation of Problem (1.1) by addressing the following points:

1. How do we access the 3dv1d subroutine which computes the efficiency?
2. How many variables need to be specified in order to compute the efficiency?
3. How many constraints exist and what type are they?
4. How much time may an optimisation solver take while still being useful?

How do we access the subroutine which computes efficiency?

To run 3dv1d, we need two particular input files, the 3dv1d.'testcase'.dta file and the restart.'testcase'.in file. (Here the text 'testcase' refers to the arbitrary name given to the specific problem instance). The .dta file contains information which specifies some basic requirements for the steam turbine, such as the space available, in addition to

providing the values for each discrete variable. Meanwhile, the 'restart' file contains a list of continuous variables, along with an initial value for each of them.

When we run 3dv1d (having providing this pair of files) the software package will load a large number of data files in order to set up the problem, a process which takes a few seconds. Once this is done, a specific script called 'v2opt76' is run. This script is the only part of 3dv1d which we have the power to edit. It is here than an external optimisation solver is imported, and then applied to the continuous variables (the discrete variables are fixed as these were required for setting up the problem).

In short, Siemens have split Problem (1.1) into the following two subproblems:

$$\underset{\boldsymbol{y}\in\Omega_2(\boldsymbol{\alpha})\subseteq\mathbb{R}^n}{\text{minimize}} \quad g(\boldsymbol{y};\boldsymbol{\alpha}), \tag{1.2}$$

where

$$g(\boldsymbol{y};\boldsymbol{\alpha}) = \underset{\boldsymbol{x}\in\Omega_1(\boldsymbol{\alpha})\subseteq\mathbb{R}^n}{\text{minimize}} \quad f(\boldsymbol{x},\boldsymbol{y};\boldsymbol{\alpha}), \tag{1.3}$$

Siemens are not currently able to solve Problem (1.1) algorithmically. Rather, they use an external optimisation software to solve Problem (1.3), while relying on expert experience to estimate a good solution to Problem (1.2).

Remark. *Reiterating what was said above in light of our notation, in order to run 3dv1d, we require a .dta input file which specifies the values of* $\boldsymbol{y}$ *and* $\boldsymbol{\alpha}$*. Then within the v2opt76 script, we may attempt to solve Problem (1.3).*

How many variables are there?

The number of continuous variables taken by 3dv1d depends on the size of the steam turbine to be built, specifically on the number of *stages* (see Section 1.2) which is specified in the .dta file. Using the notation above, this means that n, the number of continuous variables is actually a natural number valued function of $\boldsymbol{\alpha}$. Siemens have provided us with 5 different sample .dta files (or values of $\boldsymbol{\alpha}$) which describe 5 different test problems containing 1, 2, 9, 9 and 27 stages respectively. These testcases involves roughly 20, 40, 100, 100 and 300 variables respectively.

How many and what type of constraints are involved?

An optimisation problem defined by 3dv1d will contain a small number of analytic linear constraints, along with a large number of nonlinear simulation based constraints (using the terminology of Digabel and Wild [6]) all of which are inequality constraints. The number of linear constraints is roughly twice the number of stages in the steam turbine. However, we lack even an approximate rule for the number of nonlinear constraints. For our two 9 stage test problems, there are roughly 300 nonlinear constraints while the 27 stage test problem has closer to 1000.

In short, we may further refine Problem (1.3) into the form:

$$\begin{aligned} g(\boldsymbol{y};\boldsymbol{\alpha}) = \underset{\boldsymbol{x}\in\mathbb{R}^{n(\boldsymbol{\alpha})}}{\text{minimize}} \quad & f(\boldsymbol{x},\boldsymbol{y};\boldsymbol{\alpha}) \qquad\qquad , \\ \text{subject to} \quad & c_j(\boldsymbol{x},\boldsymbol{y};\boldsymbol{\alpha}) \leq 0, \quad j \in \mathcal{I}(\boldsymbol{\alpha}), \end{aligned} \tag{1.4}$$

where $\boldsymbol{c}$ is a nonlinear vector valued function, and $\mathcal{I}$ is index set of constraints. While some of these constraints are linear, we write Problem (1.4) as if they are all nonlinear for ease of notation.

In addition to these known constraints, we note that the 3dv1d software package is designed with a very practical application in mind: that of approximating the design of a steam turbine. While this means we can expect a sensible approximation of a reasonable steam turbine design, 3dv1d was never designed to return a value given nonsensical input (for example a blade with negative length is nonsense). If we fail to acknowledge this and apply 3dv1d as an objective function blindly, we will encounter some problems.

In practice, when particular inputs are fed into 3dv1d which correspond to nonsensical steam turbine designs, 3dv1d will exit and return an error. In other words, there exists some *hidden* constraints (by *hidden* we are referring to the Taxonomy of Digabel and Wild [6]), such that 3dv1d is incapable of yielding a meaningful output if these constraints are violated. This means that our calculation of $g(\boldsymbol{x},\boldsymbol{y};\boldsymbol{\alpha})$ (see Problem (1.3)) will fail if we trial any value of $\boldsymbol{x}$ which violates these hidden constraints while solving

Problem (1.3). All we know about these hidden constraints is that no point which satisfies all known constraints violates these hidden ones, meaning that the feasible domain is contained within the domain which satisfies the hidden constraints.

This is highly problematic given that most of our constraints are (again using the terminology of Digabel and Wild) simulation-based. This being the case, it is not possible for us to construct a feasible steam turbine design a priori. Hence, we must hope that our starting point does not fail 3dv1d's plausibility check, resulting in 3dv1d closing down. Moreover, we need that any optimisation solver we use navigates towards a feasible solution without ever evaluating a nonsensical steam turbine design along the way.

There are two contexts in which this is particularly troublesome. First of all, some numerical optimisation solvers involve the random sampling of nearby points. To use such a method (which we do) runs the risk of sampling a point that violates one of the hidden constraints.

The second problem comes from the fact that we don't have many functions which are ideal for numerical testing. In general, when testing the effectiveness of an optimiser, one either needs many different functions, or many different starting points. Given that we only have five .dta files (meaning five values of $\boldsymbol{\alpha}$), we need to be able to generate a multitude of starting points when testing an optimiser on 3dv1d. A natural step is to restart 3dv1d whenever a randomly generated starting point violates a hidden constraint and try again. In our experience however, we have not found this approach to be effective as nearly all randomly sampled points which we trialled violated the hidden constraints. In Section 1.4.3, we explain how this problem is addressed.

How fast does a solver need to be?

From the time that Siemens get the specifications for a steam turbine (that is the value of $\boldsymbol{\alpha}$) from a customer, they have roughly three days to produce a produce a steam turbine design and quote a price. Since each set of customer constraints: $\boldsymbol{\alpha}$ defines a unique problem, this is not work which can be done in advance. Therefore, any proposed

solution to Problem (1.1) must be solvable within three days.

As stated before, Siemens currently approach Problem (1.1) by splitting it into Problems (1.2) and (1.4), relying on expert opinion to solve the former and using an external optimisation software to solve the latter. However, Siemens' long term goal is to solve the entirety of Problem (1.1) automatically. Therefore, any solution we propose to Siemens, either for Problem (1.2) or (1.4) must keep this time constraint in mind. For this reason, we have taken 2 hours to be the soft upper bound for the time available to solve Problem 1.4.

1.4 Additional details

Having formulated the problem which we are trying to solve, in this section we provide information on such topics which we think are important to know before proceeding to later chapters. These include:

1. Do we attempt to solve both Problem (1.2) and (1.3)?
2. How long does it take to compute f (see Problem (1.4))?
3. How do we randomly sample points which don't violate the hidden constraints?
4. What modifications need to be made to 3dv1d first?

1.4.1 Which problem did we solve?

Although the goal for this project was to address both Problems (1.2) and (1.4), ultimately we have only worked on Problem (1.4). The reason for this is that we agree with Siemens' approach to split Problem (1.1) into (1.2) and (1.4).

However, since the statement of Problem (1.2) presupposes that Problem (1.4) has already been solved, we did not view there to be any point in working on Problem (1.2) until being finished with Problem (1.4). Since solving Problem (1.4) had more complications then initially expected, we ultimately did no work on Problem (1.2). For

the remainder of this thesis, we no longer view the efficiency function f to depend on $\boldsymbol{\alpha}$, but rather view each pair of discrete variables $\boldsymbol{y}$ and specifications $\boldsymbol{\alpha}$ to define a separate problem instance.

1.4.2 How long does it take to estimate efficiency?

It should not be a surprise that the amount of time required to compute the efficiency depends on the number of variables, which in turn depends on $\boldsymbol{\alpha}$, the vector of customer specifications. As stated previously, Siemens have provided us with five different .dta files, each containing a different pair of discrete variables and customer specified constants $(\boldsymbol{y}, \boldsymbol{\alpha})$ and each defining a separate problem.

For any of these problems, 3dv1d can compute the efficiency is significantly less than a second. However, we have been cautioned by Siemens that the two smallest sample functions are too small to have practical meaning. Therefore, while we sometimes use these functions to check that 3dv1d is working as we'd expect, we can't draw any meaningful conclusions when using these for testing. Therefore, we have not used these two smallest problems, or testcases in any numerical experiments described in this thesis.

Moreover, we don't use the largest sample problem for any rigorous testing later because its cost, while small, is still orders of magnitude larger than the others. For perspective, given the two middle sized test problems (each of which have 9 stages), we can numerically approximate their gradients in roughly 0.1 seconds. The largest test function (which has 27 stages) requires closer to 5 seconds to approximate the gradient.

For this reason, and the fact that the higher dimensionality of the largest test problem makes it harder, we have found it impractical to include in our testing. Most of our numerical experiments were conducted on the time scale of 3 days or less. We acknowledge that a longer numerical experiment could in principle have been conducted, but we did not do so. In short, we only use the two middle sized problems for testing purposes. We refer to these test functions as tc001 and tc004.

1.4.3 Generating random starting points

While we lack any concrete rules as to how to avoid points corresponding to nonsensical steam turbine designs, Siemens has provided us with some guidelines which, if followed, reduce the chances of randomly sampling one. These rules apply particularly to the variable types: inner flow path diameter description, and the stage load distribution (see Section 1.2).

Ultimately both of these variable types must be distributed in the same fashion in that there is one of these variables for each *stage* (recall Section 1.2) of the turbine. Thus each of these variable types may be written in the form: $\{x_1, x_2, \ldots, x_n\}$, where the subscript refers to steam turbines stage, and each x_i is contained within a common interval $l \leq x_i \leq u, \ \forall i$. We arrange the x_i's such that the dummy variables x_0 and x_{n+1} are given by $x_0 = l$, $x_{n+1} = u$ (the upper and lower bounds respectively), and the remainder are positioned roughly on the line between these points. When generating a randomly sampled point, we first fit the line, and then randomly perturb each x_i by a margin small enough to ensure the general shape of the distribution holds.

If the stage load distribution and the inner flow path diameter is generated in this way, while the remaining variables are sampled from within the set which is feasible with respect to bound and linear constraints, then there is a good chance that the resulting randomly generated steam turbine design will not violate the hidden conditions.

1.4.4 Practical modifications to 3dv1d

Recall from Section 1.3 that in order to actually calculate the efficiency of a steam turbine, we must use a subroutine of 3dv1d which may only be accessed from within the script v2opt76. The first body of work we completed for this project was to implement two interface methods in order to call the efficiency function from an external python script. Here we refer to these two methods as the 'fast' method and the 'slow' method.

When evaluating the objective function given a set of variables using the slow method, we instruct python to write two input files required by 3dv1d, thus speci-

fying both the discrete variables and customer specifications $(\boldsymbol{y}, \boldsymbol{\alpha})$ in the .dta file and the continuous variables in the restart file. Next we command 3dv1d to start up from python, modify the v2opt76 script to evaluate the objective function at the point indicated by the restart file, and write the result as an output file which is read by python.

The merit of this approach is that it gives us the freedom to modify discrete variables within the context of a python script. However, this approach is excessively time consuming given that the time required by 3dv1d to read its list of input files (those other than the one containing discrete variables) and construct the problem greatly exceeds the time required to evaluate the efficiency of the turbine itself. By using the slow method, each function evaluation takes a few seconds rather than a fraction of a second. While this approach was not useful for the direction this project ultimately went it, this development would have been necessary in order to attempt to solve Problem (1.2).

The second, or fast approach involves creating a dynamic interaction between the script v2opt76 and the python package. This is accomplished by starting 3dv1d independently of python and instructing it to do nothing until it receives an input file (written from python) containing instructions. Once this file is read, 3dv1d carries out these instructions, writes the outcome to an output file, and then resumes waiting, all within the v2opt76 script. Meanwhile, the python package is designed such that in order to call the objective, we write the file containing instructions for 3dv1d, and then wait for the output file to appear. The merit of this approach is that we do not need to reload all of the input files for every function evaluation, meaning that the cost of each function evaluation is comparable to if we were calling the objective function from within the v2opt76 script (in fortran). On the other hand, since we load 3dv1d only once, the set of dimensional and discrete variables is fixed. Therefore, this method is exactly what we need for developing solutions to Problem (1.4)

While these interface methods usually work for running a single algorithm on 3dv1d, we have not managed to make these methods entirely stable. In our experience, we have found that experiments using this interface work well on a time scale of 1 week or less, but are unreliable over longer periods of time. Ultimately, we resigned ourselves to shorter experiments as an alternative to spending time in developing these interfaces.

Chapter 2

Applying SQP to 3dv1d

The task of finding an optimal steam turbine design (given fixed discrete variables) may be written in the form of the general nonlinear programming (NP) problem [7]:

$$\begin{aligned} \underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad & f(\boldsymbol{x}) \\ \text{subject to} \quad & c_j(\boldsymbol{x}) \leq 0, \quad j \in \mathcal{I}, \end{aligned} \tag{2.1}$$

where f, the objective function, is the efficiency of the steam turbine as computed by 3dv1d, $\boldsymbol{c}$ is the constraint function, and $\mathcal{I}$ is the finite index set for inequality constraints.

There exist several professional software packages designed to solve Problem (2.1) including: Knitro [8], Tomlab [9], and Lancelot [10]. Currently, Siemens makes use of the nonlinear programming routines in the Numerical Algorithms Group (NAG) library [4]. In the chapter of the NAG library manual on Optimisation (code E04), in Section 4 (Choice of available functions), there is a flowchart to help with finding the solver most suited to the problem. Recommended for dense problems with constraints are the functions e04ucc, e04ucf and e04wdc, of which e04ucf is the function that Siemens uses.

We begin this chapter in Section 2.1 by outlining the main definitions and results required to understand sequential quadratic programming (SQP), which is the general class of algorithms to which e04ucf belongs. In Section 2.2 we discuss some inputs and outputs for NAG which we have found to be of particular significance when applying it to 3dv1d. Finally, in Section 2.3 we apply e04ucf to 3dv1d and discuss the consequences.

2.1 Sequential quadratic programming

The NAG function e04ucf is a sequential quadratic programming (SQP) algorithm, which is a specific example from a more general class of algorithms called Line Search Methods. In this section, we begin by presenting some definitions and results from the theory of constrained optimisation following Nocedal and Wright [11]. After this, we outline how line search methods and more specifically SQP work, and what conditions must be satisfied to ensure their convergence.

2.1.1 The Theory of Constrained Optimisation

To solve Problem 2.1, we must understand what it means to *minimise* a constrained function. We begin by defining what it means for a point to be feasible.

Definition 2.1 (Feasible Set [11])**.** *Given Problem 2.1, we define the* set of feasible points*:* Ω *to be*

$$\Omega := \{\boldsymbol{x} \in \mathbb{R}^n \mid c_j(\boldsymbol{x}) \leq 0 \ \forall j \in \mathcal{I}\}. \tag{2.2}$$

When solving Problem 2.1 to obtain a steam turbine design, the most important property that a solution must have is that it be feasible. Following this, we also wish for a solution to be *optimal*, a concept which we now define.

Definition 2.2 (Global Minimiser [11])**.** *The vector* $\boldsymbol{x}^*$ *is called a* global optimiser *of Problem 2.1, or* optimal*, if* $\boldsymbol{x} \in \Omega$ *and* $f(\boldsymbol{x}^*) \leq f(\boldsymbol{y}),\ \forall \boldsymbol{y} \in \Omega$*.*

While it is ideal to find an optimal solution, this can be very difficult and costly unless the problem has special structure such as convexity. Instead, we relax the notion of optimal to the more achievable condition of *locally optimal*.

Definition 2.3 (Local Minimiser [11])**.** *We call the vector* $\boldsymbol{x}^*$ *a* local optimiser *of Problem 2.1, or* locally optimal*, if* $\boldsymbol{x}^* \in \Omega$ *and there exists* $\mathcal{N}$*, a neighbourhood of* $\boldsymbol{x}^*$*, such that* $f(\boldsymbol{x}^*) \leq f(\boldsymbol{y}),\ \forall \boldsymbol{y} \in \mathcal{N} \cap \Omega$*.*

In order to construct an algorithm which locates a locally optimal solution to Problem 2.1, we need the ability to identify when a point is locally optimal. In the definitions

which follow, we work towards defining the KKT conditions, which are *necessary* (but not sufficient) conditions for identifying a local minimiser.

Definition 2.4 (LICQ)**.** *Given the point* $\boldsymbol{x}$ *and* active set*:* $\mathcal{A}(\boldsymbol{x}) = \{j \in \mathcal{I} \mid c_j(\boldsymbol{x}) = 0\}$*, we say that the* linear independence constraint qualification (LICQ) *holds if the set of active constraint gradients* $\{\nabla c_j,\ j \in \mathcal{A}(\boldsymbol{x})\}$ *is linearly independent.*

In practice, we assume that the LICQ holds, and only question when a solver terminates unexpectedly. If it does apply, then we can define first order optimality conditions for Problem 2.1.

Definition 2.5 (KKT Conditions)**.** *The point* $\boldsymbol{x}^*$ *is said to satisfy the Karush-Kuhn-Tucker (KKT) conditions for Problem 2.1 if there exists a vector* $\boldsymbol{\lambda}$ *such that:*

$$\frac{\partial f}{\partial x_i}(\boldsymbol{x}^*) = -\sum_{j \in \mathcal{I}} \lambda_j \frac{\partial c_j}{\partial x_i}(\boldsymbol{x}^*), \qquad \forall i \in \{1, 2, \ldots, n\} \tag{2.3a}$$

$$c_j(\boldsymbol{x}^*) \leq 0, \qquad \text{for all } j \in \mathcal{I}, \tag{2.3b}$$

$$\lambda_j \geq 0, \qquad \text{for all } j \in \mathcal{I}, \tag{2.3c}$$

$$\lambda_i c_j(\boldsymbol{x}^*) = 0 \qquad \text{for all } j \in \mathcal{I}. \tag{2.3d}$$

If such a $\boldsymbol{\lambda}$ *exists, we call* $\boldsymbol{x}^*$ *a KKT point.*

Theorem 1 ([11])**.** *Suppose that* $\boldsymbol{x}^* \in \Omega$ *is a local minimiser of* f*. If the functions* f *and* $\boldsymbol{c}$ *are all continuously differentiable and the LICQ condition holds at* $\boldsymbol{x}^*$*, then the KKT conditions are satisfied at* $\boldsymbol{x}^*$*.*

Since Theorem 1 guarantees that if $\boldsymbol{x}^*$ is a feasible local optimiser of Problem 2.1, then the KKT conditions are satisfied, we need an algorithm which is guaranteed to converge to a KKT point.

Remark. *It is important to note that the KKT conditions are only* 1^{st} *order optimality conditions. To truly guarantee that* $\boldsymbol{x}^*$ *is a local optimiser, second order conditions are needed. However, this additional theory is ultimately not relevant to this thesis, and thus we neglect it. For more information on second order conditions, see [11, Chapter 12].*

2.1.2 A practical sequential quadratic programming algorithm

The SQP algorithm: e04ucf (implemented by NAG) which we have used is based on the software package: NPSOL[12]. This specific implementation of SQP is an instance of a line search method (LSM) which are the class of algorithms that may be written generically in the form of Algorithm 1. Such algorithms consist of two main steps: finding a search direction and determining a step length, which we outline in greater detail within Sections 2.1.2.1 and 2.1.2.2 respectively. It's also important to note that the variant of SQP we employ is a Quasi-Newton (QN) method. This means that instead of using second order derivative information for finding a search direction (see Section 2.1.2.1), we instead use an approximation which we update during the algorithms run. We explain the details of this process in Section 2.1.2.3. In Section 2.1.2.2 we introduce a penalty parameter ρ which we use to aid in finding a feasible solution. We describe how this parameter is handled in Section 2.1.2.4. Finally in Section 2.1.2.5 we provide pseudocode for e04ucf/NPSOL in addition to stating its convergence theorem.

Algorithm 1: General line search method

```
   /* Initialisation:                                              */
   input : f // Objective function
           1 x_0 // Starting point
           2 ϵ // Stationary Tolerance
 3 Determine p_0, a direction of descent of f from x_0.
 4 k = 0;
 5 while ‖p_k‖ ≥ ϵ do
 6 |   Determine step length: α_k.
 7 |   x_{k+1} = x_k + α_k p_k // Update iterate
 8 |   k = k + 1
 9 |   Determine p_k, a direction of descent of f from x_k.
   output: x_k.
```

Remark. *Line 6 of Algorithm 1, where the step length α_k is calculated, is called a Line Search. This should not be confused with a LSM.*

2.1.2.1 Determining a descent direction

The first step in a line search method is determining a search direction. In the unconstrained case, this step is simple with $-\nabla f(\boldsymbol{x}_k)$ being a suitable direction of descent whenever $\|\nabla f(\boldsymbol{x}_k)\| > \epsilon$. For Problem 2.1 though, we need to take both the constraints and objective function into account. We do this by solving the following Quadratic Programming problem:

$$\begin{aligned} \boldsymbol{p}_k = \ & \underset{\boldsymbol{d} \in \mathbb{R}^n}{\arg\min} && \nabla f(\boldsymbol{x}_k)^T \boldsymbol{d} + \tfrac{1}{2} \boldsymbol{d}^T \boldsymbol{H}_k \boldsymbol{d} \\ & \text{subject to} && \nabla c_j(\boldsymbol{x}_k)^T \boldsymbol{d} + c_j(\boldsymbol{x}_k) \leq 0, \quad j \in \mathcal{I} \end{aligned} \tag{2.4}$$

Problem 2.4 involves the matrix $\boldsymbol{H}_k$ which has not yet been defined. This is the Quasi-Newton matrix which combines approximations for the second derivatives of f and $\boldsymbol{c}$. We discuss this further in Section 2.1.2.3.

While Problem 2.4 is the conceptually simplest Quadratic Programming problem to solve to obtain a descent direction, there exist modifications of this which perform better in practice. For more details, see [11, Chapter 18]. However, the Quadratic Programming sub-problem solved within e04ucf is equivalent that Problem 2.4.

2.1.2.2 Determining a step length

Once a descent direction $\boldsymbol{p}_k$ has been determined, the next task is to determine a step length. Here there are two qualitatively different approaches which we might take: exact line search (ELS) and inexact line search (ILS).

Definition 2.6 (Exact line search [11])**.** *Given the merit function $\phi : \mathbb{R}^n \to \mathbb{R}$, a point $\boldsymbol{x} \in \mathbb{R}^n$ and a descent direction $\boldsymbol{p}$, an exact line search (ELS) is a type of line search algorithm which returns a step length α that satisfies:*

$$\alpha_k = \underset{\alpha \geq 0}{\arg\min} \quad \varphi(\boldsymbol{x} + \alpha \boldsymbol{p}). \tag{2.5}$$

While we have defined ELS to return the global minimum of the 1D function φ, in the

context of this thesis, we do no concern ourselves with finding the global minimum so long as we find a local minimum (we do not care about which local minimum).

While conceptually intuitive, in practice this approach proves to be computationally expensive and does not perform better than the second approach: inexact line search (ILS). The idea behind ILS is that when we move away from $\boldsymbol{x}_k$, the shape of f and $\boldsymbol{c}$ may change, meaning that the work required for ELS has limited use. Instead, we choose a step length α_k which satisfies the Wolfe Conditions. Before we can define these, we first need the notion of the *directional derivative.*

Definition 2.7 (Directional Derivative [13])**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function and $\boldsymbol{x}, \boldsymbol{d} \in \mathbb{R}^n$, then we define the directional derivative to be:*

$$f'(\boldsymbol{x}; \boldsymbol{d}) = \lim_{t \to 0^+} \left(\frac{f(\boldsymbol{x} + t\boldsymbol{d}) - f(\boldsymbol{x})}{t} \right).$$

Definition 2.8 (Wolfe Conditions [11])**.** *Given the 1D merit function $\varphi : \mathbb{R} \to \mathbb{R}$, the step length α_k is said to satisfy the* Weak Wolfe Conditions *if it holds that:*

$$\varphi(\alpha_k) \leq \varphi(0) + c_1 \alpha_k \varphi'(0), \tag{2.6a}$$

$$\varphi'(\alpha_k) \geq c_2 \varphi'(0), \tag{2.6b}$$

where $0 < c_1 \ll c_2 < 1$ are constants.

While more details may be found in [11], the Wolfe Conditions ensure that each iteration yields a sufficient decrease of the merit function. Moreover, satisfying the Wolfe Conditions has beneficial properties for the QN matrix approximation, which we discuss in the following section.

In NAG's SQP implementation: e04ucf, we use Augmented Lagrangian merit function to assess the quality of a given point:

$$\mathcal{L}(\boldsymbol{x}_k, \boldsymbol{\lambda}_k, \boldsymbol{t}; \rho) := f(\boldsymbol{x}_k) + \sum_{j \in \mathcal{I}} \lambda_j (c_j(\boldsymbol{x}_k) + t_j) + \frac{1}{2} \rho \sum_{j \in \mathcal{I}} (c_j(\boldsymbol{x}_k) + t_j)^2. \tag{2.7}$$

This merit function makes use of variables which we have not yet defined: $\boldsymbol{\lambda}_k$, $\boldsymbol{t}$ and ρ. We define these now.

The vector $\boldsymbol{\lambda}_k$ found in Equation (2.7) refers to the dual variables (which also appear in Definition 2.5). There exist one of these variables for each constraint, and once the KKT conditions are satisfied, the magnitude of λ_i measures how restrictive the constraint c_j is to our optimal solution. Whenever we use a quadratic programming solver to solve Problem 2.4, we receive a vector of dual variables $\boldsymbol{\mu}$ as an output. While we do not take $\boldsymbol{\lambda}_{k+1} = \boldsymbol{\mu}$, we will use $\boldsymbol{\mu}$ to construct $\boldsymbol{\lambda}_{k+1}$.

Meanwhile, $\boldsymbol{t}$ is a vector of slack variables which enable the merit function $\mathcal{L}$ to work even when the constraints are inequality as opposed to equality constraints. For each iteration k, we compute $\boldsymbol{t}$ as follows:

$$t_j(\boldsymbol{x}_k) = \begin{cases} \max(0, -c_j(\boldsymbol{x}_k)) & \text{if } \rho = 0; \\ \max(0, -c_i(\boldsymbol{x}_k)) - \lambda_j/\rho & \text{otherwise,} \end{cases} \quad \forall j \in \mathcal{I}. \tag{2.8}$$

Finally, ρ is a penalty parameter. How ρ is handled warrants its own section and is discussed in Section 2.1.2.4.

Given a $(\boldsymbol{x}_k, \boldsymbol{\lambda}_k)$ variable pair, we compute $\boldsymbol{t}$ using Equation (2.8). Next we solve Problem 2.4 to find the search direction $\boldsymbol{p}_k$, and the alternative vector of dual variables $\boldsymbol{\mu}$. Finally, given the step length α, we compute a trial point $(\overline{\boldsymbol{x}}, \overline{\boldsymbol{\lambda}}, \overline{\boldsymbol{s}})$ using:

$$\begin{pmatrix} \overline{\boldsymbol{x}} \\ \overline{\boldsymbol{\lambda}} \\ \overline{\boldsymbol{t}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_k \\ \boldsymbol{\lambda}_k \\ \boldsymbol{t} \end{pmatrix} + \alpha \begin{pmatrix} \boldsymbol{p}_k \\ \boldsymbol{\mu} - \boldsymbol{\lambda}_k \\ \boldsymbol{r} \end{pmatrix}, \tag{2.9}$$

where $\boldsymbol{r}$ is evaluated from:

$$r_j = -c_j(\boldsymbol{x}_k) - \nabla c_j(\boldsymbol{x}_k)^T \boldsymbol{p}_k - s_j, \ \forall j \in \mathcal{I}. \tag{2.10}$$

In short, starting with $\boldsymbol{x}_k$ and $\boldsymbol{\lambda}_k$, we compute $\boldsymbol{t}$, $\boldsymbol{\mu}$, $\boldsymbol{p}_k$ and $\boldsymbol{r}$, ultimately using these

terms to define our 1D line search function as:

$$\varphi(\alpha;\rho) = \mathcal{L}(\boldsymbol{x}_k + \alpha\boldsymbol{p}_k, \boldsymbol{\lambda}_k + \alpha(\boldsymbol{\mu} - \boldsymbol{\lambda}_k), \boldsymbol{t} + \alpha\boldsymbol{r}; \rho). \tag{2.11}$$

Given φ, we now need to apply a line search algorithm to find a step length α which satisfies the Wolfe Conditions (Definition 2.8). The specific algorithm implemented by NAG (which is based on NPSOL) makes use a *safeguarded cubic interpolation* method. We do not describe it here, both because we could not find a copy of the paper which describes it [14], but also because inexact line search algorithms are not the focus of this chapter. Rather we content ourselves with the fact that for each iteration k, a step length α_k satisfying Definition 2.8 can be reliably obtained.

2.1.2.3 Updating the Quasi-Newton Matrix

Quasi-Newton (QN) methods are a class of algorithms where the second derivative is approximated using prior information instead of being computed directly. While these methods might not converge as quickly as they would given the true 2$^{\text{nd}}$ derivative, they have the advantages of being computationally cheaper, and not relying on 2$^{\text{nd}}$ derivative information being available. Given our context, these properties are desirable.

When applying QN methods to constrained optimisation, we do not use the QN matrix $\boldsymbol{H}_k$ to approximate $\nabla^2 f(\boldsymbol{x}_k)$, but rather to approximate $\nabla^2_{\boldsymbol{x}}\mathcal{L}(\boldsymbol{x}_k, \boldsymbol{\lambda}_k, \boldsymbol{t}; \rho)$ (recall Equation (2.7)) given the values of $\boldsymbol{\lambda}_k$ and t which we described how to compute in Section 2.1.2.2.

In order for the Quadratic Programming sub-problem (Problem 2.4) to have a bounded solution, we require that the QN matrix $\boldsymbol{H}_k$ be positive definite within the null space of constraints for all iterations. This must apply regardless of whether or not $\nabla^2_{\boldsymbol{x}}\mathcal{L}(\boldsymbol{x}_k, \boldsymbol{\lambda}_k, \boldsymbol{t}; \rho)$ is positive definite or not.

We ensure that this is the case by updating the QN matrix in the following way. Given $\boldsymbol{H}_k$, $\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$ and $\boldsymbol{y}_k = \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k) + \sum_{j\in\mathcal{I}} \lambda_j \nabla(c_j(\boldsymbol{x}_{k+1}) - c_j(\boldsymbol{x}_k))$ (see

[15, Equation 6.7]), we compute $\boldsymbol{H}_{k+1}$ using the BFGS update [11, 2.19]:

$$\boldsymbol{H}_{k+1} = \boldsymbol{H}_k - \frac{\boldsymbol{H}_k \boldsymbol{s}_k \boldsymbol{s}_k^T \boldsymbol{H}_k^T}{\boldsymbol{s}_k^T \boldsymbol{H}_k \boldsymbol{s}_k} + \frac{\boldsymbol{y}_k \boldsymbol{y}_k^T}{\boldsymbol{y}_k^T \boldsymbol{s}_k}. \tag{2.12}$$

Remark. *We have written the formula showing how $\boldsymbol{H}_k$ is updated. In practical algorithms, it is the matrix $\boldsymbol{H}_k^{-1}$ which is stored and updated instead.*

This iteration process ensures that $\boldsymbol{H}_{k+1}$ is positive definite provided that it is well defined and $\boldsymbol{H}_k$ is positive definite. For $\boldsymbol{H}_{k+1}$ to be well defined, we require that $\boldsymbol{y}^T \boldsymbol{s} > 0$. Fortunately, this requirement matches the second Wolfe Condition (see Definition 2.8). Therefore, if we have selected a step length which satisfies the Wolfe Conditions, we are assured that $\boldsymbol{H}_{k+1}$ will be well defined. Finally, we initialise $\boldsymbol{H}_0 = \boldsymbol{I}_n$ with the identity matrix, which is positive definite, thus ensuring that $\boldsymbol{H}_k$ will be positive definite for all iterations .

2.1.2.4 Updating the penalty parameter ρ

The penalty parameter ρ is a value which plays a role in the definition of the Augmented Lagrangian function (see Equation (2.7)) which we use to construct our line search function in Section 2.1.2.2. The purpose of this parameter is to aid SQP in converging to a feasible solution. With large values of ρ, we force the algorithm to navigate to the feasible set. However, according to the experience of Gill et al [16], small values of ρ result in SQP running more quickly. Therefore, ρ is initialised to 0 and then slowly increased as it is needed. The question is, how big does ρ need to be at each iteration? Gill et al [16, Lemma 4.3] have addressed this question in the following result:

Lemma 2. *There exists $\hat{\rho} > 0$ such that:*

$$\varphi'(0; \rho) \leq -\frac{1}{2} \boldsymbol{p}_k^T \boldsymbol{H}_k \boldsymbol{p}_k, \tag{2.13}$$

for all $\rho \geq \hat{\rho}$, for every iteration k. Here φ is the line search function which we defined

in Equation (2.11).

Motivated by the proof of Lemma 2 (which we do not show here), at each iteration we compute the value:

$$\hat{\rho_k} = \frac{2\|\boldsymbol{\mu} - \boldsymbol{\lambda}_k\|}{\|\boldsymbol{c}(\boldsymbol{x}_k) + \boldsymbol{t}\|}, \tag{2.14}$$

where $\boldsymbol{\lambda}_k$ is the current value of the dual variables, $\boldsymbol{\mu}$ is the vector of dual variables obtained from solving Problem 2.4, and $\boldsymbol{t}$ is the vector of slack variables which we compute using Equation (2.8). While $\hat{\rho_k}$ does not necessarily satisfy Lemma 2 for all iterations, it does for the specific iteration k. Motivated by this, we start with $\rho_k = 0$ for $k = 0$, and update as follows:

$$\rho_k = \begin{cases} \rho_{k-1} & \text{if } \varphi'(0, \rho_{k-1}) \leq -\frac{1}{2}\boldsymbol{p}_k^T \boldsymbol{H}_k \boldsymbol{p}_k \\ \max(\hat{\rho_k}, 2\rho_{k-1}) & \text{otherwise} \end{cases}, \tag{2.15}$$

where $\hat{\rho_k}$ is as defined in Equation (2.14).

2.1.2.5 Convergence conditions for SQP

Having given a brief description the key ingredients for e04ucf, NAG's SQP implementation, we write out pseudocode for this algorithm in Algorithm 2. Moreover, we present the following result which captures the circumstances under which convergence is guaranteed.

Theorem 3 ([16]). *Let $\{\boldsymbol{x}_k\}_{k=1}^{\infty}$ be the sequence if iterates generated by Algorithm 2. If the following assumptions are satisfied:*

- *The sequences $\{\boldsymbol{x}_k\}_{k=0}^{\infty}$ and $\{\boldsymbol{x}_k + \boldsymbol{p}_k\}_{k=0}^{\infty}$ are contained in a closed and bounded set $\Omega \subset \mathbb{R}^n$ for all k;*

- *Both f and c_j $(\forall j \in \mathcal{I})$, in addition to their first and second derivatives are uniformly bounded in norm in Ω;*

- *$\boldsymbol{H}_k$ is positive definite, with bounded* condition number *(see [11]), and smallest eigenvalue uniformly bounded away from zero, or there exists $\gamma > 0$ such that, for*

Algorithm 2: A practical SQP algorithm [16]

```
/* Initialisation:                                                    */
input : f // Objective function
        1 c // Constraint function
        2 x_0 // Starting point
        3 ϵ // Stationary Tolerance
4 k = 0;
5 Solve Equation (2.4) to obtain the solution primal and dual variables: p_k
    and μ respectively;
6 ρ_k = 0;
7 while ‖p_k‖ ≥ ϵ do
8     Compute s using Equation (2.8).
9     Update ρ_k using Equation (2.15);
10    if α = 1 satisfies the Wolfe Condition (see Definition 2.8) then
11        Set p_k = 1.
12    else
13        Construct α ∈ (0, 1) which satisfies the Wolfe Conditions using an
            ILS algorithm.
14    Update H_k using Equation (2.12).
15    x_{k+1} = x_k + α_k p_k; λ_{k+1} = λ_k + α_k(μ − λ_k);
16    k = k + 1
17    Solve Equation (2.4) to obtain the solution primal and dual variables:
        p_k and μ respectively;
output: x_k.
```

all k,

$$\boldsymbol{p}^T \boldsymbol{H}_k \boldsymbol{p} \geq \gamma \|\boldsymbol{p}\|^2;$$

- *The linear independence constraint qualification (see Definition 2.4) is satisfied at every iterate $\boldsymbol{x}_k$, and strict complementarity holds;*
- *Problem 2.4 always has a solution;*

then it holds that:

$$\lim_{k \to \infty} \|\boldsymbol{x}_k - \boldsymbol{x}^*\| = 0.$$

Theorem 3 provides a starting point for investigation if e04ucf, NAG's SQP implementation, ever fails to converge. Since we know the conditions which assure convergence, it follows that these are the points to check whenever convergence does not occur.

2.2 NAG's e04ucf software

While in Section 2.1 we defined SQP and stated the circumstances under which convergence to a stationary point is assured, in this section we discuss the practicalities of running NAG's e04ucf function. This involves explaining the key input parameters to e04ucf as well as the outputs which are written to the result file. Our source for this section is the NAG library's documentation [4].

2.2.1 e04ucf inputs

When running NAG's e04ucf function in practice, the user submits an input file in which the values of controllable parameters are specified. In this section, we list through the relevant parameters, explain their meanings, and identify the values which we have used.

- **Function precision**. This parameter is the accuracy to which f and $\boldsymbol{c}$ can be computed. To our knowledge these functions, which are obtained from 3dv1d, are deterministic and built with double precision numbers. We use the value 10^{-14}.

- **Optimality tolerance**. This parameter specifies the accuracy to which the value of f at the approximate solution should match the value of f at the true solution. For this value, we use 10^{-6}.

- **Feasibility tolerance**. This value specifies the maximum violation of the constraint functions which will be tolerated at the approximate solution. We use 10^{-8} for this value.

- **Difference interval**. Since we do not have an analytic derivative for f or $\boldsymbol{c}$, these values must be approximated numerically using finite differences. If no value is specified for this parameter, a subroutine will run which determines a suitable step length for each variable. Otherwise, the derivative will be approximated using the step length provided. For this value, we use 5×10^{-5}.

- **Step limit**. This value is a bound for the initial step length of the line search. Here we use the value of 1.

- **Major Iteration Limit**. This is the maximum number of iterations we permit before stopping the algorithm. While rarely important, we allow for 10000 iterations.

While by no means exhaustive, these are the inputs which we have found to be most important to understand and control.

2.2.2 e04ucf outputs

When the function e04ucf is run, an output file is written which contains information about the progress of the algorithm and the solution obtained. Here we list through and explain information which we can access concerning the operation of e04ucf. At each iteration of the SQP algorithm, the following information is printed out:

- **Merit Function**. The value of the merit function (see Section 2.1.2.2) at each iteration. Note that when there is no constraint violation, then this is equal to the value of the objective function.

- **Step**. The step length taken: α_k.

- **Norm Gz**. The norm of *projected gradient* of the objective function. This value should approach 0 if the algorithm is successful. (Intuitively, this value corresponds to the gradient of f projected into the feasible domain. For more details, see [15].)

- **Violtn**. The quantity by which the constraints are violated at each iteration. This value should approach 0 if the algorithm is successful.

- **Cond Hz**. The *Condition Number* of the QN matrix at each iteration projected into the search direction. This value measures the ratio of the largest eigenvalue of the QN matrix to the smallest eigenvalue, and indicates a problem if too large.

In addition to the outputs we receive per iteration, we also are told whether or not the algorithm has succeeded, and if not why. This is indicated by the value of the output parameter *ifail*. In practice, there are three different values which we actually observe:

- ifail = 1. This indicates that the algorithm has succeeded and the outputted point satisfies the KKT conditions.

- ifail = 3. The algorithm has stopped given that it could not find any feasible point.

- ifail = 6. The algorithm has stopped because it cannot find any point better than $\boldsymbol{x}_k$. However, $\boldsymbol{x}_k$ does not satisfy the KKT conditions.

Having described all relevant inputs and outputs to e04ucf, the next step is to apply it to 3dv1d.

2.3 Applying SQP to 3dv1d

Having provided an overview for the theory of constrained optimisation and specifically sequential quadratic programming in Section 2.1, and the operation of NAG's SQP algorithm in Section 2.2, we now apply e04ucf to the specific problem of 3dv1d. In

Section 2.3.1 we construct a numerical experiment to test e04ucf on 3dv1d and present the outcome. Afterwards in Section 2.3.2, we take a closer look at 3dv1d itself in order to understand the results obtained.

2.3.1 Running a numerical experiment on 3dv1d

In order to assess the effectiveness of e04ucf on 3dv1d, since e04ucf is a deterministic algorithm, we need apply it to 3dv1d many times. This may be accomplished either by applying e04ucf to many testcases, or to many starting points on specific testcases. As mentioned in Section 1.3, we only have access to two testcases which are large enough to be realistic, yet small enough for numerical experiments to be computationally feasible.

For each of these testcases, we run 3dv1d both from the provided starting position, but also from 20 randomly generated starting positions. At present, we are focussed on the suitability of e04ucf for 3dv1d, and not on the solutions themselves. Hence for each starting position, we report only the qualitative outcome, of which the possibilities are:

1. e04ucf converges to a locally optimal feasible solution.
2. e04ucf stalls at a point which does not satisfy the KKT conditions.
3. e04ucf stops due to concluding that it cannot find any feasible points.
4. e04ucf attempts to evaluate 3dv1d at a point corresponding to an nonsensical steam turbine, causing the 3dv1d to close down as it is not designed to compute the efficiency of such a turbine (which would be nonsense anyway). We label this option as 'other'.

We plot the qualitative outcome of this experiment in Figures 2.3.1 and 2.3.2.

Remark. *In our experiment, we do not distinguish between e04ucf concluding that no feasible solution exists, and e04ucf getting stuck at a point which is not feasible.*

The first thing that we observe is that e04ucf converged to a KKT point only once. On all other occasions when the algorithm didn't encounter a nonsensical steam turbine

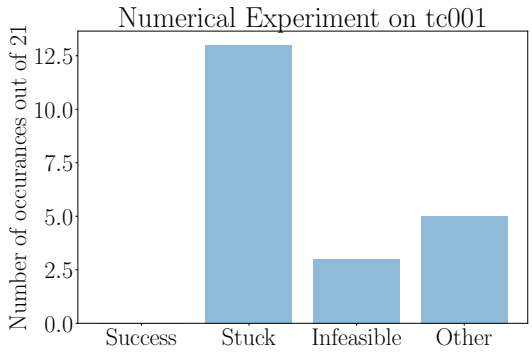

Figure 2.3.1 – The outcome of running e04ucf from 20 randomly generated starting positions, in addition to the given starting point for testcase tc001.

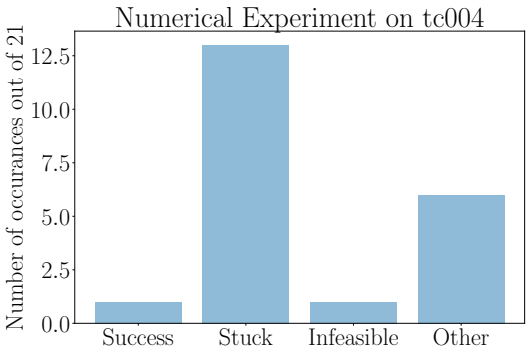

Figure 2.3.2 – The outcome of running e04ucf from 20 randomly generated starting positions, in addition to the given starting point for testcase tc004.

design, it either failed to find a feasible solution, or got stuck at a point which didn't satisfy the KKT conditions, yet from which no descent direction for the merit function could be found. This phenomenon should be treated with suspicion given that for any smooth function f, if the KKT conditions are not satisfied but the LICQ (see Definition 2.4) condition holds, then a descent direction usually exists. What then is going wrong?

When we consult the output files generated by e04ucf, one phenomenon observed is that the condition number of the QN matrix sometimes grows to very large values. While this might appear as a cause for concern given that the QN matrix's condition number remaining bounded was one of the conditions for Theorem 3, ultimately it is not a problem. From the perspective of convergence theory, while a bounded QN matrix is a common requirement to guarantee convergence, it is understood that QN methods often converge whether or not this condition is satisfied. (For another class of QN methods, it has been shown the condition number of $\boldsymbol{H}_k$ may grow linearly without forfeiting convergence guarantees [17].)

Regardless of this, Siemens have already found a practical solution to this problem. Whenever e04ucf stalls, we restart e04ucf from the place where the previous run finished. By doing so, we reset the QN matrix to the identity and let e04ucf continue. This course of action often works very well, so much so that when running e04ucf, we are in the habit

of never running it once (unless it finds a KKT point), but rather calling it repeatedly until it gets stuck at the same location from where it was started. The contents of Figures 2.3.1 and 2.3.2 are actually the results of running e04ucf repeatedly in this way.

In short, the QN matrix growing large is problematic but not the entire issue. To see why e04ucf fails, we must take a closer look at the points to which it converges.

2.3.2 Finding why e04ucf fails on 3dv1d

In order to see what causes e04ucf to fail, we attempt to visualise the shape of the objective function near $\tilde{\boldsymbol{x}}$, the point at which e04ucf converges to and gets stuck at. We systematically loop through all variables, and plot the shape of the function as each individual variable $\tilde{x}_i$ is varied within a small range. While most of these plots we generate are of little interest, we have displayed two of them in Figures 2.3.3 and 2.3.4.

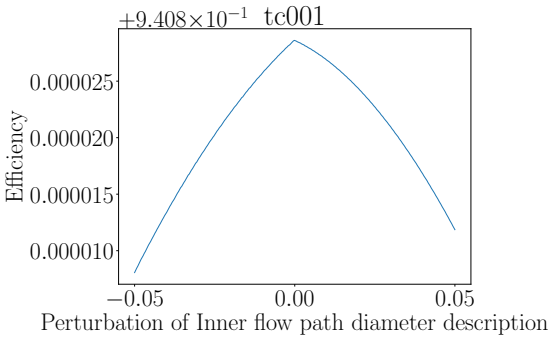

Figure 2.3.3 – The efficiency of the steam turbine corresponding to testcase tc001 plotted against a variable describing the inner flow path diameter description is varied near to where e04ucf stops.

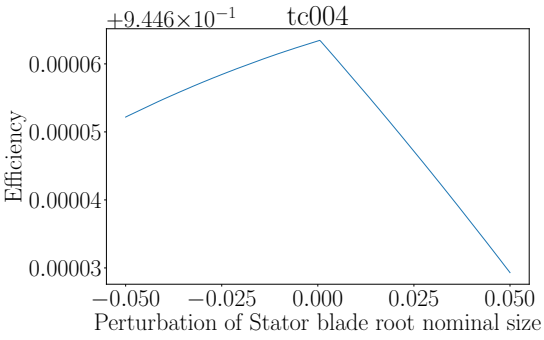

Figure 2.3.4 – The efficiency of the steam turbine corresponding to testcase tc004 plotted against a variable which describes the stator blade root nominal size is varied near to where e04ucf stops.

What we see in these figures is that the directional derivative of the objective function f is not continuously differentiable in that particular direction, thus implying that f is not continuously differentiable. We have generated these plots at many points for both testcases and have consistently found locations where the directional derivative is not continuously differentiable, both for the objective function f and constraint function $\boldsymbol{c}$. This provides a possible answer as to why e04ucf fails to converge to a KKT point,

given that all of the theory described in this chapter has been predicated on f and $\boldsymbol{c}$ being continuously differentiable.

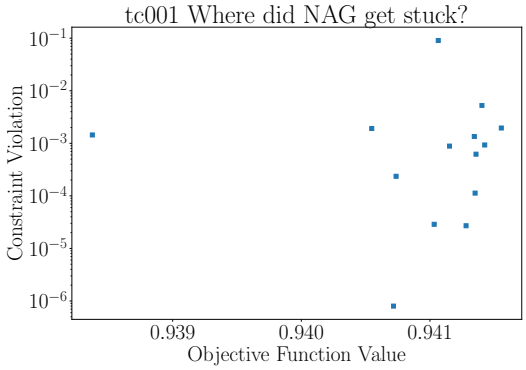

Figure 2.3.5 – For each run of e04ucf to tc001 which terminated with 'ifail'=6, we plot the objective function value against the total constraint violation evaluated at the final iterate.

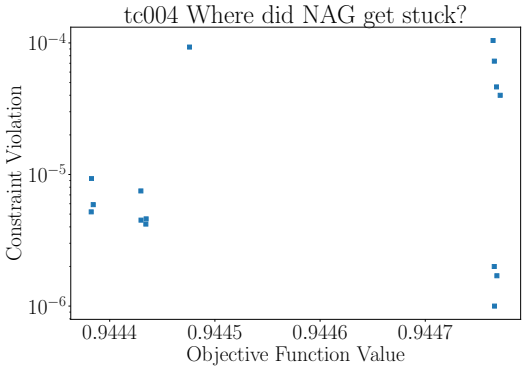

Figure 2.3.6 – For each run of e04ucf to tc004 which terminated with 'ifail'=6, we plot the objective function value against the total constraint violation evaluated at the final iterate.

Before dismissing e04ucf on qualitative grounds, we must first check how good the solutions it has found are quantitatively. Returning to Figures 2.3.1 and 2.3.2, we identify all the points which e04ucf converged to when it ultimately got stuck. We evaluate f and $\|\boldsymbol{c}\|$ at each of these points and plot the outcome in Figures 2.3.5 and 2.3.6 (where each point in the plot refers to a point at which e04ucf got stuck).

When consulting these plots, we see that for tc001, e04ucf was not very successful at finding a feasible solution. This is not the algorithms fault, given that it is only guaranteed to find a feasible solution in the end, but will explore the infeasible domain beforehand. The problem is that when e04ucf stalls prematurely, we cannot assume anything about the feasibility of point to which it had converged. While the total constraint violation for tc004 is much less than for tc001, we have the same qualitative problem (unless the feasibility tolerance is large enough to ignore these violations).

Meanwhile, we are not even in a position to say how close e04ucf has come to finding a KKT point. In our experience of applying e04ucf to 3dv1d, the final value of *Norm Gz* (see Section 2.2.1) is normally $\sim 10^{-2}$. However, we cannot read too much into this value, given that e04ucf is designed to terminate with error $ifail = 6$ the moment that it cannot find a descent direction despite *Norm Gz* not being less than ϵ. While is possible

that e04ucf has indeed converged to a local optimum, we would need an algorithm which is designed for non-smooth functions to inform us of this. Indeed, we do just this later in Section 3.4.3.

2.4 Conclusion

In this chapter, we have outlined the theory behind e04ucf, the algorithm which Siemens has used to optimize their steam turbine designs. We have shown that our problem lies outside the context in which this algorithm is guaranteed to succeed and have demonstrated experimentally that indeed e04ucf consistently terminates prematurely.

While in this thesis we proceed with the conclusion that 3dv1d is a non-smooth function, and its non-smoothness is what causes the problem for 3dv1d, we recognise that there are other possibilities. First and foremost, Siemens have been in the practice of implicitly assuming 3dv1d to measure the accuracy of a true steam turbine with very high accuracy. While we have continued with this approach, it is possible that 3dv1d should be thought of as an imprecise smooth function. If the function precision value (recall Section 2.2.1) is taken to be sufficiently large, then the kinks we observed in Section 2.3.2 become irrelevant. Therefore we recommend that regardless of the potential benefit of later chapters, Siemens should make a priority of determining how precise 3dv1d is.

However, proceeding as we do with the assumption that 3dv1d is precise, we will need a stronger theory, one that is suited to when f and $\boldsymbol{c}$ are not everywhere differentiable. This is why, having started this project by diagnosing the failure of applying e04ucf to 3dv1d, the direction of research which we chose to pursue was that of non-smooth optimisation.

Chapter 3

Applying Non-smooth Methods to 3dv1d

Having established in the previous chapter that the objective and constraint functions corresponding to 3dv1d are not everywhere differentiable, we now explore the theory of solving such problems; which are written in the form:

$$\begin{aligned} \underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad & f(\boldsymbol{x}) \\ \text{subject to} \quad & c_j(\boldsymbol{x}) \leq 0, \quad j \in \mathcal{I}, \end{aligned} \tag{3.1}$$

where $\mathcal{I}$ is the index set of inequality constraints, and the objective and constraint functions: f and $\boldsymbol{c}$ are Lipschitz continuous as opposed to continuously differentiable.

Remark. *Unlike in Chapter 2, here we only consider inequality constraints. We justify this as it simplifies the theory, and is sufficient for the problem of 3dv1d, which only has inequality constraints.*

In this chapter, we begin by reviewing existing work done on non-smooth optimisation (NSO). Next in Section 3.2, we list the main definitions and results which must be understood when transitioning to NSO from SO. In Section 3.3 we identify from our literature review two particular algorithms which seem most suitable for the problem of 3dv1d and explain them in detail. Finally, in Section 3.4 we apply the algorithm which

is immediately applicable for 3dv1d, and comment on its practicality.

3.1 Literature Review

Once the objective function f or constraint function $\boldsymbol{c}$ are not smooth, the theory presented in Chapter 2 breaks down. While an optimisation algorithm such as the one shown in Chapter 2 may perform well at times, it also may fail due to its inability to compute a gradient at a point where no gradient exists, fail to converge, or fail to ever pass an optimality test [18]. If we are to solve Problem 3.1, we require algorithms based on a more general theory. Fortunately, several of these exist.

The simplest generalisation of the gradient to non-smooth functions, the subgradient (we define this rigorously later in Definition 3.5), was discovered by Shor in 1962, and developed further by Shor, Ermoliev and Poljak (according to Poljak in his review of Soviet research [19]). Subgradients are a natural generalisation to gradients defined for convex functions. Upon subgradients, the class of algorithms called Subgradient Methods are built. These algorithms imitate Gradient Descent from SO, only replacing the gradient with a subgradient at each iteration.

There are two main limitations of Subgradient Methods. First, if $\boldsymbol{v}$ is a subgradient of f at $\boldsymbol{x}$, it does not follow that $-\boldsymbol{v}$ is a descent direction from $\boldsymbol{x}$. As a consequence of this, subgradient methods are not descent methods, meaning that $f(\boldsymbol{x}_k) \leq f(\boldsymbol{x}_{k-1})$ does not necessarily hold for all iterations k. Moreover, if $\boldsymbol{x}^*$ is a local minimum of f, it is not guaranteed that $\|\boldsymbol{v}_k\| \to 0$ as $\boldsymbol{x}_k \to \boldsymbol{x}^*$ (where $\boldsymbol{v}_k$ is a subgradient of f at $\boldsymbol{x}_k$ for all iterations k).

An alternative to Subgradient Methods are the Cutting Plane methods for convex optimisation. Where subgradient methods use a single subgradient with each iteration, and discard it afterwards, Cutting Plane methods keep track of all subgradients computed thus far, making continual use of them. The earliest examples of such methods are algorithms which were discovered separately by Cheney and Goldstein [20] and Kelley [21]. The main advantage of cutting plane methods over subgradient methods is that

cutting plane methods are able to identify when they have located a local minimiser.

The most commonly used extension of the subgradient to non-convex functions is the Clarke generalised gradient [22], which is well defined for any function which is Lipschitz continuous. Using this in place of gradient, Clarke generalised the KKT conditions [23], thus deriving optimality conditions for non-smooth non-convex Constrained Optimisation. By replacing subgradients with the Clarke generalised gradients, both subgradient methods and cutting plane methods can be generalised for non-convex functions.

Remark. *For the remainder of this thesis, when we use the term* subgradient*, what we are referring to is Clarke's generalised gradient.*

Bundle Methods are the extension of Cutting Plane methods which are applicable to non-convex functions. Discovered by Wolfe [24] and Lemaréchal (according to Schramm and Zowe [25]), these methods collect a 'bundle' or set of subgradients and function values, continually adding more until a descent direction is obtained. Other examples of such methods include [26–28]. An issue with this approach is encountered when the linear approximation to f using a particular subgradient does not underestimate f. (Note that this can't happen when f is convex). When such a subgradient is included in the bundle of gradients, it may obstruct the discovery of a descent direction.

An extension of the Bundle Method which deals with this problem are the Proximal Bundle Methods [29–33]. These methods make use of a 'bundle' of subgradients as before, but will discard a subgradient once the current iterate is sufficiently far from the point at which the subgradient was calculated. One proximal bundle method worth mentioning is the discrete gradient method of Bagirov et al[34], which does not require an analytic subgradient but rather approximates them numerically, a feature which is desirable with 3dv1d in mind, given that we lack any analytic subgradient.

Another variant of the Bundle approach which has received much attention recently is the gradient sampling Algorithm [35]. This method exploits the fact that a non-smooth function which is Lipschitz continuous will be differentiable almost everywhere, meaning that the gradient will be well defined for any randomly sampled point with probability one. Using this fact, the subdifferential (the set of all subgradients, which

we will defined rigorously in Section 3.2) may be approximated using a 'bundle' of gradients computed at randomly sampling points nearby. This has been used by Burke et al [36] to generalise the method of steepest descent, and has been developed further since [37–42].

A particular gradient sampling algorithm to take note of is the sequential quadratic programming with gradient sampling method of Curtis and Overton [5]. This algorithm generalises SQP and has been proven to have convergence guarantees for solving a general non-smooth non-convex constrained optimisation problem. Since NAG's SQP algorithm has consistently given Siemens sensible, even if not provably optimal solutions, we see SQP-GS as the natural next step to take. In this chapter, we provide a brief description for SQP-GS and demonstrate its effectiveness applied to 3dv1d

It is important to note that our choice to proceed with SQP-GS does not imply that there no other valid approaches which me might take. Another class of algorithms which we have not mentioned thus far are derivative free optimisation (DFO) methods. This class has two major subcategories [43]: Direct Search Methods, and Model-Based Methods.

The idea of Direct Search methods is to evaluate the objective (and constraint) function at a finite number of points which are called the *polling set*. Intuitively, the polling set may be thought of as a generalisation of the set of points which need to be evaluated in order to compute a finite difference gradient approximation. Based on the function values evaluated, a direct search method will then move to a new point and update the polling set, exactly how it does so depends on the specific method. One algorithm worth mentioning here is the mesh adaptive direct search (MADS) method [44], which has convergence guarantees for non-smooth, non-convex functions. Implementations of this method can be found in software packages such as NOMAD [45].

The other category of DFO methods: Model-Based Methods, involves using a set of points at which the objective has been evaluated to construct a model (usually quadratic) to approximate the objective function locally, and selecting the next point to evaluate by minimising this model within a trust region. While many of these meth-

ods such as NEWUOA [46, 47], BOBYQA [48], BC-DFO [49], and DFO-LS [50] are designed with SO in mind, there has been development in the non-smooth setting in recent years. For example, the algorithm of that of Liuzzi et al [51] combines ideas from DFO Model-Based Methods with Bundle methods to construct a non-smooth model in place of a smooth one.

Aside from the DFO methods listed previously, there also exists the 'nonderivative gradient sampling' method of Kiewel [52] which makes use of ideas from gradient sampling methods (see above) in the derivative free context, or the method of Riis et al [53] which makes use of discrete gradients which are not related to those used by Bagirov et al [34]. These are all methods which we might have considered using, and indeed are well suited to the problem of 3dv1d given the lack of availability of gradient information.

It is also worth pointing out here that while algorithms have been constructed to solve NSO problems, the cost of doing so is far greater than for smooth optimisation. So much so, that smooth optimisation algorithms, specifically BFGS, are often observed to perform competitively when applied to NS algorithms [54]. Ultimately, NSO algorithms can only be assured to converge in so general a setting because they have to do much extra work in the process. For a NSO algorithm to be competitively fast in practice, we must identify additional structure in the problem which is stronger than Lipschitz Continuous but weaker than Continuously Differentiable. For a non-exhaustive list of such function classes, see Lemaréchal's review of the theory of NSO[55]. If we can classify 3dv1d into any of these categories between Lipschitz continuous and differentiable, this may enable us to reduce the computational cost of a solution.

Ultimately, we chose to use the sequential quadratic programming with gradient sampling algorithm of Curtis and Overton [5] given its suitability to the problem of 3dv1d while being a natural extension from e04ucf, the method previously used. We overlooked SQPGS's need of gradient information given that the finite difference approximations of gradients (at smooth points) appeared to have been working well for Siemens in the past. Moreover, later in this thesis we decided to use the discrete gradient method of Bagirov et al [34], this time motivated by its deterministic approach

to constructing a bundle, along with its derivative free capabilities. For the remainder of this thesis, these are the only two algorithms which we discuss, despite many of the others mentioned above also being potentially usable for our problem.

Remark. *More recently, we have found additional solvers suitable to the full non-smooth non-convex Constrained Optimisation problem, including SolvOpt [56] and the Bundle sequential quadratic programming method of Schichl and Fendl [57]. However, we became aware of these too late for them to be included in our analysis.*

3.2 Theory of non-smooth optimisation

In order to understand the theory of NSO, we must define precisely what it means to be differentiable. For this, the first definition we need is that of Lipschitz Continuity:

Definition 3.1. *A function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be Lipschitz Continuous with Lipschitz constant L with respect to the Euclidean norm $\|\cdot\|$ if for every $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$:*

$$|f(\boldsymbol{x}) - f(\boldsymbol{y})| \leq L\|\boldsymbol{x} - \boldsymbol{y}\|.$$

In the following definitions and results (which we cite from various sources but don't prove), we justify the assertion that Lipschitz Continuous functions are differentiable almost everywhere.

Recall Definition 2.7 where we define the *directional derivative*. Using this, we can define the notion of differentiability in general. Here however, there are multiple definitions which differ from each other mainly in the context that they apply. We take our definition to be that of Gâteaux Differentiability.

Definition 3.2 (Gâteaux Differentiable [58])**.** *Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a real valued function defined on an open set in $\mathbb{R}^n$. The function f is said to be Gâteaux differentiable at a point $\boldsymbol{x}$ if there exists a bounded linear operator $T : \mathbb{R}^n \to \mathbb{R}^m$ such that:*

$$T\boldsymbol{d} = f'(\boldsymbol{x}; \boldsymbol{d}) \tag{3.2}$$

for every $\boldsymbol{d} \in \mathbb{R}^n$. *The (uniquely determined) operator* T *is called the Gâteaux derivative of* f *at* $\boldsymbol{x}$ *and is denoted by* $D_f(\boldsymbol{x})$.

Remark. *The definition of Gâteaux Differentiability given by Benyamini and Lindenstrauss [58] is more general that Definition 3.2. We have simplified the original definition to better match our context of use.*

Another definition of differentiability is Fréchet differentiability. Although we do not define it here, we note its significance due to the following result.

Theorem 4 (Rademacher's Theorem [13])**.** *Any Lipschitz function on* $\mathbb{R}^n$ *is almost everywhere Fréchet differentiable.*

All that remains in order to justify that a Lipschitz Continuous function is Gâteaux differentiable almost everywhere is to show the circumstances under which Fréchet differentiability implies Gâteaux differentiability.

Theorem 5 ([59])**.** *If the function* $f : \mathbb{R}^n \to \mathbb{R}^m$ *is Fréchet differentiable at* $\boldsymbol{x} \in \mathbb{R}^n$, *then* f *is Gâteaux differentiable at* $\boldsymbol{x}$.

Remark. *Theorem 5 is a simplified version of that given by Bell [59]. We have simplified the original result to better match our context of use.*

Between Theorems 4 and 5, we see that a Lipschitz continuous function $f : \mathbb{R}^n \to \mathbb{R}^m$ will be (Gâteaux) differentiable almost everywhere. It is critical to note that 'almost everywhere' does not mean everywhere. To see this, consider the following example.

Example 3.1. *Consider the function* $f(x) = x\sin(\log x)$ *on the set* $[0, 1]$. *We compute* $f'(x)$ *to find that:*

$$f'(x) = \sin(\log x) + \cos(\log x).$$

First we observe that the function $f'(x) \in (-2, 2),\ \forall x \in [0, 1]$. *The boundedness of the derivative implies that* f *is Lipschitz Continuous. However, the limit* $\lim_{x\to 0^+} f'(x)$ *does not exist, meaning* f *is Lipschitz Continuous at* 0 *even though the directional derivative at* 0 *does not exist.*

Remark. *For the remainder of this thesis, when we use the term* differentiable*, we are referring to the notion of* Gâteaux differentiable*.*

Definition 3.3 (Kink)**.** *Given f a Lipschitz Continuous function, we define a* kink *of f to be any subset of $\mathbb{R}^n$ on which f is not differentiable.*

Having established the definition of differentiable, we can now extend this notion to Lipschitz continuous functions. First we generalise the directional derivative.

Definition 3.4 (Clarke Generalised Derivative [60])**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a Lipschitz continuous function. Given a point $\boldsymbol{x}$ and a direction $\boldsymbol{d}$, the Clarke Generalised Derivative is defined as:*

$$f^\circ(\boldsymbol{x};\boldsymbol{d}) = \limsup_{\substack{\boldsymbol{y}\to\boldsymbol{x} \\ t\downarrow 0}} \frac{f(\boldsymbol{y}+t\boldsymbol{d}) - f(\boldsymbol{y})}{t}.$$

As a consequence of Lipschitz Continuity, f° is uniquely defined everywhere.

This definition enables that of the generalised gradient.

Definition 3.5 (Clarke Generalised Gradient [60])**.** *Given the Lipschitz continuous function $f : \mathbb{R}^n \to \mathbb{R}$, the Clarke Generalised Gradient of f at $\boldsymbol{x} \in X$ is defined by:*

$$\partial_C f(\boldsymbol{x}) = \{\boldsymbol{v} \in \mathbb{R}^n : \ f^\circ(\boldsymbol{x};\boldsymbol{d}) \geq \langle \boldsymbol{v} | \boldsymbol{d} \rangle \,, \ \forall \boldsymbol{d} \in \mathbb{R}^n\}.$$

Having defined the Clarke Generalised Gradient, we now list a few properties of this gradient taken from [60]. Before doing this however, we require the definition of the *closure* of a set.

Definition 3.6 (Closure [61])**.** *Let $U \subset \mathbb{R}^n$ be given. We define a* limit point *of U to be any point $\boldsymbol{y} \in \mathbb{R}^n$ such that there exists a sequence of points $\{\boldsymbol{x}_k\}_{k=1}^{\infty}$, with $\boldsymbol{x}_k \in U$, $\forall k$ such that $\boldsymbol{y} = \lim_{k\to\infty} \boldsymbol{x}_k$. Let U be the set of* limit points *of U. Then we define the closure of U to be $U \cup V$, and write it as $\overline{U}$.*

Proposition 3.1. *Given $f : \mathbb{R}^n \to \mathbb{R}$, a Lipschitz continuous function (with Lipschitz constant L), the Clarke Generalised Gradient has the following properties:*

1. *[60, Proposition 2.1.2]* $\partial_C f(\boldsymbol{x})$ *is a nonempty, convex, weak-compact subset of* $\mathbb{R}^n$ *and* $\|\boldsymbol{v}\| \leq L$ *for every* $\boldsymbol{v} \in \partial_C f(\boldsymbol{x})$.

2. *[60, Proposition 2.1.2] For every* $\boldsymbol{d} \in \mathbb{R}^n$*, it holds that:*

$$f^\circ(\boldsymbol{x}; \boldsymbol{d}) = \max\{\langle \boldsymbol{v} | \boldsymbol{d} \rangle \ : \ \boldsymbol{v} \in \partial_C f(\boldsymbol{x})\}.$$

3. *[60, Proposition 2.1.5] Let* $\boldsymbol{x}_i$ *and* $\boldsymbol{v}_i$ *be sequences such that* $\lim_{i\to\infty} \boldsymbol{x}_i = \boldsymbol{x}$*, and* $\boldsymbol{v}_i \in \partial_C f(\boldsymbol{x}_i)$ *for each* i*. Then* $\lim_{i\to\infty} \boldsymbol{v}_i \in \partial_C f(\boldsymbol{x})$.

4. *[60, Proposition 2.1.5] It holds that:*

$$\partial_C f(\boldsymbol{x}) = \bigcap_{\delta>0} \left(\bigcup_{\boldsymbol{y} \in \mathbb{B}_\delta(\boldsymbol{x})} \partial_C f(\boldsymbol{y}) \right).$$

5. *[60, Proposition 2.2.4] If* f *is Gâteaux differentiable at* $\boldsymbol{x}$*, then* $\partial_C f(\boldsymbol{x}) = \{D_f(\boldsymbol{x})\}$ *(see Definition 3.2). Conversely, if* $\partial_C f(\boldsymbol{x})$ *contains a single element* $\boldsymbol{v}$*, then* f *is Gâteaux differentiable and* $D_f(\boldsymbol{x}) = \boldsymbol{v}$.

6. *[60, Proposition 2.5,1] Let* $\mathcal{D}_f$ *the set on which* f *is Gâteaux differentiable. It holds that*

$$\partial_C f(\boldsymbol{x}) = \overline{Conv\,\{\lim \nabla f(\boldsymbol{y}) : \boldsymbol{y} \to \boldsymbol{x}, \boldsymbol{y} \in \mathcal{D}_f\}},$$

   where $Conv\,\{X\}$ *refers to the convex hull of the set* X.

Remark. *In many other sources [34, 42, 55, 62], the property listed in Item 6 is given as the definition of the Clarke Generalised Gradient. We have seen fit to use the definition from Clarke's textbook.*

When optimizing an unconstrained NS function, the 1st order optimality condition for the point $\boldsymbol{x}^*$ is simply that $\mathbf{0} \in \partial_C f(\boldsymbol{x}^*)$. However, we are concerned with Problem 3.1 which involves constraints, and hence require more elaborate 1st order optimality conditions. While Definition 2.3 still applies, Theorem 1 is no longer applicable for identifying a local minimiser. Thus, we need a generalised version of Theorem 1. Though

the definitions which follow come from Clarke [60], we use the notation of Curtis and Overton [5]. In Chapter 2, we needed to assume the LICQ conditions before we could demonstrate optimality. Now we instead require the property of *calmness*.

Definition 3.7 ([5]). *A local minimiser $\boldsymbol{x}^*$ of Problem 3.1 is called calm if $\exists K > 0$ such that for all sequences $(\boldsymbol{x}_k, \boldsymbol{t}_k) \to (\boldsymbol{x}^*, 0)$, with each pair $(\boldsymbol{x}_k, \boldsymbol{t}_k)$ satisfying $\boldsymbol{c}(\boldsymbol{x}_k) \leq \boldsymbol{t}_k$, we have $f(\boldsymbol{x}^*) - f(\boldsymbol{x}_k) \leq K\|\boldsymbol{t}_k\|$.*

To understand Definition 3.7 intuitively, we define the ℓ_1 penalty function $\phi_\rho(\boldsymbol{x})$ applied to Problem 3.1 by:

$$\phi_\rho(\boldsymbol{x}) := \rho f(\boldsymbol{x}) + v(\boldsymbol{x}), \tag{3.3}$$

where v is the *constraint violation function*:

$$v(\boldsymbol{x}) = \sum_{j \in \mathcal{I}} \max(-c_j(\boldsymbol{x}), 0). \tag{3.4}$$

A calm local minimiser $\boldsymbol{x}^*$ is a point for which there exists $\rho^* > 0$ such that $\boldsymbol{x}^*$ is also a local minimiser for the function $\phi_\rho(\boldsymbol{x})$ for all values of $\rho \in [0, \rho^*]$. In other words, calm local optimisers are the only ones we might expect to find when using a penalty method to account for constraints. Using the definition of calmness, we finally generalise the KKT conditions.

Theorem 6 ([5]). *Suppose f and $\boldsymbol{c}$ are Lipschitz continuous on $\mathbb{R}^n$, let $\boldsymbol{x}^*$ be a calm local minimizer of Problem 2.1. Then there exists dual variables $\boldsymbol{\lambda}^* \geq \boldsymbol{0}$ such that the primal-dual pair $(\boldsymbol{x}^*, \boldsymbol{\lambda}^*)$ satisfies:*

$$\partial_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\lambda}^*) \ni 0,$$
$$\lambda_j^* c_j(\boldsymbol{x}^*) = 0,\ j \in \mathcal{I},$$

where $\partial_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\lambda}^*)$ is the (partial) Clarke subdifferential [60, p. 48] with respect to the variable $\boldsymbol{x}$ of $\mathcal{L}(\cdot, \boldsymbol{\lambda}^*)$ at $\boldsymbol{x}^*$ and $\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \boldsymbol{\lambda}^T \boldsymbol{c}(\boldsymbol{x})$ is the Lagrangian.

Remark. *When Theorem 6 is applied to a smooth function, it simplifies to Theorem 1.*

With NS optimality conditions now defined, there is one last difficulty with NSO which we discuss in this section. A NS optimiser will attempt to generate a sequence of points $\{\boldsymbol{x}_k\}$ such that $\boldsymbol{x}_k \to \boldsymbol{x}^*$ where $\mathbf{0} \in \partial_C f(\boldsymbol{x}^*)$. Unlike smooth optimisation where if $\nabla f(\boldsymbol{x}^*) = \mathbf{0}$ and $\boldsymbol{x}_k \to \boldsymbol{x}^*$ then $\nabla f(\boldsymbol{x}_k) \to \mathbf{0}$, there is no similar result in NSO. Therefore, it is not enough for us to approximate $\partial_C f(\boldsymbol{x}_k)$ given that all the subgradients in $\partial_C f(\boldsymbol{x}_k)$ might all point in the same direction even though $\|\boldsymbol{x}_k - \boldsymbol{x}^*\| < \epsilon$.

Definition 3.8 (Goldenstein ϵ-subdifferential [63])**.** *Given a Lipschitz continuous function $f : \mathbb{R}^n \to \mathbb{R}$ and $\epsilon > 0$, we define the Goldstein ϵ-subdifferential to be*

$$\partial_\epsilon f(\boldsymbol{x}) = \overline{Conv\,\{\partial_C f(\boldsymbol{y}) \mid \boldsymbol{y} \in \mathbb{B}_\epsilon(\boldsymbol{x})\}}.$$

If $\mathbf{0} \in \partial_\epsilon f(\boldsymbol{x})$, then we call $\boldsymbol{x}$ an ϵ-stationary point of f.

There are two advantages of using the Goldstein subdifferential in practice. First of all, if $\boldsymbol{x}_k \to \boldsymbol{x}^*$ where $\mathbf{0} \in \partial_C f(\boldsymbol{x}^*)$, once $\|\boldsymbol{x}_k - \boldsymbol{x}^*\| \leq \epsilon$, then $\mathbf{0} \in \partial_\epsilon f(\boldsymbol{x}_k)$. The second advantage is that $\partial_\epsilon f(\boldsymbol{x})$ may be naturally approximated using a bundle of gradients which we now define:

Definition 3.9 (Bundle)**.** *Given the function f and $\boldsymbol{x} \in \mathbb{R}^n$, we define a* bundle *of points around $\boldsymbol{x}$ to be any finite set of points, each of which is ideally but not necessarily close to $\boldsymbol{x}$.*

Finally we define the approximate subdifferential, which requires a bundle in order to construct.

Definition 3.10 (Approximate Subdifferential)**.** *Given the function f, $\boldsymbol{x} \in \mathbb{R}^n$, and $\mathcal{B}$, a bundle (see Definition 3.9) of points close to $\boldsymbol{x}$, we define the* approximate subdifferential *to be:*

$$\tilde{\partial} f(\boldsymbol{x}; \mathcal{B}) = \overline{Conv\,\{\nabla f(\boldsymbol{y}) \mid \boldsymbol{y} \in \mathcal{B}\}}, \tag{3.5}$$

A key property of the Approximate Subdifferential is that if $\mathcal{B} \in \mathbb{B}_\epsilon(\boldsymbol{x})^m$ for some number m, then $\tilde{\partial} f(\boldsymbol{x}; \mathcal{B}) \subseteq \partial_\epsilon f(\boldsymbol{x})$. In short, we can approximate the Goldstein subdifferential with the Approximate Subdifferential using an appropriately chosen bundle

of points. The accuracy of the Approximate Subdifferential depends on a few factors including the number of points in the Bundle and their positioning. Both SQP-GS and DGM make use of a Bundle, but generate them in very different ways. In the section which follows, we describe these two algorithms in detail.

3.3 Two applicable algorithms

In Section 3.1 we identified the SQP-GS algorithm of Curtis and Overton [5] as an algorithm with a similar structure to 3dv1d that is equipped to solve Problem 3.1 with the sole assumption that f and $\boldsymbol{c}$ are Lipschitz continuous. However, this algorithm requires the ability to compute $\nabla f(\boldsymbol{x})$ at any point $\boldsymbol{x}$ where f is differentiable. In the context of 3dv1d, we lack this feature and must rely on numerical approximations for the gradients.

Motivated by this, we identified a second algorithm: the discrete gradient method of Bagirov et al. Like SQP-GS, the DGM is a bundle method. Unlike SQP-GS, this method is not currently equipped to handle constraints, and requires more assumptions to be made about the structure of f. However, it does not assume analytic gradients to be available either. Although we don't directly apply this method to 3dv1d, we outline this method after SQP-GS given that we will make use of it in Chapter 4.

Therefore, in this section we describe in detail how both of these methods operate, how each of them construct their bundle of points, and under what circumstances we might expect them to converge to a generalised KKT point.

3.3.1 Sequential quadratic programming with gradient sampling

Sequential quadratic programming with gradient sampling (SQP-GS) is an algorithm which solves Problem 3.1 by minimising the ℓ_1 penalty function $\phi_\rho(\boldsymbol{x})$ (see Equation (3.3)). Given a particular value of ρ, a minimiser of $\phi_\rho(\boldsymbol{x})$ may not be feasible. However, by iteratively reducing the magnitude of ρ and optimising ϕ_ρ, we can converge

to a calm local minimiser.

Similar to SQP, SQP-GS is a line search method, meaning that it matches the form of Algorithm 1. In order to explain how it operates, we must once again define how SQP-GS: 1. selects a search direction, 2. determines a step length, and 3. modifies its Quasi-Newton matrix. However, there are two new features of the algorithm to explain: how the bundle of points used to construct the approximate subdifferential is generated, and how the penalty parameter ρ is reduced in order to find a feasible solution.

Remark. *In Chapter 2, the Lagrangian function (see Equation* (2.7)*) was constructed in such a way that when ρ was small, infeasible points were not punished very hard, while if ρ was large, infeasible points would be penalised. In this section, we construct a merit function where ρ has precisely the opposite effect. Therefore in this chapter we talk about reducing ρ as we progress instead of increasing it.*

3.3.1.1 Generating the bundle of points

The strategy used by SQP-GS, and indeed GS algorithms in general to approximate $\partial_C(\phi_\rho)(\boldsymbol{x})$ is to compute $\tilde{\partial}(\phi_\rho)(\boldsymbol{x}_k;\mathcal{B}_k)$ using Equation (3.5) where $\mathcal{B}_k$ refers to a bundle of points near to $\boldsymbol{x}_k$.

As implied by the phrase gradient sampling, SQP-GS generates this bundle randomly. Given the sampling radius ϵ and number of samples $n_s \geq n+1$, SQP-GS generates n_s randomly sampled points which lie within the distance ϵ of $\boldsymbol{x}_k$. When ϵ is large, $\mathbf{0} \in \tilde{\partial}(\phi_\rho)(\boldsymbol{x}_k;\mathcal{B}_k)$ doesn't imply that $\boldsymbol{x}_k$ is a local minimiser, but rather that there exists a local minimiser of ϕ_ρ within a distance ϵ of $\boldsymbol{x}_k$.

It must be noted that for SQP-GS, we must generate n_s sample points for f and each $c_j,\ j \in \mathcal{I}$ independently, meaning $n_s(1+|\mathcal{I}|)$ random points in total. We denote the set of points randomly generated for f at the k-th iteration by $\mathcal{B}_k^f$ and the set of points for each c_j by $\mathcal{B}_k^{c_j}$.

3.3.1.2 Determining a descent direction

In order to find a descent direction from $\boldsymbol{x}_k$ for the function ϕ_ρ, we solve the following quadratic programming problem:

$$\begin{aligned} \boldsymbol{p}_k = \quad & \underset{(\boldsymbol{d}, z, \boldsymbol{r}) \in \mathbb{R}^{n+|\mathcal{I}|+1}}{\arg\min} \quad \rho z + \frac{1}{2}\boldsymbol{d}^T \boldsymbol{H}_k \boldsymbol{d} + \sum_{j \in \mathcal{I}} r_j \\ & \text{subject to} \quad f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{y})^T \boldsymbol{d} \leq z, \quad \boldsymbol{y} \in \mathcal{B}_k^f, \\ & \qquad\qquad c_j(\boldsymbol{x}_k) + \nabla c_j(\boldsymbol{y})^T \boldsymbol{d} \leq r_j, \quad \boldsymbol{y} \in \mathcal{B}_k^{c_j},\ j \in \mathcal{I},, \\ & \qquad\qquad\qquad\qquad r_j \geq 0, \quad j \in \mathcal{I}. \end{aligned} \tag{3.6}$$

Problem 3.6 is a natural extension of Problem 2.4 for when f and $\boldsymbol{c}$ are NS. Where in Problem 2.4 the objective depended on the gradients of f and $\boldsymbol{c}$, in Problem 3.6 we must consider the gradients evaluated at each point within our bundle $\mathcal{B}_k$.

Next, given $\boldsymbol{p}_k$, the solution obtained from solving Problem 3.6 we define:

$$q_\rho(\boldsymbol{d}; \boldsymbol{x}, \mathcal{B}_k, \boldsymbol{H}) := \rho \max_{\boldsymbol{y} \in \mathcal{B}_k^f} \{f(\boldsymbol{x}) + \langle \nabla f(\boldsymbol{y}) | \boldsymbol{d} \rangle\} + \sum_{j \in \mathcal{I}} \max_{\boldsymbol{y} \in \mathcal{B}_k^{c_j}} \{\max(c_j(\boldsymbol{x}) + \langle \nabla c_j(\boldsymbol{y}) | \boldsymbol{d} \rangle, 0)\} + \tfrac{1}{2}\boldsymbol{d}^T \boldsymbol{H} \boldsymbol{d}. \tag{3.7}$$

In practice, we never need to compute q_ρ directly, given that its value is obtained from any practical solver of Problem 3.6, as q_ρ would be the outcome if the problem was min instead of argmin. Using this, we now construct the function:

$$\begin{aligned} \Delta q_\rho(\boldsymbol{d}; \boldsymbol{x}, \mathcal{B}_k, \boldsymbol{H}) &:= q_\rho(\boldsymbol{0}; \boldsymbol{x}, \mathcal{B}_k, \boldsymbol{H}) - q_\rho(\boldsymbol{d}; \boldsymbol{x}, \mathcal{B}_k, \boldsymbol{H}), \\ &= \phi_\rho(\boldsymbol{x}) - q_\rho(\boldsymbol{d}; \boldsymbol{x}, \mathcal{B}_k, \boldsymbol{H}) \geq 0. \end{aligned} \tag{3.8}$$

Curtis and Overton [5] show that Δq_ρ is an upper bound for the directional derivative of ϕ_ρ in the direction of $\boldsymbol{d}$. As such, it is relevant for the next section where we discuss the line search procedure.

3.3.1.3 Determining a step length

Having solved Problem 3.6 to obtain the search direction $\boldsymbol{p}_k$, the next step is to determine the step length α_k before setting $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$. For each iteration, we require that if

$\|\boldsymbol{p}_k\| > 0$, then $\phi_\rho(\boldsymbol{x}_{k+1})$ must be sufficiently smaller than $\phi_\rho(\boldsymbol{x}_k)$. The condition which we enforce is:

$$\phi_\rho(\boldsymbol{x}_k + \alpha\boldsymbol{p}_k) \leq \phi_\rho(\boldsymbol{x}_k) - c_1\alpha\Delta q_\rho(\boldsymbol{p}_k; \boldsymbol{x}_k, \mathcal{B}_k, \boldsymbol{H}_k). \tag{3.9}$$

Since Δq_ρ gives and upper bound for the directional derivative of ϕ_ρ in the direction $\boldsymbol{p}_k$, Equation (3.9) is a weaker condition, but in the same form as Equation (2.6a).

To compute such α satisfying Equation (3.9), we apply Algorithm 3 to the function $\varphi(\alpha) = \phi_\rho(\boldsymbol{x}_k + \alpha\boldsymbol{p}_k)$.

Algorithm 3: SQP-GS line search algorithm

```
/* Initialisation:                                              */
input : φ // Objective function
        1 t ∈ (0,1) // Backtracking constant
        2 c_1 > 0 // Armijo constant
        3 φ'(0) // Directional derivative
4 α = 1;
5 while φ(α) > φ(0) − c_1 α φ'(0) do
6 |  α = tα;
output: α.
```

Remark. *Note that when applying Algorithm 3 to SQP-GS, we insert* $\Delta q_\rho(\boldsymbol{p}_k; \boldsymbol{x}_k, \mathcal{B}_{\epsilon,k}, \boldsymbol{H}_k)$ *instead of* $\phi'_\rho(\boldsymbol{x}_k; \boldsymbol{p}_k)$ *as the directional derivative.*

3.3.1.4 Updating the Quasi-Newton Matrix

As in SQP, with SQP-GS we use a QN matrix which approximates $\nabla^2\phi_\rho(\boldsymbol{x}_k)$. However, the QN matrix used by SQP-GS differs from that used by NAG's SQP in two ways.

Firstly, while we still use Equation (2.12), we can no longer define $\boldsymbol{s} = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$ and $\boldsymbol{y} = \nabla_{\boldsymbol{x}}\mathcal{L}(\boldsymbol{x}_{k+1}, \boldsymbol{\lambda}_{k+1}) - \nabla_{\boldsymbol{x}}\mathcal{L}(\boldsymbol{x}_k, \boldsymbol{\lambda}_k)$, given that $\mathcal{L}(\cdot)$ is no longer a smooth function. Nor can we simplistically replace $\nabla_{\boldsymbol{x}}\mathcal{L}(\boldsymbol{x}_k, \boldsymbol{\lambda}_k)$ and $\nabla_{\boldsymbol{x}}\mathcal{L}(\boldsymbol{x}_{k+1}, \boldsymbol{\lambda}_{j+1})$ with $-\boldsymbol{p}_k$ and $-\boldsymbol{p}_{k+1}$ given that $\boldsymbol{p}_k$ and $\boldsymbol{p}_{k+1}$ are computed using the values of $\boldsymbol{H}_k$ and $\boldsymbol{H}_{k+1}$ respectively.

Instead, we replace $\nabla_{\boldsymbol{x}}\mathcal{L}(\boldsymbol{x}_k, \boldsymbol{\lambda}_k)$ with the subgradient $\boldsymbol{v}_k$ which approximates the minimum norm element of $\tilde{\partial}\phi_\rho(\boldsymbol{x}_k; \mathcal{B}_k)$ (see Definition 3.10). In order to compute $\boldsymbol{v}_k$, we record the dual variables $(\lambda^f_{k,0}, \ldots, \lambda^f_{k,n_s})$, $(\lambda^{c_0}_{k,0}, \ldots, \lambda^{c_{|\mathcal{I}|}}_{k,n_s})$, $(\lambda^{c_0}_{k,0}, \ldots, \lambda^{c_{|\mathcal{I}|}}_{k,n_s})$ which are

obtained from solving Problem 3.6. Then we calculate:

$$\boldsymbol{v}_k = \sum_{i=1}^{n_s} \lambda_{k,i}^{f} \nabla f(\boldsymbol{x}_{k,i}^{f}) + \sum_{j \in \mathcal{I}} \sum_{i=1}^{n_s} \lambda_{k,i}^{c_j} \nabla f(\boldsymbol{x}_{k,i}^{c_j}), \tag{3.10}$$

where $\boldsymbol{x}_{k,i}^{f} \in \mathcal{B}_k^{f}$, and $\boldsymbol{x}_{k,i}^{c_j} \in \mathcal{B}_k^{c_j}$ are the points at which the gradient were sampled earlier. Finally we define $\boldsymbol{y}$ (see Equation (2.12)) to be $\boldsymbol{v}_k - \boldsymbol{v}_{k-1}$.

Remark. *Note that for SQP-GS the term $\boldsymbol{y} = \boldsymbol{v}_k - \boldsymbol{v}_{k-1}$ is delayed by an iteration, given that $\boldsymbol{v}_k$ and $\boldsymbol{v}_{k-1}$ correspond to $\boldsymbol{x}_k$ and $\boldsymbol{x}_{k-1}$ respectively, while in Section 2.1.2.3 $\boldsymbol{y}$ depended on the gradients of the Lagrangian at $\boldsymbol{x}_{k+1}$ and $\boldsymbol{x}_k$.*

The second difference in how the QN matrix is updated is that in SQP-GS a *limited memory* QN method is used. What this means is that $\boldsymbol{H}_k$ does not depend directly on $\boldsymbol{H}_{k-1}$. Instead of simply updating $\boldsymbol{H}_k$ each iteration, we store the last $k_{\boldsymbol{H}}$ values of $\boldsymbol{y}_k = \boldsymbol{v}_k - \boldsymbol{v}_{k-1}$ and $\boldsymbol{s}_k = \boldsymbol{x}_k - \boldsymbol{x}_{k-1}$. Then, after initialising $\boldsymbol{H}_k = \boldsymbol{I}$, we apply the following transformation

$$\boldsymbol{H}_k \leftarrow \boldsymbol{H}_k - \frac{\boldsymbol{H}_k \boldsymbol{s}_\ell \boldsymbol{s}_\ell^T \boldsymbol{H}_k^T}{\boldsymbol{s}_\ell^T \boldsymbol{H}_k \boldsymbol{s}_\ell} + \frac{\boldsymbol{y}_\ell \boldsymbol{y}_\ell^T}{\boldsymbol{y}_\ell^T \boldsymbol{s}_\ell} \tag{3.11}$$

exactly $k_{\boldsymbol{H}}$ times for $\ell = k, k-1, \ldots, k-k_{\boldsymbol{H}}+1$. Computing $\boldsymbol{H}_j$ in this way means that old information about the shape of f and $\boldsymbol{c}$ is eventually forgotten as its relevance decreases.

3.3.1.5 How SQP-GS modifies its parameters

SQP-GS locates a local optimum to Problem 3.1 by finding local minimisers for ϕ_ρ (see Equation (3.3)) for a sequence of values for ρ. If $\boldsymbol{x}^*$, a local minimiser for Problem 3.1, is 'calm' (see Definition 3.7), then once ρ is sufficiently small any $\boldsymbol{x}^*$ which minimises ϕ_ρ also solves Problem 3.1. In this section, we discuss how to generate this sequence of ρ values.

The penalty parameter ρ is not the only parameter being modified as SQP-GS runs. There are also the sampling radius ϵ, which is the maximum distance from $\boldsymbol{x}_k$ that the set of points $\mathcal{B}_k$ are sampled and the violation tolerance θ which is the total violation of the constraint function which SQP-GS is temporarily willing to tolerate without

reducing the penalty parameter.

Given values for ρ, θ, and ϵ, we search for a local minimiser to ϕ_ρ. This is accomplished by iteratively constructing the search direction $\boldsymbol{p}_k$, which depends on $\mathcal{B}_k$ and in turn ϵ. When ϵ is large, the approximate subgradient $\tilde{\partial}\phi_\rho(\boldsymbol{x}_k; \mathcal{B}_k)$ captures the behaviour of the function ϕ_ρ nearby, but isn't accurate close to $\boldsymbol{x}_k$. Therefore, we do not worry about minimising ϕ_ρ too precisely and stop once the following condition is met:

$$\Delta q_\rho(\boldsymbol{p}_k; \boldsymbol{x}_k, \mathcal{B}_k, \boldsymbol{H}_k) \leq \nu\epsilon^2, \tag{3.12}$$

where ν is a tolerance constant which is specified by the user.

Once Equation (3.12) applies, we conclude that we are approaching a local minimum of ϕ_ρ. The next step is to reduce ϵ so that we might get closer. However, as we do so we are also faced with two options. If $\boldsymbol{x}_k$, the point which minimised ϕ_ρ satisfies $v(\boldsymbol{x}_k) \leq \theta$ (see Equation (3.4)), then we conclude that $\boldsymbol{x}_k$ is tolerably feasible for now, but reduce θ in order to force $v(\boldsymbol{x}_k) \to 0$. Otherwise, we reduce the value ρ in order to give greater weight to the constraint function in future. By doing so, we likely change where the minimiser of ϕ_ρ is located, therefore justifying why we didn't minimise ϕ_ρ too precisely.

3.3.1.6 Convergence results for SQP-GS

Now that we have explained the different steps of SQP-GS, we write it in full algorithmic form in Algorithm 4. The rest of this section focuses on the main results from [5] which show the circumstances under which SQP-GS is guaranteed to converge.

Assumption 3.1. *The functions f and $\boldsymbol{c}$ are locally Lipschitz continuous (which is a weaker property than being Lipschitz continuous) and continuously differentiable on open dense subsets with full measure in $\mathbb{R}^n$.*

Assumption 3.2. *The sequences of iterates and sample points generated by Algorithm 4 are contained in a convex set over which the functions f and $\boldsymbol{c}$ and their first derivatives are bounded. In addition, there exists constants $\overline{\xi} > \underline{\xi} > 0$ such that $\underline{\xi}\|\boldsymbol{d}\|^2 \leq \boldsymbol{d}^T\boldsymbol{H}_k\boldsymbol{d} \leq \overline{\xi}\|\boldsymbol{d}\|^2$ for all k and $\boldsymbol{d} \in \mathbb{R}$.*

Algorithm 4: SQP with Gradient Sampling [5]

```
/* Initialisation:                                                    */
```
input : $\epsilon > 0$, $\rho > 0$, $\theta > 0$, $n_s > n+1$, $c_1 \in (0,1)$, $t \in (0,1)$, $\beta_\epsilon \in (0,1)$, $\beta_\rho \in (0,1)$, $\beta_\theta \in (0,1)$, $\nu > 0$, $\boldsymbol{x}_0 \in \Omega_f$.

Set $k = 0$;
```
/* Main loop                                                          */
```
while *Not Converged* **do**
  Generate $\mathcal{B}_k$
  Construct $\boldsymbol{H}_k$ (see Section 3.3.1.4)
  Compute the search direction $(\boldsymbol{p}_k, z_k, r_k)$ from Problem (3.6).
  if $\Delta q_\rho(\boldsymbol{p}_k; \boldsymbol{x}_k, \mathcal{B}_k, \boldsymbol{H}_k) \leq \nu\epsilon^2$ **then**
    if $v(\boldsymbol{x}_k) \leq \theta$ **then**
      $\theta = \beta_\theta \theta$;
    else
      $\rho = \beta_\rho \rho$;
    $\epsilon = \beta_\epsilon \epsilon$
    $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k$
    $\boldsymbol{p}_k = 0$
  else
    Set $\boldsymbol{p}_k = \max_{\ell \in \mathbb{N}} \{t^\ell : f(\boldsymbol{x}_k + t^\ell \frac{\boldsymbol{p}_k}{\|\boldsymbol{p}_k\|}) - f(\boldsymbol{x}_k) < -c_1 t^\ell \Delta q_\rho(t^\ell; \boldsymbol{x}_k, \mathcal{B}_k, \boldsymbol{H}_k)\}$
  $k = k + 1$;

The convergence of SQP-GS relies on Assumptions 3.1 and 3.2. Note that these assumptions are very similar to the informally stated requirements of SQP in Section 2.1.2.5.

Theorem 7 (Global Convergence of SQP-GS[5])**.** *Let $\{\boldsymbol{x}_k\}$ be the (infinite) sequence of iterates generated by Algorithm 4. If Assumptions 3.1 and 3.2 apply, then with probability one, one of the following holds true:*

1. *The penalty parameter satisfies $\rho = \rho^*$ for some $\rho^* > 0$ for all large k and every cluster point of $\{\boldsymbol{x}_k\}$ is stationary for the penalty function ϕ_{ρ^*}. Moreover, with K defined as the (infinite) subsequence of iterations during which ϵ is decreased, we have $\{v(\boldsymbol{x}_k)\}_{k \in K} \to 0$, meaning that all cluster points of $\{\boldsymbol{x}_k\}_{k \in K}$ are feasible for Problem (3.1) in addition to being stationary for ϕ_{ρ^*}.*

2. *The penalty parameters satisfies $\rho \to \rho^* = 0$ and every cluster point of $\{\boldsymbol{x}_k\}$ is stationary for the constraint violation function v.*

3.3.2 Discrete gradient method

The second existing algorithm we have worked with in this project is the DGM of Bagirov et al. The DGM differs from SQP-GS in several ways: it does not yet have an official implementation which is designed for constrained optimisation, it assumes that the objective function f has more structure than merely being locally Lipschitz continuous, and it is deterministic. We see fit to explain this algorithm here as we will make use of it in future chapters.

The DGM is designed for functions which are *semismooth* and *quasidifferentiable*. We do not define these terms in detail here (they may be found in [34]) because any function which is the maximum of finitely many smooth functions has these properties. In Chapter 4 we will discuss why we are willing to assume that 3dv1d has this structure.

Like SQP-GS, DGM is also a line search method of the form of Algorithm 1. Since it does not make use of a QN matrix, we can describe it entirely by stating how it determines the search direction $\boldsymbol{p}_k$ and step length α_k. This we do in Sections 3.3.2.2 and 3.3.2.3 before presenting convergence results in Section 3.3.2.4. However, we begin in Section 3.3.2.1 by defining the process by which the DGM approximates a subgradient.

3.3.2.1 Approximating a subgradient

Unlike SQP-GS, the DGM does not assume that we have access to analytic subgradients, but it is equipped with a procedure to approximate them. When approximating a subgradient, we do so given a particular direction. We begin by defining the unit sphere, or set of directions:

$$S_1 = \{\boldsymbol{d} \in \mathbb{R}^n \mid \|\boldsymbol{d}\| = 1\}. \tag{3.13}$$

Given the point $\boldsymbol{x} \in \mathbb{R}^n$ and the direction $\boldsymbol{d} \in S_1$, we wish to approximate a subgradient $\boldsymbol{v}$ with the property that $f'(\boldsymbol{x}; \boldsymbol{d}) = \langle \boldsymbol{v} | \boldsymbol{d} \rangle$ (which the DGM requires to be defined everywhere). To do so, we need two discrete step lengths. In the derivation of the DGM [34], one of these discrete step lengths is $h > 0$ while the other is $z(h)$, where $z \in P$ with

P defined by:

$$P = \{z \mid z(h) \in \mathbb{R},\ z(h) > 0,\ h > 0,\ h^{-1}z(h) \to 0,\ h \to 0\}. \tag{3.14}$$

While this notation is important for theoretical purposes, in practice h and $z(h)$ may be thought of as two finite difference step lengths where $0 < z(h) \ll h \ll 1$.

Definition 3.11 (Discrete Gradient [34]). *Let $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{d} \in S_1$, $h > 0$, $i \in \{1, 2, \ldots, n\}$ and $z \in P$ be given. We define $\Gamma^i(\boldsymbol{x}, \boldsymbol{d}, z, h)$ to be the discrete gradient of f in the direction $\boldsymbol{d}$, and compute it as follows:*

$$\begin{aligned} \Gamma^i_j(\boldsymbol{x}, \boldsymbol{d}, z, h) &= \frac{f(\boldsymbol{y}_j) - f(\boldsymbol{y}_{j-1})}{z(h)},\ j = 1, \ldots, n,\ j \neq i, \\ \Gamma^i_i(\boldsymbol{x}, \boldsymbol{d}, z, h) &= \frac{f(\boldsymbol{x} + h\boldsymbol{d}) - f(\boldsymbol{x}) - h \sum\limits_{j=1,\, j\neq i}^{n} \Gamma^i_j(\boldsymbol{x}, \boldsymbol{d}, z, h) d_j}{h d_i}, \end{aligned} \tag{3.15}$$

where

$$\boldsymbol{y}_0 = \boldsymbol{x} + h\boldsymbol{d},\ \text{and}\ \boldsymbol{y}_j = \boldsymbol{y}_{j-1} + z(h)\boldsymbol{e}_j,\ \forall j = 1, 2, \ldots, n. \tag{3.16}$$

Here e_i refers to the i-th Cartesian unit vector.

Remark. *Note that from the definition of the Discrete Gradient, it holds that: $f(\boldsymbol{x} + h\boldsymbol{d}) - f(\boldsymbol{x}) = h\left\langle \Gamma^i(\boldsymbol{x}, \boldsymbol{d}, z, h) | \boldsymbol{d} \right\rangle$, which means that $\left\langle \Gamma^i(\boldsymbol{x}, \boldsymbol{d}, zh) | \boldsymbol{d} \right\rangle$ is a natural approximation for $f'(\boldsymbol{x}; \boldsymbol{d})$.*

The following proposition assures us that the discrete gradient will be bounded, a property which is important when proving convergence of DGM.

Proposition 3.2 ([34]). *Let f be a Lipschitz continuous function defined on $\mathbb{R}^n$ and let $L > 0$ be its Lipschitz constant. Then for any $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{d} \in S_1$, $z \in P$ and $h > 0$, it holds that:*

$$\left\| \Gamma^i(\boldsymbol{x}, \boldsymbol{d}, z, h) \right\| \leq C(n)L, \quad C(n) = \sqrt{n^2 + 2\sqrt{n^3} - 2\sqrt{n}}. \tag{3.17}$$

With discrete gradients now defined, the next step is to construct a descent direction.

3.3.2.2 Constructing a descent direction

Similar to SQP-GS, DGM computes a search direction by computing an approximate subdifferential $\tilde{\partial} f(\boldsymbol{x}_k; \mathcal{B}_k)$ at the point $\boldsymbol{x}_k$ using Equation (3.5). The main difference between DGM and SQP-GS lies in how the bundle $\mathcal{B}_k$ is constructed.

Where SQP-GS randomly samples n_s points for the bundle at each iteration, DGM adds points to the bundle through a deterministic process. Moreover for DGM, since we are not dealing with constraints, we only need to store the gradients in the bundle and not the points themselves.

For each iteration k, we initialise the bundle to contain the single subgradient: $\mathcal{B}_0 = \{\Gamma^f(\boldsymbol{x}_k, \boldsymbol{p}_{k-1}, z, h)\}$. Next we test to see if $\tilde{\boldsymbol{p}_k} = -\Gamma^f(\boldsymbol{x}_k, \boldsymbol{p}_{k-1}, z, h)$ is a descent direction by checking if the following holds:

$$f\left(\boldsymbol{x}_k + h\frac{\tilde{\boldsymbol{p}_k}}{\|\tilde{\boldsymbol{p}_k}\|}\right) - f(\boldsymbol{x}_k) \leq -c_2 h \|\tilde{\boldsymbol{p}_k}\|. \tag{3.18}$$

If Equation (3.18) holds, then we accept $\tilde{\boldsymbol{p}_k}$ as a valid direction of descent and we're done. Otherwise, we conclude that our bundle needs more subgradients, and compute $\Gamma^f(\boldsymbol{x}_k, \tilde{\boldsymbol{p}_k}, z, h)$ to add to the bundle.

From now on, in order to compute a trial direction, we solve the following quadratic programming problem using the subgradients stored in the bundle.

$$\begin{aligned} \tilde{\boldsymbol{p}_k} = \quad & \underset{(\boldsymbol{d}, z) \in \mathbb{R}^{n+1}}{\arg\min} \quad z + \frac{1}{2}\|\boldsymbol{d}\|^2 \\ & \text{subject to} \quad \boldsymbol{v}^T\boldsymbol{d} \leq z, \quad \boldsymbol{v} \in \mathcal{B}_k \end{aligned} \tag{3.19}$$

Note that Problem 3.19 precisely matches Problem 3.6 if the QN matrix is replaced with the identity matrix and constraints are removed. This Quadratic Programming problem has $n+1$ variables and 1 constraint for each subgradient in the bundle $\mathcal{B}_k$.

When the number of subgradients in the bundle is much less then n, we solve the

dual of Problem 3.19 instead:

$$\begin{aligned} \underset{\boldsymbol{\lambda} \in \mathbb{R}^{|\mathcal{B}_k|}}{\text{minimize}} \quad & \left\| \sum_{j=1}^{|\mathcal{B}_k|} \lambda_j \boldsymbol{v}_j \right\|^2 \\ \text{subject to} \quad & \sum_{j=1}^{|\mathcal{B}_k|} \lambda_j = 1, \quad , \\ & \lambda_j \geq 0, \quad j = 1, \ldots, |\mathcal{B}_k| \end{aligned} \tag{3.20}$$

Problem 3.20 only has $|\mathcal{B}_k|$ variables and $|\mathcal{B}_k| + 1$ constraints. Usually this will be a much smaller number of variables then for Problem 3.20 meaning it will be cheaper to solve. Problem 3.20 precisely describes the problem of finding the minimum normed element of a convex set.

We state this procedure more rigorously in Algorithm 5.

Algorithm 5: DGM computation of descent directions [34]

input : $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{d}_0 \in S_1$, $h \ll 1$, $c_2 \in (0,1)$, $\nu \ll 1$.

Set $i = \text{argmax}\{|d_j|,\ j = 1, \ldots, n\}$.

Compute $\boldsymbol{v}_0 = \Gamma^i(\boldsymbol{x}, \boldsymbol{d}_0, z, h)$.

Initialise $\mathcal{B}_0$, the bundle of gradients to be $\{\boldsymbol{v}_0\}$.

Compute $\boldsymbol{p}_0 = -\text{argmin}\{\|\boldsymbol{v}\| \mid \boldsymbol{v} \in \overline{Conv\,\{\mathcal{B}_0\}}\}$. $k = 0$.

while $\|\boldsymbol{p}_k\| > \nu$ *and* $f(\boldsymbol{x} + h\frac{\boldsymbol{p}_k}{\|\boldsymbol{p}_k\|}) - f(\boldsymbol{x}) > -c_2 h\|\boldsymbol{p}_k\|$ **do**

  Set $i = \text{argmax}\{|p_j|,\ j = 1, \ldots, n\}$.

  $\boldsymbol{v}_{k+1} = \Gamma^i(\boldsymbol{x}, \boldsymbol{p}_k, z, h)$.

  $\mathcal{B}_{k+1} = \mathcal{B}_k \cup \{\boldsymbol{v}_{k+1}\}$. (Add $\boldsymbol{v}_{k+1}$ to the bundle of gradients.)

  Compute $\boldsymbol{p}_{k+1} = -\text{argmin}\{\|\boldsymbol{v}\| \mid \boldsymbol{v} \in \overline{Conv\,\{\mathcal{B}_{k+1}\}}\}$.

  $k = k + 1$.

output: $\boldsymbol{p}_k$

Remark. *There are two circumstances under which Algorithm 5 terminates. The first is when a direction* $\boldsymbol{d}$ *satisfies* $f(\boldsymbol{x} + h\frac{\boldsymbol{d}}{\|\boldsymbol{d}\|}) - f(\boldsymbol{x}) > -c_2 h\|\boldsymbol{d}\|$ *which implies that* $\boldsymbol{d}$ *is a descent direction. Otherwise the algorithm terminates when* $\|\boldsymbol{d}\| < \nu$*, which is an indication that* $\boldsymbol{0} \in \partial_C f(\boldsymbol{y})$ *for some* $\boldsymbol{y}$ *near to* $\boldsymbol{x}$*.*

In order for Algorithm 5 to be practical, we need it to produce a descent direction within a finite number of iterations, a property which is proven in the following result.

Proposition 3.3 (Convergence of Algorithm 5 [34]). *Let f be a Lipschitz function defined on $\mathbb{R}^n$ (with Lipschitz constant L). Then, for $\nu \in (0, \overline{C})$, Algorithm 5 terminates after finite number of steps m, where*

$$m \leq 2\left(\frac{\log_2(\nu/\overline{C})}{\log_2(r)} + 1\right), \quad r = 1 - \left(\frac{\nu(1-c_1)}{2\overline{C}}\right)^2,$$

where $\overline{C} = C(n)L$ and $C(n)$ is the constant from Proposition 3.2.

3.3.2.3 Line search for DGM

In Algorithm 6 Line 6, we state that the step length at the k-th iteration is selected such that $\alpha_k = \text{argmax}\{\alpha \geq 0 \mid f(\boldsymbol{x}_k + \alpha\frac{\boldsymbol{p}_k}{\|\boldsymbol{p}_k\|}) - f(\boldsymbol{x}_k) \leq -c_1\alpha\|\boldsymbol{p}_k\|\}$. In practice though, we compute α_k by applying Backtracking line search (Algorithm 3).

3.3.2.4 Convergence of DGM

Having described how search directions and step lengths are selected, we now present the DGM in Algorithm 6. In the remainder of this section, we write out the conditions which must be satisfied to assure convergence.

Algorithm 6: discrete gradient method [34]

input : $\boldsymbol{x}_0 \in \mathbb{R}^n$, $h \ll 1$, $z \in P$, $0 < c_1 < c_2 < 1, \nu \ll 1$.

1 Randomly sample $\boldsymbol{d} \in S_1$.
2 Compute $\boldsymbol{p}_0$ by applying Algorithm 5 with inputs $(\boldsymbol{x}_0, \boldsymbol{d}, h)$.
3 $k = 0$.
4 **while** *True* **do**
5 **if** $\|\boldsymbol{p}_k\| > \nu$ **then**
6 $\alpha_k = \text{argmax}\{\alpha \geq 0 \mid f(\boldsymbol{x}_k + \alpha\frac{\boldsymbol{p}_k}{\|\boldsymbol{p}_k\|}) - f(\boldsymbol{x}_k) \leq -c_1\alpha\|\boldsymbol{p}_k\|\}$.
7 $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k$.
8 Compute $\boldsymbol{p}_{k+1}$ by applying Algorithm 5 with inputs $(\boldsymbol{x}_{k+1}, \frac{\boldsymbol{p}_k}{\|\boldsymbol{p}_k\|}, h, c_2, \nu)$.
9 **else**
10 $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k$.
11 $k = k + 1$.

output: $\boldsymbol{x}_k$

Before we state the convergence result, we must first make the following assumption.

Intuitively, the following statement assumes that each discrete gradient computed in Algorithm 5 corresponds to a real subgradient sufficiently close to $\boldsymbol{x}_k$.

Assumption 3.3. *Let $\boldsymbol{x} \in \mathbb{R}^n$ be a given point. For any $\epsilon > 0$ there exists $\delta > 0$ and $h_0 > 0$ such that $D_0(\boldsymbol{y}, h) \subset \partial_\epsilon f(\boldsymbol{x}) + S_\epsilon$ for all $\boldsymbol{y} \in S_\delta(\boldsymbol{x})$ and $h \in (0, h_0)$, where:*

$$D_0(\boldsymbol{x}, h) = \overline{Conv\,\{\boldsymbol{v} \in \mathbb{R}^n \mid \exists (\boldsymbol{d} \in S_1, z \in P\,) : \boldsymbol{v} = \Gamma^i(\boldsymbol{x}, \boldsymbol{d}, z, h)\}},$$

and

$$S_\epsilon = \{\boldsymbol{y} \in \mathbb{R}^n \mid \|\boldsymbol{x} - \boldsymbol{y}\| < \epsilon\}.$$

Theorem 8 (DGM [Convergence [34]). *If the function f is the maximum of a finite number of smooth functions, Assumption 3.3 is satisfied on $\mathbb{R}^n$ and the set $\{\boldsymbol{x} \in \mathbb{R}^n \mid f(\boldsymbol{x}) \leq f(\boldsymbol{x}_0)\}$ is bounded for any $\boldsymbol{x}_0 \in \mathbb{R}^n$, then every accumulation point of $\{\boldsymbol{x}_k\}$ belongs to the set $\{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{0} \in \partial_C f(\boldsymbol{x})\}$.*

Remark. *We have simplified the original result [34, Theorem 7.1] into Theorem 8 in order to reduce amount of notation and definitions required.*

3.4 Applying SQP-GS to 3dv1d

In Section 3.3.1.1, we stated that $\mathcal{B}_k^f$ and $\mathcal{B}_k^{c_j}$, $j \in \mathcal{I}$ must be sampled independently of each other. This means that $n_s(|\mathcal{I}| + 1)$ distinct sample points are required and not merely n_s. In addition, if $\boldsymbol{x}_k \in \mathbb{R}^n$, then we need $n_s \geq n + 1$ gradient samples for each function.

If the functions f and c_j are known and can be computed separately, then having $n_s(|\mathcal{I}| + 1)$ sample points as opposed to n_s does not effect the number of function evaluations as we will be computing each of f and c_j exactly $n_s + 1$ times anyway. However, when (as is the case for 3dv1d) we only may access f and $\boldsymbol{c}$ through an oracle which evaluates f and $\boldsymbol{c}$ at the same time, this increases the number of gradient computations we need by a factor of $|\mathcal{I}|$.

Since in our experience, this is the main bottleneck encountered when applying SQP-GS to 3dv1d, SQP-GS ends up being prohibitively expensive. In order to apply SQP-GS to 3dv1d in reasonable time, we are forced to relax this condition, thus using the same n_s sample points for f and each c_j. By doing so, we forfeit the convergence guaranteed in Theorem 7. Furthermore, the condition $n_s \geq n+1$ is also unreasonable due to the cost required. With convergence assurance already forfeited, we also relax this condition.

Finally it should be noted that Algorithm 4 lacks any termination condition. Ultimately, we would like $\Delta q_\rho(\alpha_k; \boldsymbol{x}_k, \mathcal{B}_k, \boldsymbol{H}_k) \to 0$, thus implying that we can't find a search direction. However, this outcome is meaningless unless $\epsilon \to 0$ given the dependence of the accuracy of $\tilde{\partial} f(\boldsymbol{x}_k; \mathcal{B}_k)$ on ϵ. Therefore, we introduce the minimal sampling radius ϵ_{min} and terminate Algorithm 4 once $\epsilon < \epsilon_{min}$.

With these modifications and using finite differences to approximate gradients, SQP-GS is fully equipped for the problem of 3dv1d. In Section 3.4.1 we describe how we set up a numerical experiment to test the effectiveness of SQP-GS on 3dv1d before presenting the results in Section 3.4.2.

Remark. *One approach which might be taken to reduce the cost of gradient calculation is to use* automatic differentiation *[11, Section 8.2]. Here we are limited by the source of our objective function: 3dv1d. Since 3dv1d is a black box function, we can't know if automatic differentiation is applicable to it. In fact, our understanding is that some subroutines within 3dv1d involve iteratively solving sub-problems to within a tolerance, a feature which automatic differentiation is not designed for. Moreover, we do not have access to the full source code for 3dv1d. For these reasons, we did not consider using automatic differentiation for this project.*

3.4.1 Setting up the numerical experiment

In order to test SQP-GS on 3dv1d, we start SQP-GS from the provided starting point as well as from 20 randomly generated starting points. For each starting point, we run SQP-GS exactly 4 times, corresponding to 4 different values for the initial penalty parameter. Given that SQP-GS is probabilistic, we note that for full rigour we should

Name	Value	Name	Value
ϵ	0.1	β_ϵ	0.9
θ	0.1	β_θ	0.4
c_1	10^{-8}	t	0.6
β_ρ	0.6	$k_{\boldsymbol{H}}$	10
ν	10^2	n_s	5
ϵ_{min}	5×10^{-4}	h	5×10^{-5}

Table 3.4.1 – Table of SQP-GS parameters and the values used.

be running SQP-GS multiple times from each starting point and for each initial penalty parameter. However, for reasons discussed in Section 1.4.4 we have only conducted numerical experiments which require at most a week to run. Since each run of SQP-GS on the testcases tc001 or tc004 in practice takes roughly 2 hours, including multiple runs from each starting point would exceed this bound.

We list out all the parameters needed for SQP-GS in Table 3.4.1. Of these, we point out that we have taken $n_s = 5$, a number far lower than the values of n (98 for tc001 and 72 for tc004). Given that we have violated the condition that $n_s \geq n + 1$, we have found that taking $n_s = 5$ offers the best combination of speed and reliability. In addition, we ensure that the discrete step length h used to approximate the gradient with finite differences is notably smaller than the sampling radius ϵ at all times, lest a pair of sampling gradients be computed from an overlapping cloud of points.

Finally, we have omitted the starting value for ρ from Table 3.4.1 given that we run SQP-GS multiple times with different values. Specifically, we run SQP-GS 4 times with the initial penalty parameter taken to be 1, 10, 100 and 1000. In practice values less than 1 have qualitatively the same outcome in that they converge quickly to a feasible solution and stop soon after, while values over 1000 tend to drive SQP-GS towards infeasible regions where nonsensical steam turbine designs are encountered which force 3dv1d to close down (see Section 1.3).

3.4.2 Outcome of the numerical experiment

In order to discuss the results of the numerical experiment, as in Section 2.3.1 we first must understand the different qualitative outcomes encountered to running SQP-GS to 3dv1d.

The two basic outcomes which we see are when SQP-GS converges to a feasible or infeasible local minimum. However, like SQP, SQP-GS may also navigate to a nonsensical design, meaning this is another possibility (which we indicate in Figures 3.4.1 and 3.4.2 by 'other'). Finally, due to the SQP-GS convergence conditions which we have violated, SQP-GS will on occasion get stuck. This manifests itself similarly to when SQP gets stuck, meaning that SQP-GS generates the search direction $\boldsymbol{p}_k$, finds no suitable step length, and repeats this process indefinitely.

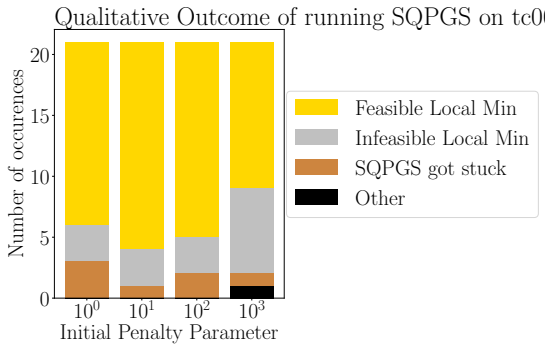

Figure 3.4.1 – The plot above shows the qualitative outcome of every SQP-GS applied to tc001 run given different starting positions and initial penalty parameters.

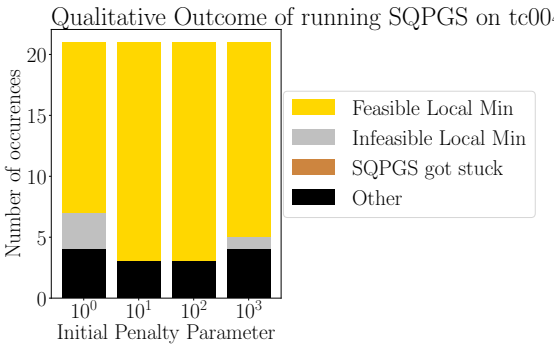

Figure 3.4.2 – The plot above shows the qualitative outcome of every SQP-GS applied to tc004 run given different starting positions and initial penalty parameters.

In Figures 3.4.1 and 3.4.2 we show how many times (out of 21) each of these outcomes occurs when applying SQP-GS to tc001 and tc004. From these plots, it seems that taking the initial penalty parameter to be 1 is too small, while 1000 is to large.

Taking Figures 3.4.3 and 3.4.4 into account, we see that when the starting penalty parameter is too small, then SQP-GS converges to a feasible point very quickly, but doesn't find as good a local optima with respect to the objective function. On the other hand, when the initial penalty parameter is too large, SQP-GS will neglect the constraints initially, navigate to a region with better objective function values, and

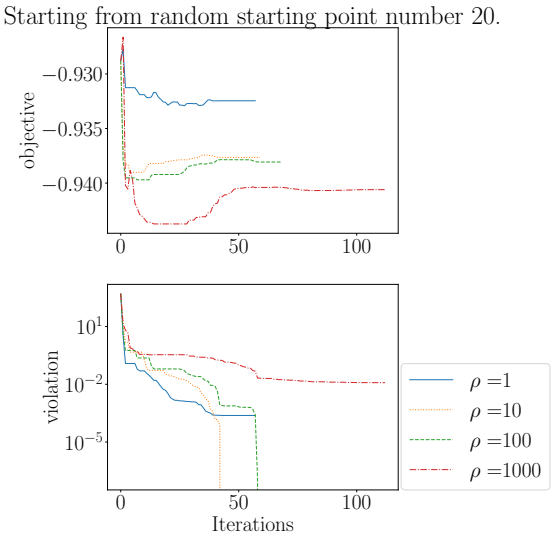

Figure 3.4.3 – Progress made by SQP-GS on tc001 per iteration.

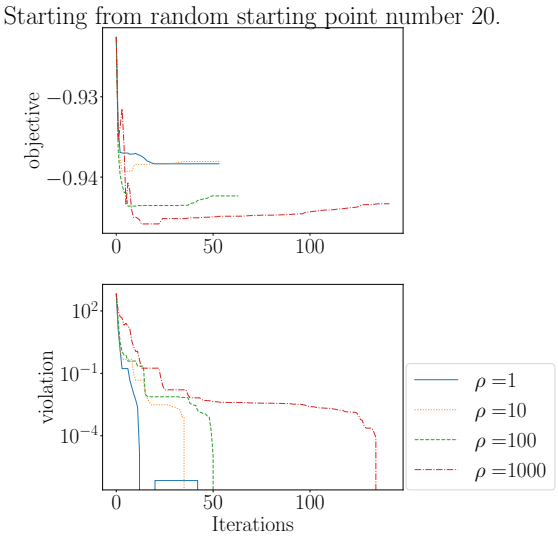

Figure 3.4.4 – Progress made by SQP-GS on tc004 per iteration.

then struggle to find a feasible solution when ρ is ultimately reduced. From these experiments we have run on tc001 and tc004, it certainly seems like taking the initial penalty parameter to be 10 or 100 is the better option.

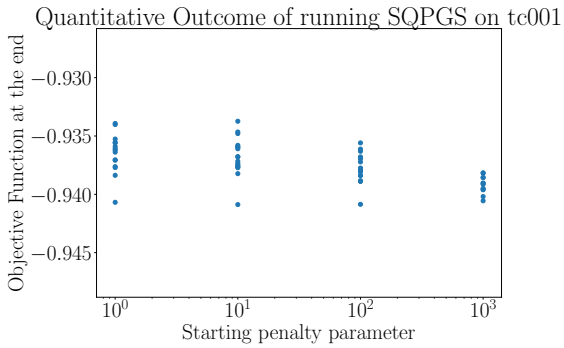

Figure 3.4.5 – For each starting point on tc001, we plot the final objective function value vs the initial penalty parameter which was used to find that point.

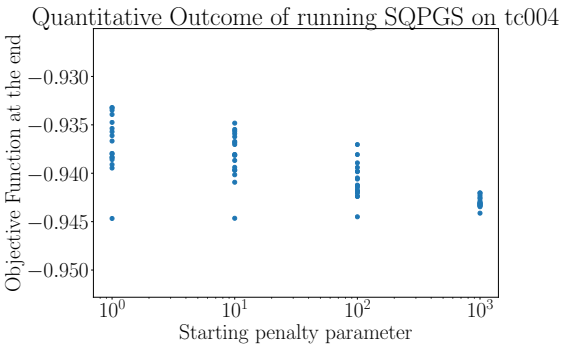

Figure 3.4.6 – For each starting point on tc004, we plot the final objective function value vs the initial penalty parameter which was used to find that point.

This observation was based on the qualitative outcome of the experiment, not the quantitative outcomes. Taking the initial penalty parameter to be large has poor results with respect to how often SQP-GS converges. However, Figures 3.4.3 and 3.4.4 display the performance of SQP-GS when starting at a point from which the largest value of the initial penalty parameter resulted in the best solution in the end. Motivated by this,

we generate Figures 3.4.5 and 3.4.6 in which we see the effects of the initial penalty parameter on the final objective function value.

There are two main features of these plots which stand out. First of all, we see that there is a particular starting point for which the initial penalty parameter makes no difference, and which consistently produces the best result. Although the plots do not show this information, this particular point corresponds to the Siemens restart file, that is the starting point selected manually as a good starting position. If we ignore these points, then taking the initial penalty parameter to be 1000 yields the best results. It must be noted however that running SQP-GS consistently requires far more iterations to converge as we see in Figures 3.4.3 and 3.4.4.

3.4.3 A practical heuristic for 3dv1d

In Section 2.3, we conducted a numerical experiment which showed that e04ucf, the algorithm currently used by Siemens to solve 3dv1d usually terminates with an error. While this does not mean that e04ucf has failed to find a suitable steam turbine design, it does mean that the outcome of running e04ucf cannot be accepted as suitable without scrutiny afterwords. Now in Section 3.4.2, we have shown that when we apply SQP-GS to 3dv1d, we face the trade off of cost and reliability vs the quality of the solution.

The natural progression from this point is to ask: might it be practical to apply SQP-GS starting from the point where e04ucf stalls? In Section 3.4, we concluded that in order for SQP-GS to computationally affordable (but still expensive), we needed to relax the conditions which required that the number of gradients sampled at each iteration exceeded the number of variables, and that these points be sampled separately for each constraint function. However, in our experience of running SQP-GS to 3dv1d, we have observed that these relaxations do not seem to be a problem when SQP-GS operates within (or at least close to) the feasible set. This implies that whenever e04ucf stalls at a point which is feasible, or at least nearly so, then SQP-GS may be applied to either make further progress, or confirm that the e04ucf did indeed converge to a local minimum.

There are two ways in which e04ucf gets stuck at an infeasible point. The first corresponds to when 'ifail'=3 (see Section 2.2.2), and refers to when e04ucf can't find any feasible point. In these cases, running SQP-GS with these relaxations is not effective. However when 'ifail'=6, this merely means that e04ucf can't find a direction of descent. Whether or not $\boldsymbol{x}_k$ is feasible or not is irrelevant.

In Section 2.3 we ran e04ucf from 20 different starting positions and presented the qualitative outcome. Of these 20 points, we examine those at which e04ucf stalled due to the error of $ifail = 6$ (recall Section 2.2.2). We now run SQP-GS from each of these points (of which there are roughly 13 for each testcase) in order to see how far from a local optimum these points actually were. When we do this, we take the values of $\rho_0 = 1$ but $\theta_0 = 0$ (recall Section 3.4.1). This choice of parameters restricts SQP-GS to prioritise finding a feasible solution quickly.

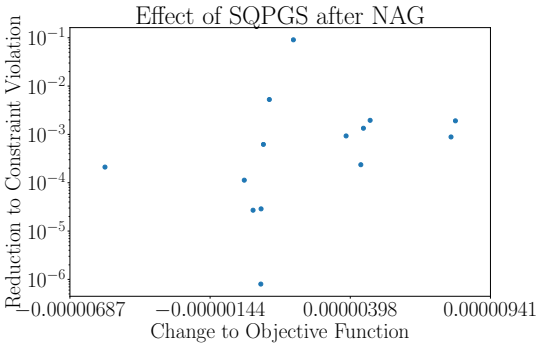

Figure 3.4.7 – For each run of e04ucf to tc001 which terminated with 'ifail'=6, we plot the change made by SQP-GS to both the objective function, and constraint violation.

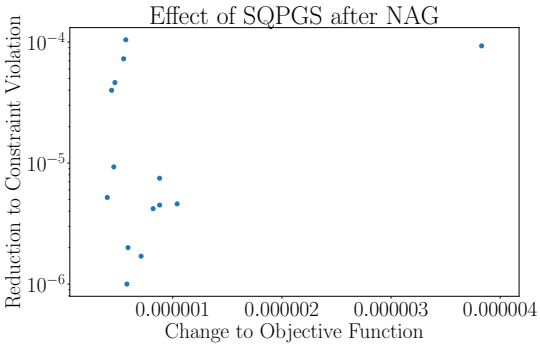

Figure 3.4.8 – For each run of e04ucf to tc004 which terminated with 'ifail'=6, we plot the change made by SQP-GS to both the objective function, and constraint violation.

In Figures 3.4.7 and 3.4.8, we plot the effect of running SQP-GS after e04ucf, by plotting the difference in objective function value and constraint violation between the starting point of SQP-GS and the stopping point. This enables us to finally see how close e04ucf came to finding locally optimal solutions. We see on looking at these figures that SQP-GS has not made a significant difference to the value of f, thus implying that e04ucf did nearly all of this. However, SQP-GS was impactful in reducing the constraint violation to 0. Although this detail is not clearly shown, for both tc001 and tc004, SQP-

GS managed to reduce the total constraint violation to 0, making negligible changes to the objective function value in the process.

Remark. *One may note that in Figure 2.3.5, one of the points has a constraint violation* $\sim 10^{-1}$*. However, despite our declaring that SQP-GS was able to complete the process of finding a feasible solution, there is no point in Figure 3.4.7 where the constraint violation has been modified by this much. This is because the point from Figure 2.3.5 corresponds to 'ifail'=3, and thus was neglected later.*

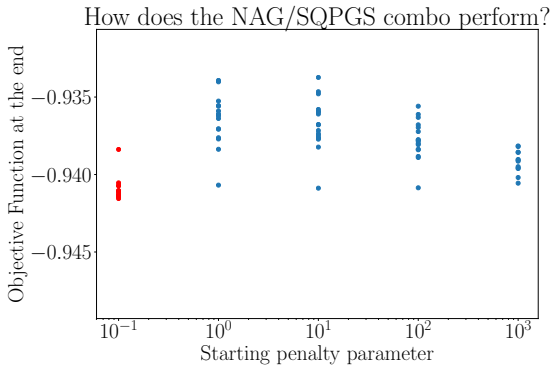

Figure 3.4.9 – We copy Figure 3.4.5 while also inserting a new set of results above 'Starting penalty parameter'=0.1. This column refers to the e04ucf/SQP-GS combo, not to a particular run of SQP-GS. We show the quality of the optimiser found by SQP-GS given a initial penalty parameter vs that of SQP-GS when run after e04ucf.

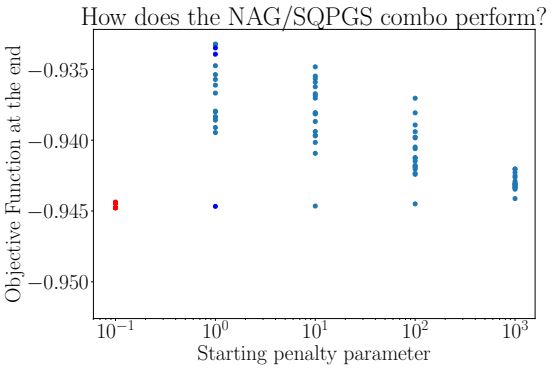

Figure 3.4.10 – We copy Figure 3.4.5 while also inserting a new set of results above 'Starting penalty parameter'=0.1. This column refers to the e04ucf/SQP-GS combo, not to a particular run of SQP-GS. We show the quality of the optimiser found by SQP-GS given a initial penalty parameter vs that of SQP-GS when run after e04ucf.

Having established the quantitative effect which running SQP-GS after e04ucf stalls has, we now compare this combination of algorithms against running SQP-GS on its own. We consider five contending algorithms: SQP-GS with $\rho_0 = 0$ after e04ucf, and SQP-GS with $\rho_0 \in \{1, 10, 100, 1000\}$. For each algorithm on each of 20 randomly generated starting positions (we use the same 20 for all algorithms), we record the value of the objective function evaluated the point it converged to (all these points were feasible). In Figures 3.4.9 and 3.4.10 we plot these results, where the values corresponding to combining SQP-GS after e04ucf are plotted on the left and in red.

These plots, combined with Figures 3.4.7 and 3.4.8 indicate that e04ucf is far more effective than SQP-GS at getting close to the best known solution. This comes at the cost of inconsistently finding feasible solutions. By combining SQP-GS with e04ucf however, we eliminate this weakness. Moreover, this combination of algorithms consistently yields better solutions than SQP-GS, as is particularly clear in Figure 3.4.10.

3.5 Conclusions and future work

In this chapter, we have identified SQP-GS as an algorithm which has been designed for a context which matches 3dv1d. In order to run SQP-GS practically, we have violated some conditions from the convergence theory for how many gradient samples we use. As a result of this, we sometimes observe SQP-GS to fail to converge on account of this modification. However, it still succeeds much of the time, leading to a few conclusions.

In our experience running SQP-GS on 3dv1d, we have never observed SQP-GS getting stuck at a feasible point. Within the interior of the feasible set, the kinks of the constraint function do not affect our ability to find a descent direction. This means that we only need enough gradient samples to capture the information related to the objective function's kinks. The fact that SQP-GS does not get stuck (for these testcases) within this set implies that 5 sample points is adequate to capture all required features of f. Moreover this implies that the objective function f, while non-smooth is not an unpleasantly structured non-smooth function.

In Section 3.4.3, we demonstrated using SQP-GS that even though e04ucf was failing to converge to locally optimal, or even feasible solution, it was successful in getting close to it. Starting from the points where e04ucf has failed, SQP-GS both faster and more reliably than SQP-GS on its own, while often finding better solutions. We view this approach to be the most practical short term option for Siemens.

However, we still question whether we might find or construct a method which can like e04ucf cheaply navigate towards a solution when it is easy to find a descent direction, and like acrshortsqpgs identify a feasible solution when it finds one. Since far less kinks

are to be found in the feasible domain, we reason that we'd like a potential solution to be adaptive in how much work it puts in to finding a descent direction, thus being cheap when the task is simple and thorough when more work is needed. This is why we have identified the DGM of Bagirov et al as a starting point for further research, as it does not construct its bundle randomly, but in a fashion is designed to find a descent direction with as small a bundle as possible.

While we have chosen to proceed in this way, we recognize that it is merely one of several valid options which might be taken from here. Some alternative directions of research which we might have taken, and which might be investigated in the future include:

1. Running a larger experiment on 3dv1d with the aim of discovering further structure.

2. Investigating other algorithms from the long list of possible options discussed in Section 3.1, both from the original starting point or following after e04ucf as we did for SQP-GS.

Chapter 4

Applying exact line search to non-smooth optimisation

Line search methods (LSMs) are the class of optimisation algorithms which may be written in the form of Algorithm 1. Every LSM consists of two main steps: determining the descent direction $\boldsymbol{p}_k$, and a suitable step length α_k. Previously in this thesis, we have discussed a particular smooth LSM (SQP) in Chapter 2, and two existing non-smooth LSMs (SQP-GS and DGM) in Chapter 3.

While we applied SQP-GS to 3dv1d in Chapter 3, we have not yet used DGM. There are two reasons why we have not yet done so: DGM has not yet (to our knowledge) been generalised to constrained optimisation and DGM is only applicable for functions which have more structure than merely being locally Lipschitz continuous. To avoid a bloat of definitions and notation, we did not precisely define the minimum requirement for DGM, rather noting that it is *sufficient* for the function f to be piecewise smooth.

Definition 4.1 (piecewise smooth Functions)**.** *The function $f : \mathbb{R}^n \to \mathbb{R}$ is called* piecewise smooth (PS) *if there exist N twice continuously differentiable functions f_j, $j \in \{1, 2, \ldots, N\}$ and open sets Ω_j such that $\bigcup_{i=1}^{N} \overline{\Omega_i} = \mathbb{R}^n$, ∇f_j is Lipschitz continuous with constant L_j, $f(\boldsymbol{x}) = f_j(\boldsymbol{x})$, $\forall \boldsymbol{x} \in \overline{\Omega_j}$, $\forall j \in 1, 2, \ldots, N$ and $\Omega_i \cap \Omega_j = \varnothing$, $\forall i, j \in \{1, \ldots, N\}$.*

While we don't know the theoretical properties of 3dv1d given its black-box nature, we can make an educated guess. For this purpose, we have conducted a large number of

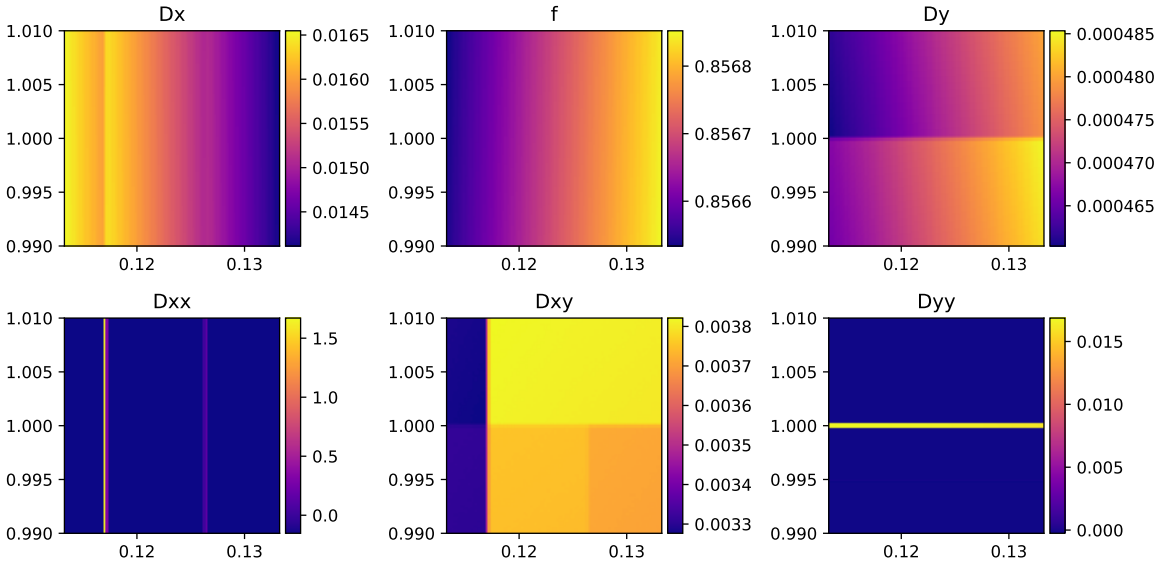

Figure 4.0.1 – Starting from the value of $\boldsymbol{x}$ where e04ucf stalls, we modify $x^{(1)}$ (x axis) and $x^{(12)}$ (y axis) and plot the value of f, as well as the numerical approximations to the first and second derivatives.

2D scans of 3dv1d, in which we vary specific pairs of variables, and plot the outcomes. While insufficient to grasp the full complexity of the function, this is the most we can do in order to visualise, and hence categorise 3dv1d. Figure 4.0.1 is just one of the many such plots we have generated. For more plots, and details behind their generation, consult Appendix A.1. While kinks are not clearly visible in the plot of f, they are easy to see in the plots of the first and second derivatives. What stands out is that the kinks appear finite in number, and nicely shaped. Motivated by this, as well as the advantages of doing so, we choose to hypothesize that the following assumption applies to 3dv1d:

Assumption 4.1. *The objective function f and constraint functions c_j, $j \in \mathcal{I}$ are all PS function with the property that $f(\boldsymbol{x}) = \max_i(f_i(\boldsymbol{x}))$ where f_i are defined as in Definition 4.1.*

Remark. *By assuming that f's non-smoothness comes from being a max-function, we have implicitly assumed it to have the property of* subdifferential regularity *[55]. This means that for every $\boldsymbol{x}, \boldsymbol{d} \in \mathbb{R}^n$, it holds that: $f'(\boldsymbol{x};\boldsymbol{d}) = f^\circ(\boldsymbol{x};\boldsymbol{d})$ (recall Definition 3.4). Without this property, we might easily fail to identify a descent direction. For example,*

the function $g(x) = -|x|$ *lacks this property and indeed we see that* $g^\circ(0;1) = g^\circ(0;-1) = 1$ *despite the fact that both* 1 *and* -1 *are directions of descent from* 0*.*

Remark. *While we assume 3dv1d to be of the form of a max function, it is important to note that we can only evaluate* f*, not* f_i *for each* i*. This is a critical point given that while there exist algorithms better equipped to solve max functions, these require us to be able to evaluate* f_i *for each* i*.*

By assuming f and $\boldsymbol{c}$ to be PS, we overcome the second of two problems with applying DGM to 3dv1d. In order to resolve the first problem, we note that if f and $\boldsymbol{c}$ are PS, then ϕ_ρ (see Equation (3.3)) is also PS. Since a constrained optimisation problem may be solved by iteratively optimising the penalty function, in this Chapter we neglect the constraints entirely noting that if we can optimise a PS function reliably, this will still be applicable to solving Problem 3.1. Therefore, in this chapter we use the function ϕ in order to make clear that it is not the same as f from Chapter 2.

We begin this chapter in Section 4.1 by investigating the LS method used by DGM to construct the step length, and proposing a new LS algorithm designed for NSO. Next in Section 4.2, motivated by the intention of constructing a new line search method designed to use our LS algorithm, we state key results from the theory of Generalised Gradient Descent in order to make it clear what properties a new algorithm must possess for convergence to be assured. Finally in Section 4.3, we present the outcome of our efforts to construct this new LSM and comment on its applicability.

4.1 Line search for piecewise smooth functions

When comparing the NSO algorithms from Chapter 3 to SQP from Chapter 2, we see that while the methods used to find the descent direction $\boldsymbol{p}_k$ differ greatly, the construction of the step length α_k is very similar. In this section, we examine the calculation of the step length more closely and question whether the algorithms used to compute α_k for smooth optimisations algorithms are the best choice for NSO.

We begin in Section 4.1.1 by arguing that the theoretical principles behind how the step length α_k is calculated do not apply to NSO. We make this point by use of an example which demonstrates how a smooth LS process may be very inefficient in the NS context.

Motivated by this, in Section 4.1.3 we present a new line search procedure of our own design. This algorithm is designed to combine the desirable features of both inexact line search and exact line search (recall Definition 2.6), using whichever is most cost effective at the time. In Section 4.1.3 we equip DGM with our new LS procedure: the hybrid line search and compare it against DGM with its default ILS method.

4.1.1 Is backtracking line search suitable for NSO?

When determining the step length α_k for a smooth optimiser of the form of Algorithm 1, the most common approach is compute α_k using inexact line search. This means that we do not rigorously optimise the 1D function $\varphi(\alpha) = \phi(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k)$, but rather find a value which merely ensures that the Wolfe Conditions (see Definition 2.8) are satisfied. Of these conditions, Equation (2.6a) forces $\phi(\boldsymbol{x}_{k+1}) < \phi(\boldsymbol{x}_k)$ for all iterations k, while Equation (2.6b) ensures that α_k is sufficiently large to guarantee that $\|\nabla\phi(\boldsymbol{x}_k)\|$ does not go to 0 until $\boldsymbol{x}_k \to \boldsymbol{x}^*$, where $\boldsymbol{x}^*$ is a stationary point of ϕ. Together, the Wolfe Conditions also have the desirable effect of ensuring that a Quasi-Newton positive definite matrix which is updated using Equation (2.12) will remain positive definite.

In practice Equation (2.6b) may be neglected for methods of the form of Algorithm 1 which do not make use of a QN matrix, provided that the α_k is constructed in a manner which ensures that it is not too small. One LS algorithm which accomplishes this is backtracking line search (BLS) (Algorithm 7), which is the method used by both SQP-GS and DGM.

This method is designed to be the minimal modification to the iteration process: $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \nabla\phi(\boldsymbol{x}_k)$ which assures convergence. While this algorithm can be applied within NS optimisers (demonstrated by SQP-GS and DGM), we question whether this is a natural step to take. This is because in NSO, $\operatorname{argmin}_{\boldsymbol{v}\in\partial_C\phi(\boldsymbol{x})} \|\boldsymbol{v}\|$ is not a con-

Algorithm 7: backtracking line search

```
/* Initialisation:                                          */
input : f // Objective function
        1 t // Backtracking constant
        2 c_1 > 0 // Armijo constant
3 α = 1;
4 while φ(x + αp) > φ(x) + c_1 α_k φ'(x;p). do
5 |  α = tα;
output: α.
```

tinuous function of $\boldsymbol{x}$, meaning that as $\boldsymbol{x}_k \to \boldsymbol{x}^*$, it does not necessarily hold that $\operatorname{argmin}_{\boldsymbol{v} \in \partial_C \phi(\boldsymbol{x}_k)} \|\boldsymbol{v}\| \to 0$. Because of this, there is no reason to expect $\|\boldsymbol{p}_k\| \to 0$ as $k \to \infty$. Therefore if convergence is to occur, we need $\alpha_k \to 0$, which in turn requires increasing costs from Algorithm 7 when ϕ is not differentiable at $\boldsymbol{x}^*$. We illustrate this idea in Example 4.1.

Example 4.1 ([64])**.** *Consider running DGM on the function* $\phi(x, y) = 10|x| + |y|$ *starting from* $\boldsymbol{x}_0 = (2, 2)$*. We generate the sequence* $\{\boldsymbol{x}_k\}$*, which in the context of this example, we write as* (x_k, y_k)*. By plotting* $\boldsymbol{x}_k$*, we observe a trajectory which is illustrated in Figures 4.1.1 to 4.1.3 (Note that we split the trajectory into 3 separate plots in order to visualise all relevant length scales).*

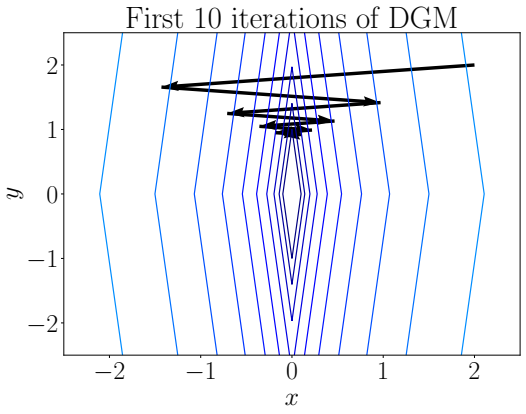

Figure 4.1.1 – We plot above the progress made by DGM on tc001 per iteration.

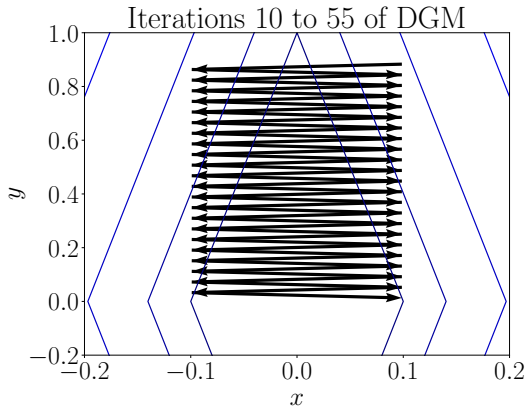

Figure 4.1.2 – We plot above the progress made by DGM on tc004 per iteration.

During the early iterations which are visualised in Figure 4.1.2, the algorithm behaves

exactly as we'd expect it to given its usage of BLS. However, in Figure 4.1.3 we see a sequence of iterations where DGM performs very badly despite the BLS operating as intended. Although this sequence of iterations eventually ends and we ultimately converge to the local minimum, in Figure 4.1.4 we see that the cost of the LS increases steadily as we get closer to the local minimum.

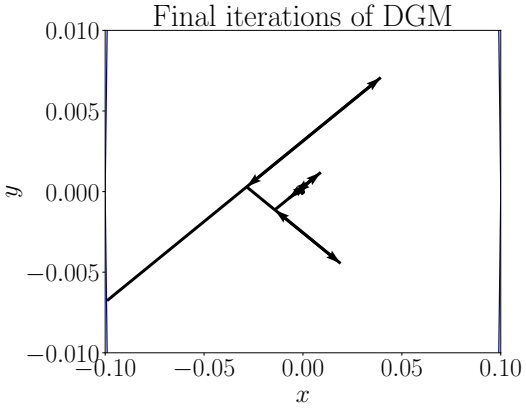

Figure 4.1.3 – We plot above the progress made by DGM on tc001 per iteration.

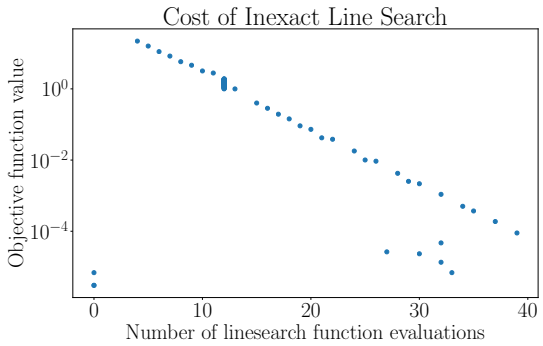

Figure 4.1.4 – We plot above the progress made by DGM on tc004 per iteration.

Why does this happen? Suppose that $x_k > 0$ *and* $y_k > 0$*, meaning that* $\phi(\boldsymbol{x}_k) = 10x_k + y_k$*. Here the search direction computed will be* $-\nabla\phi(\boldsymbol{x}_k) = -(10, 1)$*. Therefore,* α *will be a suitable step length if:*

$$\phi(\boldsymbol{x}_k - \alpha(10,1)) < \phi(\boldsymbol{x}_k) - c_1\alpha\phi'(\boldsymbol{x}_k; (10,1)),$$
$$\Leftrightarrow 10|x_k - 10\alpha| + |y_k - \alpha| < 10x_k + y_k - c_1\alpha\|(10,1)\|.$$

If we assume that $\boldsymbol{x}_k$ *is on of the iterates plotted in Figure 4.1.1 or Figure 4.1.2, then* $y_k > 0$ *and* $y_{k+1} > 0$*, while* $x_k x_{k+1} < 0$*. These assumptions imply:*

$$\phi(\boldsymbol{x}_k - \alpha(10,1)) < \phi(\boldsymbol{x}_k) - c_1\alpha\phi'(\boldsymbol{x}_k; (10,1)),$$
$$\Leftrightarrow 100\alpha - 10x_k + y_k - \alpha < 10x_k + y_k - c_1\alpha\|(10,1)\|,$$
$$\Leftrightarrow \alpha(99 - c_1\sqrt{101}) < 20x_k \Leftrightarrow \alpha < \frac{20x_k}{99 - c_1\sqrt{101}}.$$

The key conclusion which this calculation reveals is that as $x_k \to 0$, then the upper bound of suitable step length α also go to 0. As this happens, it takes more and more iterations of Algorithm 7 to construct a suitable step length. Ultimately, this is the phenomenon which Figure 4.1.4 illustrates.

While most of the points plotted in Figure 4.1.4 fit a particular line, there is a small cluster of points which do not match the pattern at all. These points correspond to when $\boldsymbol{x}_k$ has converged to one of the two kinks. Once this happens, the search direction $\boldsymbol{p}_k$ generated by DGM which approximates the minimal subgradient of $\partial_\epsilon \phi(\boldsymbol{x}_k)$ will be affected by the gradient of ϕ on both sides of the kink, and not only one.

There are a few points which Example 4.1 illustrates. First of all, we see that BLS is not a cheap algorithm when applied to NSO as it requires up to 40 function evaluations, a cost which is comparable to ELS. This is caused by the magnitude of the search direction being far larger than needed, which in this case occurs in part because of the non-smoothness of the example problem. Note that this issue is not unique to NSO as it also may occur for SO problems where there is ill conditioning.

Moreover we see that when a descent direction is nearly perpendicular to a nearby kink, then convergence occurs very slowly. During the sequence of iterations plotted in Figure 4.1.2, not only is little quantitative progress made from iteration to iteration, but no qualitative progress is made either. Motivated by this observation, we consider whether exact line search may have an application here, given that it would select a step length which would result in rapid convergence to a kink from where a better step direction might be constructed later.

In this chapter, we have chosen to research the possible merits of exact line search for non-smooth optimisation motivated by the phenomenon observed in Example 4.1. We recognize that this motivating phenomenon might be unique to low dimensions, and might have unrelated causes. The reason why we are influenced by this example is due to how it qualitatively resembles some of the low dimensional projections of the shape of 3dv1d in the neighbourhoods where our smooth optimiser has stalled.

4.1.2 The hybrid line search

Due to the poor performance of BLS demonstrated in Example 4.1, we now reconsider exact line search (ELS) (recall Definition 2.6) as an alternative despite the fact that it has been dismissed in the context of smooth optimisation for being too expensive. We justify this because the cost of BLS observed in Example 4.1 was comparable to ELS, meaning that ELS might not be prohibitively expensive in a NS context in light of the alternatives.

However, we recognise two issues with ELS which must be overcome if we are to apply it. First, as previously stated, ELS is needlessly expensive when applied for smooth functions. The second problem is that for ELS, we need an upper bound $\overline{\alpha}$ for the step length α such that there exists a local minimum of $\varphi(\alpha) = \phi(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k)$ within $[0, \overline{\alpha}]$. In order to apply ELS practically, we need to be able to overcome these two issues.

We begin by considering the second problem, how to obtain a bound $\overline{\alpha}$ such that we are guaranteed for there to exist a local minimiser to the function $\varphi(\alpha) = \phi(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k)$ within the interval $(0, \overline{\alpha})$. To solve this problem, we invoke a fundamental result from the theory of univariate optimisation. Specifically, if we have three points a, b, c such that $a < b < c$ and $\varphi(a) > \varphi(b) < \varphi(c)$ and φ is a continuous function, then there exists a local minimum of ϕ within the interval (a, c). Such an interval is called a *bracket*.

Therefore the problem of computing the upper bound $\overline{\alpha}$ simplifies into the problem of constructing a bracket for φ. If the specific line search method we are using is DGM, then we automatically compute $\varphi(0)$ and $\varphi(h)$ during the process of finding $\boldsymbol{p}_k$ (see Algorithm 5). In order to complete the bracket, we trial the value $\alpha = 1$ and identify two basic outcomes.

First if $\varphi(1) > \varphi(h)$, then the trio of values $0, h, 1$ form a bracket for φ and we have the bound we need. Otherwise if $\varphi(1) < \varphi(h)$ then we might not have found a bracket, but we have found a direction in which a large step length yields non-trivial descent.

If our goal was to apply ELS, then we'd need to trial larger steps until a bracket could be found. However, the entire motivation behind considering ELS was that Backtracking

LS (Algorithm 7) might be itself expensive when applied to NS functions. Here we note that unless $\varphi(h) > \varphi(1) > \varphi(0) - h\|\boldsymbol{p}_k\|$, then we are assured that the step length $\boldsymbol{p} = 1$ will either satisfy Equation (2.6a), or complete a bracket of φ. Motivated by this, in Algorithm 8 we propose a hybrid line search (HLS) which combines features from Backtracking LS and ELS.

Algorithm 8: hybrid line search.

```
/* Initialisation:                                                    */
input : φ // Objective function in the search direction
        1 δ > 0 // minimum step size permitted
        2 φ'(0) // Directional derivative in search direction
        3 c_1 ∈ (0,1) // Armijo Constant
        4 ϵ > δ // ELS Tolerance
5  Set α_1 = 1, α_2 = 1, k = 0.
6  if φ(δ) > φ(0) then
   |  output: 0
7  else
8  |  while φ(α_1) > φ(0) + c_1 α_1 φ'(0) do
9  |  |  if φ(δ) < φ(α_1) then
10 |  |  |  Compute α* by solving the 1D optimisation problem on the
   |  |  |   bracket (0, δ, α_1).
   |  |  |  output: α*
11 |  |  else
12 |  |  |  α_2 = α_1.
13 |  |  |  α_1 = tα_1.
14 |  |  |  if φ(α_1) < φ(α_2) then
15 |  |  |  |  Compute α* by solving the 1D optimisation problem on the
   |  |  |  |   bracket (δ, α_1, α_2).
   |  |  |  |  output: α*
   |  output: α_1
```

Algorithm 8 is our solution to overcome both weaknesses of ELS by only applying ELS when convenient. Whenever Backtracking LS (Algorithm 7) would find a suitable step length immediately, HLS will return this answer. However, when the step length $\alpha = 1$ does not yield adequate reduction of ϕ we instead check if a bracket has been found and if so apply ELS. The only case in which HLS will require more than 1 iteration and still return the outcome of BLS is when $\varphi(\alpha)$ is a decreasing function on the interval $[0, 1]$, but with very small slope.

Remark. *For simplicity, we have designed Algorithm 8 to reject BLS the moment it finds a bracket. It may be the case that in practice, the BLS process should be allowed a small number of iterations to converge before it is abandoned. We have not explored this possibility, but are aware of its existence.*

4.1.3 Running DGM with HLS

Having constructed our new LS algorithm, the hybrid line search (HLS) (see Algorithm 8), we now test its effectiveness experimentally. To do so, we use the DGM as the foundational line search method, and run it using both Algorithms 7 and 8 on a number of test functions which are listed in Appendix A.2. We have chosen to use DGM as our foundational LSM due to it being effectively a derivative-free generalised steepest descent method. For the remainder of this section, we will refer to DGM operated with BLS and HLS merely as BLS and HLS respectively.

Remark. *In our definition of the HLS we have not stated which external 1D optimisation algorithm that we use as a subroutine. Here we use an algorithm of our own design called the dynamic underestimating polynomial method, the derivation of which is the main purpose of Chapter 5.*

Remark. *When we defined the DGM in Algorithm 6, we neglected to state how the algorithm should terminate. This was because Theorem 8 provides a guarantee for the infinite sequence of points generated by Algorithm 6. For this experiment, we have halted the DGM when Line 11 is encountered, or in other words when we compute a search direction $\boldsymbol{p}_k$ which has magnitude less than the stationarity tolerance ν. For the experiments in this section, we have taken $\nu = 10^{-5}$.*

Our test functions (which we list in Appendix A.2) are taken mainly from the list of functions assembled by Haarala [65]. Given that each function has a predefined starting position, the first part of our experiment involves running both algorithms on each test function starting from the specified starting position. We add to this by randomly generating 20 additional starting points, which are sampled from the multivariate normal

distribution centred around the default starting point, and with covariance matrix equal to the identify times the norm of the default starting point.

Since each test function is defined for an arbitrary number of variables, we compare our two algorithms for $n = 2, 5, 10$ and 25. For both algorithms on each function, we record:

1. The value of ϕ at the termination point.
2. The total number of function evaluations required. This includes the function evaluations involved in constructing the discrete gradient in Algorithm 5.
3. The average number of functions evaluations during the line search per iteration.
4. The average number of discrete gradients computed per iteration.

ϕ	Alg	ϕ_{min}	$\#\phi$	$\#\varphi$	#g	Alg	ϕ_{min}	$\#\phi$	$\#\varphi$	#g
A.1	BLS	2.25e-11	257	1.7	1.08	HLS	5.72e-14	58	6.5	2
A.2	BLS	2.43e-07	5256	22.78	1.32	HLS	3.13e-08	265	7.06	1.89
A.3	BLS	6.24e-10	709	16.34	2.03	HLS	3.24e-10	77	4.67	6.33
A.4	BLS	3.42e-06	2032	25.29	1.17	HLS	6.85e-08	118	15	2.2
A.5	BLS	1.22e-06	2282	25.6	1.22	HLS	1.8e-07	118	15	2.2
A.6	BLS	3.62e-06	729	17.22	1.19	HLS	1.42e-06	67	8	2.25
A.7	BLS	5.73e-06	760	17.39	1.21	HLS	2.12e-15	77	16.67	2.33
A.9	BLS	6.19e-09	1139	17.72	1.51	HLS	7.56e-09	660	14.03	2.24
A.10	BLS	3.29e-07	1167	19.13	1.23	HLS	8.39e-09	572	13.42	2.19
A.17	BLS	0.00422	32540	25.5	1.69	HLS	1.62e-09	8624	8.72	1.95
A.18	BLS	1.07e-06	13923	26.21	1.36	HLS	4.07e-08	637	9.62	1.86

Table 4.1.1 – The outcome of running DGM with both HLS and BLS on the set of functions with known local minima found in Appendix A.2 taking $n = 2$

We present the outcome of this experiment corresponding to starting from the problem specific starting point *on the test functions for which the analytic solution is known* in Tables 4.1.1 to 4.1.4. The results corresponding to unknown analytic solutions or alternative starting points are given in Appendix A.3.

When we compare the two algorithms on 2-variable functions, the outcome is comprehensive. Looking at Table 4.1.1 we see that for every test function, DGM with HLS converges to a point with comparable or superior objective function value, and does

ϕ	Alg	ϕ_{min}	$\#\phi$	$\#\varphi$	#g	Alg	ϕ_{min}	$\#\phi$	$\#\varphi$	#g
A.1	BLS	2.21e-11	884	1.93	1.06	HLS	3.78e-11	459	12.85	1.35
A.2	BLS	9.49e-06	16013	21.89	2.02	HLS	8.22e-06	2739	9.61	2.63
A.3	BLS	0.000294	8388	24.87	3.01	HLS	0.0018	4833	24.08	1.9
A.4	BLS	1.18e-05	2047	22.11	1.48	HLS	1.21e-05	8585	28.53	1.27
A.5	BLS	3.67e-09	25612	28.66	2.39	HLS	1.1e-08	1831	15.24	3.91
A.6	BLS	4.77e-06	1098	19.53	1.5	HLS	3.4e-06	129	6.5	2.17
A.7	BLS	1.13e-05	1284	19.9	1.7	HLS	6.8e-06	16097	28.26	1.04
A.9	BLS	8.21e-10	5732	20.51	1.68	HLS	1.21e-09	1323	13.65	3.35
A.10	BLS	3.48e-06	3610	19.61	2.87	HLS	4.02e-06	20895	24.75	1.25
A.17	BLS	0.0439	6592	26.16	1.41	HLS	9.23e-05	3815	23.47	1.54
A.18	BLS	0.0453	2283	22.87	2.2	HLS	0.0454	2308	24.83	1.63

Table 4.1.2 – The outcome of running DGM with both HLS and BLS on the set of functions with known local minima found in Appendix A.2 taking $n = 5$. Whenever one algorithm has notably outperformed the other, whether due to being cheaper or finding a better solution, we have highlighted that algorithm.

ϕ	Alg	ϕ_{min}	$\#\phi$	$\#\varphi$	#g	Alg	ϕ_{min}	$\#\phi$	$\#\varphi$	#g
A.1	BLS	1.54e-11	2421	1.9	1.08	HLS	2.16e-11	2957	20.1	1.08
A.2	BLS	0.00358	16786	22.53	1.55	HLS	0.00818	1534	9.15	3.21
A.3	BLS	0.00896	58050	32.65	2.13	HLS	0.000244	57839	22.57	3.18
A.4	BLS	0.0108	64649	36.25	2.4	HLS	0.0546	47531	28.09	1.59
A.5	BLS	0.0161	65200	34.56	2.61	HLS	0.0923	7590	20.62	5.61
A.6	BLS	1e-05	1084	16.52	1.3	HLS	2.72e-06	894	4.31	5.69
A.7	BLS	1.4e-05	2700	21.22	2.05	HLS	8.11e-06	28389	31.11	1.24
A.9	BLS	2.36e-09	20360	22.68	2.01	HLS	7.74e-10	2379	15.4	3.83
A.10	BLS	6.79e-06	5693	22.01	1.99	HLS	5.69e-06	12524	25.8	1.92
A.17	BLS	0.11	60200	29.81	7.42	HLS	0.408	45849	25.48	2.25
A.18	BLS	0.00147	4738	23.33	3.44	HLS	0.00149	29321	27.16	1.4

Table 4.1.3 – The outcome of running DGM with both HLS and BLS on the set of functions with known local minima found in Appendix A.2 taking $n = 10$. Whenever one algorithm has notably outperformed the other, whether due to being cheaper or finding a better solution, we have highlighted that algorithm.

so with many fewer function evaluations. Moreover, we see that for most of the test functions, the average number of function evaluations for the LS is actually less for HLS than for BLS. However, while this experiment validates our observation from Example 4.1 for $n = 2$, we can't say whether HLS is actually an improvement over BLS until we check for functions of more variables.

When we look at Tables 4.1.2 to 4.1.4 however, the results are not nearly as clear. The recurring theme observed is that as n gets larger, the performance of HLS relative to BLS gets notably worse. Ultimately this means that even though for some test

ϕ	Alg	ϕ_{min}	$\#\phi$	$\#\varphi$	#g	Alg	ϕ_{min}	$\#\phi$	$\#\varphi$	#g
A.1	BLS	2.86e-11	12593	1.94	1.07	HLS	1.84e-11	10355	18.56	1.11
A.2	BLS	0.0161	39289	21.15	1.62	HLS	0.0343	4320	8.48	2.92
A.3	BLS	0.034	94380	33.91	2.25	HLS	0.101	68036	30.47	1.37
A.4	BLS	6.12	72380	37.24	1.28	HLS	2.49	72870	29.48	1.59
A.5	BLS	0.0457	135021	35.89	3.74	HLS	0.119	13427	22.52	7.39
A.6	BLS	1.26e-05	17775	19.16	7.05	HLS	7.03e-06	4904	6.6	9.1
A.7	BLS	2.15e-05	4002	22.74	2.63	HLS	1.32e-05	21915	22.42	2.45
A.9	BLS	9.7e-09	32416	22.72	3.07	HLS	5.38e-10	7405	15.45	6.45
A.10	BLS	0.0177	86642	35.27	1.9	HLS	0.0716	42994	24.54	4.13
A.17	BLS	0.232	100400	27.93	16.73	HLS	2.69	68596	29.89	1.41
A.18	BLS	0.25	12300	24.61	4.29	HLS	3.8	57664	25.48	1.16

Table 4.1.4 – The outcome of running DGM with both HLS and BLS on the set of functions with known local minima found in Appendix A.2 taking $n = 25$. Whenever one algorithm has notably outperformed the other, whether due to being cheaper or finding a better solution, we have highlighted that algorithm.

functions one does better than the other (in some cases spectacularly so), most of the time they perform comparably well or poorly. For more details on this experiment, and the outcomes when considering additional starting points, see Appendix A.3.

When taking a step back, we note that there exist a few functions for which the same algorithm consistently performs better, regardless of n. For example, DGM with HLS usually outperforms BLS for Problems A.2 and A.6. On closer inspection, this shouldn't be a surprise as a feature these test functions share is that there is no curvature to kinks. Therefore the intuition behind Example 4.1 would be particularly applicable.

Perhaps more interesting are the contrasting results for Problems A.9 and A.10. These two test functions are NS extensions of the same smooth test function, but assembled in different ways. In fact, to convert Problem A.9 into Problem A.10, all one needs to do is swap the Max operator with the Sum operator. What we observe here is that HLS performs better when the Max operator is on the outside, the case corresponding to when there are fewer kinks, but these kinks may be sharper.

This draws our attention to Problem A.4 and A.5 which are assembled from a common smooth test function in the same fashion. In Table 4.1.2 we see that once again, HLS performs better on the variant of this test function where the Max operator is on the outside. However, Tables 4.1.3 and 4.1.4 do not provide strong evidence to back

this up.

Within this experiment we have seen DGM with HLS perform notably faster than with BLS for low dimensional problems. However, as the dimension is increased, the performance DGM with HLS declines such that once $n = 25$, it is overall the slower option.

Despite this, we draw attention to the fact that across Tables 4.1.1 to 4.1.4, DGM with BLS consistently requires as many function evaluations for the LS step as HLS, thus demonstrating that the cost of the LS itself is not a strong ground to dismiss HLS. Therefore, the problem with using HLS as the DGM's line search subroutine lies not in the cost of HLS but rather in the step length it returns.

While we conclude that HLS is not an effective option for the DGM in higher dimensional problems, we do not yet dismiss it as a direction of research. The step lengths returned by HLS do not seem suitable for DGM in general, but they still do have greater qualitative meaning than a BLS step length (in that they identify a 1D local min instead of merely satisfying the Wolfe conditions). Since the cost of HLS is not the problem, we continue by attempting to construct a new NSO algorithm which we designed to make better use of the step lengths returned by HLS.

4.2 Convergence of generalised gradient descent

Before describing our new LSM designed to use the HLS, we need to understand the properties which it must have in order to take advantage of existing convergence results. For this, we use the theory developed by Goldstein [63]. Before we do this however, we first write out the equivalent result for smooth optimisation so that we may see clearly the theoretical consequences of NSO.

Theorem 9 (Zoutendijk [11])**.** *Consider the sequence of iterations* $\{\boldsymbol{x}_k\}$ *generated by Algorithm 1, where* $\boldsymbol{p}_k$ *is a descent direction and* α_k *satisfies the Wolfe Conditions (see Equations* (2.6a) *and* (2.6b)*). Suppose that* ϕ *is bounded below in* $\mathbb{R}^n$ *and that* ϕ *is continuously differentiable within an open set* $\mathcal{N}$ *containing the level set* $\mathcal{S} := \{\boldsymbol{x} \mid \phi(\boldsymbol{x}) \leq$

$\phi(\boldsymbol{x}_0)\}$, *where* $\boldsymbol{x}_0$ *is the starting point of Algorithm 1. Assume also that the gradient* $\nabla\phi$ *is Lipschtiz continuous on* $\mathcal{N}$*. Then it holds that:*

$$\sum_{k\geq 0} \frac{\langle \boldsymbol{p}_k | \nabla\phi(\boldsymbol{x}_k)\rangle^2}{\|\boldsymbol{p}_k\|^2} < \infty. \tag{4.1}$$

Corollary 4.1 ([11])**.** *In the setting of Theorem 9, if there exists* $\delta > 0$ *such that* $\cos\theta_k \geq \delta$ *for all* k*, where* θ_k *is the angle between* $\boldsymbol{p}_k$ *and* $\nabla\phi(\boldsymbol{x}_k)$*, then* $\lim_{k\to\infty} \|\nabla\phi(\boldsymbol{x}_k)\| \to 0$*.*

Note that Theorem 9 and Corollary 4.1 are quite general in application given that they do not require that $\boldsymbol{p}_k$ is computed in any particular way, but rather that it maintains certain properties.

Theorem 10 ([63])**.** *Consider the sequence of iterations* $\{\boldsymbol{x}_k\}$ *generated by Algorithm 1 when applied to function* ϕ*. If the set* $\mathcal{S} = \{\boldsymbol{x} \mid \phi(\boldsymbol{x}) \leq \phi(\boldsymbol{x}_0)\}$ *is bounded,* ϕ *is Lipschitz continuous on* $\mathcal{S}$*, and* $\{\alpha_k\}$ *is any positive sequence converging to 0 such that* $\lim_{n\to\infty} \sum_{k=0}^{n} \alpha_k = \infty$*, and* $\boldsymbol{x}_k \in \mathcal{D}_\epsilon(\phi, \boldsymbol{x}_k)$*, then* $\lim_{k\to 0} d(\boldsymbol{x}_k, Z) \to 0$*, where*

$$Z = \{\boldsymbol{x} \mid 0 \in \partial_C\phi(\boldsymbol{x})\}, \tag{4.2}$$

and

$$\mathcal{D}_\epsilon(\phi, \boldsymbol{x}) = \left\{\boldsymbol{p} \mid \max_{\boldsymbol{v}\in\partial_\epsilon\phi(\boldsymbol{x}_k)} \frac{\langle \boldsymbol{v}|\boldsymbol{p}\rangle}{\|\boldsymbol{p}\|} \leq -\frac{1}{2} \min_{\boldsymbol{v}\in\partial_\epsilon\phi(\boldsymbol{x}_k)} \|\boldsymbol{v}\|\right\}. \tag{4.3}$$

When comparing Theorem 10 to Theorem 9 and Corollary 4.1, we see that these results apply in a similar setting, albeit with different assumptions made about ϕ. Zoutendijk's results places additional restrictions on the step $\boldsymbol{p}_k$, and both results require that the descent direction is *good enough*. However, it is not obvious how the condition on the angle between $\boldsymbol{p}_k$ and $\nabla\phi(\boldsymbol{x}_k)$ in Corollary 4.1 relates to the restriction on $\boldsymbol{p}_k$ in Theorem 10. In order to better understand how these conditions relate to each other, we apply the additional assumption that ϕ is PS to simplify $\mathcal{D}_\epsilon(\phi, \boldsymbol{x}_k)$ (see Equation (4.3)).

To do this, we introduce the following notation:

Definition 4.2 (ϵ-smoothed gradient [63])**.** *The* ϵ*-smoothed gradient of the Lipschitz continuous function* ϕ *at the point* $\boldsymbol{x}$ *is defined by* $\boldsymbol{v}_\epsilon := \operatorname{argmin}_{\boldsymbol{v}\in\partial_\epsilon\phi(\boldsymbol{x})} \|\boldsymbol{v}\|$*.*

Definition 4.3 (Active Set of piecewise smooth functions)**.** *Let $\phi : \mathbb{R}^n \to \mathbb{R}$ be a piecewise smooth function (see Definition 4.1). Given the N subsets $\{\Omega_i\}_{i=1}^N$ of Ω, and N functions ϕ_i such that $\phi_i(\boldsymbol{x}) = \phi(\boldsymbol{x})\ \forall \boldsymbol{x} \in \Omega_i$, we define the* active set of ϕ at $\boldsymbol{x}$ *by* $\mathcal{A}_f(\boldsymbol{x}) = \{i \in \mathbb{N}_N \mid \phi(\boldsymbol{x}) = \phi_i(\boldsymbol{x})\}$.

For functions satisfying Assumption 4.1, it is known [55] that $\partial_C \phi(\boldsymbol{x}) = Conv\{\nabla\phi_i(\boldsymbol{x}) \mid i \in \mathcal{A}_\phi(\boldsymbol{x})\}$. Moreover, it follows from Definition 3.8 that:

$$\partial_\epsilon \phi(\boldsymbol{x}) = Conv\{\nabla\phi_i(\boldsymbol{y}) \mid \boldsymbol{y} \in \mathbb{B}_\epsilon(\boldsymbol{x}), \text{ and } i \in \mathcal{A}_\phi(\boldsymbol{y})\}. \tag{4.4}$$

By inserting Equation (4.4) into the Equation (4.3) and using the notation of the ϵ-smooth gradient, we find that $\mathcal{D}_\epsilon(\phi, \boldsymbol{x}_k)$ may now be expressed as:

$$\mathcal{D}_\epsilon(\phi, \boldsymbol{x}) = \left\{ \boldsymbol{p} \mid \max_{\boldsymbol{y}\in\mathbb{B}_\epsilon(\boldsymbol{x})} \max_{i\in\mathcal{A}_\phi(\boldsymbol{y})} \frac{\langle \nabla\phi_i(\boldsymbol{y}) | \boldsymbol{p} \rangle}{\|\boldsymbol{p}\|} \leq -\frac{1}{2}\|\boldsymbol{v}_\epsilon\| \right\}, \tag{4.5}$$

or equivalently

$$\mathcal{D}_\epsilon(\phi, \boldsymbol{x}) = \left\{ \boldsymbol{p} \mid \min_{\boldsymbol{y}\in\mathbb{B}_\epsilon(\boldsymbol{x})} \min_{i\in\mathcal{A}_\phi(\boldsymbol{y})} \|\nabla\phi_i(\boldsymbol{y})\| |\cos\theta_{i,k}| \geq \frac{1}{2}\|\boldsymbol{v}_\epsilon\| \right\}, \tag{4.6}$$

where $\theta_{i,k}$ is the angle between $\boldsymbol{p}$ and $\nabla\phi_i(\boldsymbol{y})$. With this form of $\mathcal{D}_\epsilon(\phi, \boldsymbol{x})$, we may now compare Theorem 10 to Theorem 9 and Corollary 4.1. The conclusion of Corollary 4.1 was that if we can bound $\cos\theta_k \geq \delta$, where θ_k is the angle between $\boldsymbol{p}_k$ and $\nabla\phi(\boldsymbol{x}_k)$, then $\|\nabla\phi(\boldsymbol{x}_k)\| \to 0$. Meanwhile for Theorem 10, we require that

$$\min_{\boldsymbol{y}\in\mathbb{B}_\epsilon(\boldsymbol{x}_k)} \min_{i\in\mathcal{A}_\phi(\boldsymbol{y})} \|\nabla\phi_i(\boldsymbol{y})\| |\cos\theta_{i,k}| \geq \frac{1}{2}\|\boldsymbol{v}_\epsilon\|,$$

in order to be guaranteed convergence. The problem is that if the true local minimum $\boldsymbol{x}^*$ is located on a kink, then in general $\nabla\phi_i(\boldsymbol{x}_k) \not\to 0$ as $\boldsymbol{x}_k \to \boldsymbol{x}^*$ (and neither does $\nabla\phi_i(\boldsymbol{y}) \to 0$ for $\boldsymbol{y} \in \mathbb{B}_\epsilon(\boldsymbol{x}_k)$). Thus if $\|\boldsymbol{v}_\epsilon\| \to 0$, then we would need to compute a direction $\boldsymbol{p}_k$ with $|\cos\theta_{i,k}| \to 0$ which is not realistic to expect.

If on the other hand, as was the case for smooth optimisation, we can only compute

$\boldsymbol{p}_k$ so accurately as to ensure $|\cos\theta_{i,k}| \geq \delta$, this implies that from the moment $\|\boldsymbol{v}_\epsilon\| \leq 2\delta \min_{\boldsymbol{y}\in\mathbb{B}_\epsilon(\boldsymbol{x}_k)} \min_{i\in\mathcal{A}_\phi(\boldsymbol{y})} \|\nabla\phi_i(\boldsymbol{y})\|$, we can no longer compute any element of $\mathcal{D}_\epsilon(\phi, \boldsymbol{x}_k)$.

Intuitively, the set $\mathcal{D}_\epsilon(\phi, \boldsymbol{x}_k)$ may be thought of as the cone of descent directions. Once this cone becomes too narrow, we may not be able to produce any of its elements, at which point Theorem 10 no longer applies. In short, when running any Generalised Gradient Descent method, we can't necessarily expect a better outcome than to converge to a point where the cone of descent directions is too narrow to continue. Therefore, now that we attempt to construct a new NSO algorithm, we are only concerned about being able to find a descent direction when the cone of descent directions is sufficiently wide.

4.3 The blind active set method

In this section, we present some ideas for a line search method designed with the intention of using HLS. This algorithm is intended for the setting which this chapter has focused on, specifically when ϕ is a PS function which satisfies Assumption 4.1. As we intend this method for use in the same context for which the DGM is designed, when testing its effectiveness we will use the DGM as a comparison.

The idea of our method is to operate like an active set method. What we mean by this is that by default it should behave like a smooth optimiser. However, once a kink is encountered by ELS, then we follow the kink until either a descent direction points away from the kink, or else we merge with another kink. This approach is inspired by the idea of $\mathcal{VU}$ algorithms [66–68].

We call this new method the BASM, because it involves finding, and then staying within sets of the form: $\{\boldsymbol{x} \mid \phi_i(\boldsymbol{x}) = \phi_j(\boldsymbol{x})\}$, while being 'blind' in the sense that it can only access the function $\phi(\boldsymbol{x}) = \max_i \phi_i$ and not each ϕ_i independently. Before proceeding, we assume that the HLS algorithm is equipped with the following property:

Assumption 4.2. *If the HLS returns α_k, the outcome of ELS, then it is known whether or not α_k was computing using BLS or ELS, and if ELS, whether α_k is a Smooth or NS*

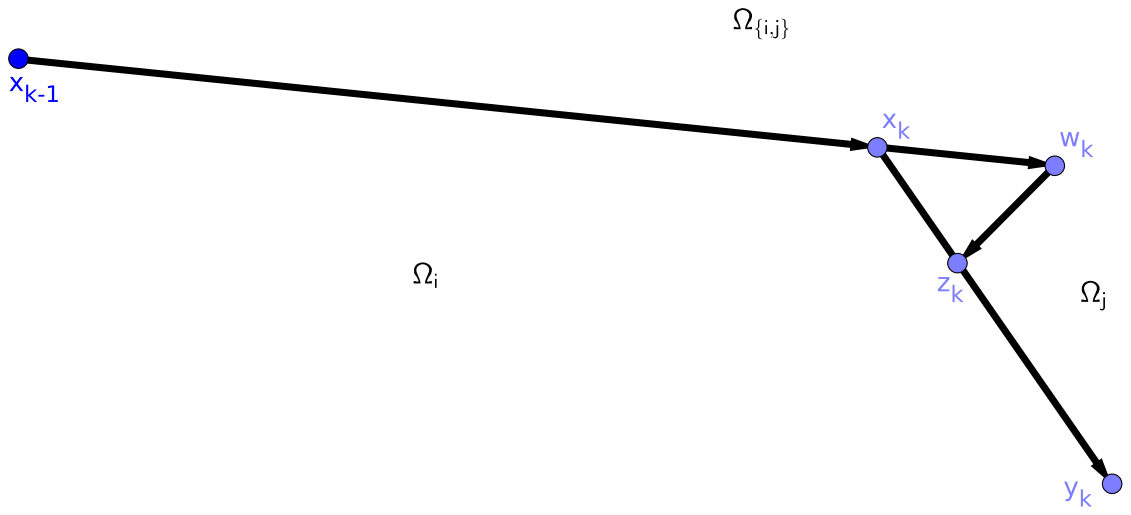

Figure 4.3.1 – A demonstration of the search direction selection procedure of the Blind Active Set method when the kink is the intersection of exactly two functions.

local optimiser of the LS function φ.

Remark. *Recall that the HLS as defined in Algorithm 8 was not complete, but rather depended on an external univariate optimiser. As Assumption 4.2 is a requirement of the univariate optimiser used, we will justify this assumption in Section 5.3.6.*

In the sections which follow, we describe how the BASM should behave once it encounters a kink, and how a descent direction is computed in this case.

4.3.1 Constructing the search direction

We design the BASM to behave like a SO algorithm until such a time as NS behaviour has been detected. Our ability to detect NS behaviour relies on Assumption 4.2. Until such a time that HLS returns a step length α which is a NS local minimum of the LS function φ, we assume that we are in a region where ϕ is locally smooth, and compute a search direction as a Smooth Optimiser would.

The moment that α is a NS local minimum of the LS function, then we know we have encountered a kink, after which we need a more elaborate process to find a search direction. The construction we use is conceptually simple, but requires many cases to be considered. In Section 4.3.1.1, we explain how we construct a search direction, and the conditions which must apply for it to work. Later in Section 4.3.1.2, we present

a more rigorous algorithm which sets up the circumstances required by the simplified construction.

4.3.1.1 Simplified construction

Let $\{\boldsymbol{x}_k\}$ be the sequence of iterations generated by the BASM. Suppose that k is an iteration such that ϕ is locally smooth near to $\boldsymbol{x}_{k-1}$, but α_{k-1} is a NS local minimum of $\varphi(\alpha) = \phi(\boldsymbol{x}_{k-1} + \alpha \boldsymbol{p}_{k-1})$ (meaning that ϕ is not locally smooth near to $\boldsymbol{x}_k$). The construction which we outline in this section depends on the following notation/assumptions.

Assumption 4.3. *The points $\boldsymbol{x}_{k-1}$ and $\boldsymbol{x}_k$ have the property that $\mathcal{A}_\phi(\boldsymbol{x}_{k-1}) = \{i\}$ and $\mathcal{A}_\phi(\boldsymbol{x}_k) = \{i, j\}$. That is, ϕ is differentiable at $\boldsymbol{x}_{k-1}$, while $\boldsymbol{x}_k$ lies on the boundary of the set Ω_i which contains $\boldsymbol{x}_{k-1}$ and in which $\phi(\boldsymbol{x}) = \phi_i(\boldsymbol{x})$.*

Notation 4.1. *Let $\boldsymbol{x}_{k-1}$ and $\boldsymbol{x}_k$ be given such that α_{k-1}, the step length satisfying $\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha_{k-1}\boldsymbol{p}_{k-1}$, was a NS local minimiser of $\varphi(\alpha) = \phi(\boldsymbol{x}_{k-1} + \alpha \boldsymbol{p}_{k-1})$. Set $\boldsymbol{w}_k = \boldsymbol{x}_k + \delta \boldsymbol{p}_{k-1}$, for some small δ. Then we write:*

- $\tilde{\boldsymbol{g}}_i = -\nabla\phi(\boldsymbol{x}_{k-1})$,
- $\boldsymbol{g}_i = -\lim_{\Omega_i \ni \boldsymbol{y} \to \boldsymbol{x}_k} \nabla\phi(\boldsymbol{y})$,
- $\boldsymbol{g}_j = -\lim_{\Omega_j \ni \boldsymbol{y} \to \boldsymbol{x}_k} \nabla\phi(\boldsymbol{y})$,
- $\tilde{\boldsymbol{g}}_j = -\nabla\phi(\boldsymbol{w}_k)$,

where i and j are the indices mentioned in Assumption 4.3.

Assumption 4.4. *The function ϕ is locally differentiable at $\boldsymbol{w}_k$.*

Remark. *The construction described by Algorithm 9 is illustrated in Figure 4.3.1.*

The intention of Algorithm 9 is to produce two points: $\boldsymbol{x}_k$ and $\boldsymbol{z}_k$, both lying on the same kink, with $\boldsymbol{z}_k - \boldsymbol{x}_k$ being a suitable direction of descent from $\boldsymbol{x}_k$. In order to make Algorithm 9 practical, we must elaborate on Line 6. Here, we have implicitly assumed that HLS returns a step length β_k which in turn yields the point $\boldsymbol{z}_k$ on the same kink as $\boldsymbol{x}_k$. However, we can't expect this to apply if HLS does not return the outcome of ELS. In Section 4.3.1.2, we present a more comprehensive version of Algorithm 9 which overcomes this problem.

Algorithm 9: The simplified blind active set method.

```
/* Initialisation:                                                    */
input : φ // Objective function
        1 x_k // The currentiterate
        2 g̃_i = −p_{k−1} // The previous search direction
        3 δ // The overstep constant
```

4 Define $\boldsymbol{w}_k = \boldsymbol{x}_k - \delta \frac{\tilde{\boldsymbol{g}}_i}{\|\tilde{\boldsymbol{g}}_i\|}$.

5 Evaluate $\tilde{\boldsymbol{g}}_j = \nabla\phi(\boldsymbol{w}_k)$, an approximation for $\boldsymbol{g}_j = \nabla\phi_j(\boldsymbol{x}_k)$.

6 Compute β_k such that $\mathcal{A}_\phi(\boldsymbol{w}_k - \beta_k\tilde{\boldsymbol{g}}_j) = \mathcal{A}_\phi(\boldsymbol{x}_k)$.

7 Define $\boldsymbol{z}_k = \boldsymbol{w}_k - \beta_k \frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|}$.

8 Set $\boldsymbol{p}_k = -\left(\frac{\|\tilde{\boldsymbol{g}}_i\|\langle\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j|\tilde{\boldsymbol{g}}_j\rangle}{\delta\|\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j\|^2}\right)(\boldsymbol{z}_k - \boldsymbol{x}_k)$.

output: $\boldsymbol{p}_k$

4.3.1.2 Full construction - non-smooth case

The main challenge to address when trying to make Algorithm 9 robust is to determine how exactly β_k should be computed. It is implicitly assumed in Algorithm 9 that $\boldsymbol{z}_k$ should lie on the same kink as $\boldsymbol{x}_k$, which we can't expect to be true if β_k was calculated using BLS. Therefore, we must ensure that ELS is used.

This brings us to the challenge of constructing a bracket for the function $\varphi(\alpha) = \phi(\boldsymbol{w}_k - \alpha\frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|})$ such that $\boldsymbol{w}_k - \alpha\frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|}$ intersects the kink for some value of α contained within the bracket. To achieve this, we make use of a new step length parameter: $\kappa \gg \delta$, and evaluate ϕ at the point $\boldsymbol{y}_k = \boldsymbol{w}_k - \kappa\frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|}$. We explain what happens next with the aid of Figure 4.3.2.

In this figure, we have plotted $\boldsymbol{x}_{k-1}$, $\boldsymbol{x}_k$, $\boldsymbol{w}_k$, and three possibilities for both $\boldsymbol{y}_k$ and $\boldsymbol{z}_k$ which are distinguished from each other using superscripts. For $\boldsymbol{y}_k^1$ and $\boldsymbol{y}_k^2$, the step to $\boldsymbol{y}_k$ takes us to the opposite side of the kink from $\boldsymbol{w}_k$, in which case it is straightforward to complete the bracket to set up a 1D optimisation (see Algorithm 10 for details). The problem is with the third option $\boldsymbol{y}_k^3$.

In Figure 4.3.2, we see that if we made the parameter κ slightly larger, then eventually $\boldsymbol{y}_k$ would be positioned on the opposite side of the kink from $\boldsymbol{w}_k$. However, there is no reason to believe this will happen in general as the kink may not intersect with the line parallel to $\tilde{\boldsymbol{g}}_j$ through $\boldsymbol{w}_k$ at all. To understand what options we have when this

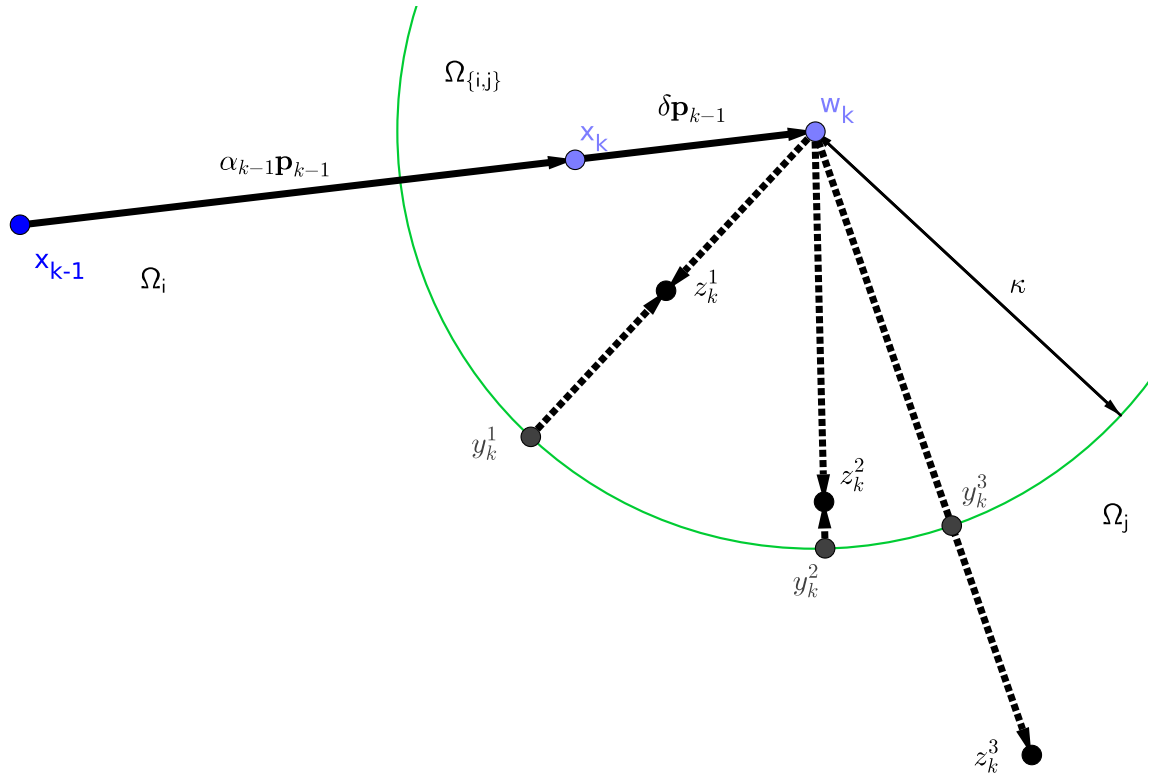

Figure 4.3.2 – A demonstration of the search direction selection procedure of the Blind Active Set method when the kink is the intersection of exactly two functions.

occurs, consider the following Proposition (after introducing some notation).

Notation 4.2. *Let $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$. We define $\omega(\boldsymbol{v}, \boldsymbol{w})$ to be the angle between $\boldsymbol{v}$ and $\boldsymbol{w}$.*

Proposition 4.1. *Let $\boldsymbol{x}_{k-1}, \boldsymbol{x}_k, \boldsymbol{w}_k, \tilde{\boldsymbol{g}}_i, \boldsymbol{g}_i, \boldsymbol{g}_j$ and $\tilde{\boldsymbol{g}}_j$ be defined according to Notation 4.1 and Algorithm 9, and $\boldsymbol{y}_k = \boldsymbol{w}_k - \kappa \frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|}$. If $2\delta < \kappa < \frac{4}{5}\|\tilde{\boldsymbol{g}}_j\|/L_j$ and $\phi(\boldsymbol{y}_k) > \phi(\boldsymbol{x}_k)$, then $\mathcal{A}_\phi(\boldsymbol{y}_k) \cap \mathcal{A}_\phi(\boldsymbol{w}_k) = \varnothing$.*

Proof. We prove this lemma by showing that if $\mathcal{A}_\phi(\boldsymbol{y}_k) \cap \mathcal{A}_\phi(\boldsymbol{w}_k) \neq \varnothing$, then $\phi(\boldsymbol{y}_k) < \phi(\boldsymbol{w}_k)$. If it is the case that $\mathcal{A}_\phi(\boldsymbol{y}_k) \cap \mathcal{A}_\phi(\boldsymbol{w}_k) \neq \varnothing$, then $\exists j \in \mathcal{A}_\phi(\boldsymbol{y}_k) \cap \mathcal{A}_\phi(\boldsymbol{w}_k)$ such that $\phi = \phi_j$ is continuously differentiable on a set containing $\boldsymbol{x}_k$ and $\boldsymbol{y}_k$. Using this, we

compute:

$$\begin{aligned}
&\phi\left(\boldsymbol{w}_k - \kappa\frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|}\right) = \phi(\boldsymbol{y}_k) < \phi(\boldsymbol{x}_k) = \phi\left(\boldsymbol{w}_k + \delta\frac{\tilde{\boldsymbol{g}}_i}{\|\tilde{\boldsymbol{g}}_i\|}\right),\\
\Leftrightarrow &\phi(\boldsymbol{w}_k) - \kappa\|\tilde{\boldsymbol{g}}_j\| + \frac{1}{2}\kappa^2\frac{\tilde{\boldsymbol{g}}_j^T\nabla^2\phi(\boldsymbol{w}_k - \eta_1\kappa\tilde{\boldsymbol{g}}_j/\|\tilde{\boldsymbol{g}}_j\|)\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|^2} <\\
&\phi(\boldsymbol{w}_k) + \delta\frac{\langle\tilde{\boldsymbol{g}}_i|\tilde{\boldsymbol{g}}_j\rangle}{\|\tilde{\boldsymbol{g}}_i\|} + \frac{1}{2}\delta^2\frac{\tilde{\boldsymbol{g}}_i^T\nabla^2\phi(\boldsymbol{w}_k + \eta_2\delta\tilde{\boldsymbol{g}}_i/\|\tilde{\boldsymbol{g}}_i\|)\tilde{\boldsymbol{g}}_i}{\|\tilde{\boldsymbol{g}}_i\|},\\
\Leftrightarrow &\kappa\|\tilde{\boldsymbol{g}}_j\| - \delta\|\tilde{\boldsymbol{g}}_j\||\cos\omega(\tilde{\boldsymbol{g}}_i,\tilde{\boldsymbol{g}}_j)| >\\
&\frac{1}{2}\kappa^2\frac{\tilde{\boldsymbol{g}}_j^T\nabla^2\phi(\boldsymbol{w}_k - \eta_1\tilde{\boldsymbol{g}}_j/\|\tilde{\boldsymbol{g}}_j\|)\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|^2} - \frac{1}{2}\delta^2\frac{\tilde{\boldsymbol{g}}_i^T\nabla^2\phi(\boldsymbol{w}_k + \eta_2\tilde{\boldsymbol{g}}_i/\|\tilde{\boldsymbol{g}}_i\|)\tilde{\boldsymbol{g}}_i}{\|\tilde{\boldsymbol{g}}_i\|^2}.
\end{aligned}$$

From here, we use the fact that $2\delta < \kappa$ to conclude:

$$\kappa\|\tilde{\boldsymbol{g}}_j\| - \delta\|\tilde{\boldsymbol{g}}_j\||\cos\omega(\tilde{\boldsymbol{g}}_i,\tilde{\boldsymbol{g}}_j)| > \|\tilde{\boldsymbol{g}}_j\|(\kappa-\delta) > \frac{1}{2}\|\tilde{\boldsymbol{g}}_j\|\kappa, \tag{4.7}$$

and

$$\begin{aligned}
\frac{1}{2}\kappa^2\frac{\tilde{\boldsymbol{g}}_j^T\nabla^2\phi(\boldsymbol{w}_k - \eta_1\tilde{\boldsymbol{g}}_j/\|\tilde{\boldsymbol{g}}_j\|)\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|^2} - \frac{1}{2}\delta^2\frac{\tilde{\boldsymbol{g}}_i^T\nabla^2\phi(\boldsymbol{w}_k + \eta_2\tilde{\boldsymbol{g}}_i/\|\tilde{\boldsymbol{g}}_i\|)\tilde{\boldsymbol{g}}_i}{\|\tilde{\boldsymbol{g}}_i\|^2}&\\
< \frac{1}{2}(\kappa^2+\delta^2)L_j < \frac{5}{8}\kappa^2 L_j.&
\end{aligned} \tag{4.8}$$

Finally, by merging Equations (4.7) and (4.8), we see that $\phi(\boldsymbol{y}_k) < \phi(\boldsymbol{x}_k)$ whenever $\frac{1}{2}\|\tilde{\boldsymbol{g}}_j\| > \frac{5}{8}\kappa L_j$. Hence, if δ and κ satisfy the conditions in this proposition, $\mathcal{A}_\phi(\boldsymbol{y}_k) \cap \mathcal{A}_\phi(\boldsymbol{w}_k) \neq \varnothing$ implies that $\phi(\boldsymbol{y}_k) < \phi(\boldsymbol{x}_k)$ and the result follows.

□

Remark. *It becomes difficult to select parameters δ and κ if $\|\tilde{\boldsymbol{g}}_j\|/L_j$ becomes too small. However, this can only happen either if $\|\tilde{\boldsymbol{g}}_j\| \to 0$, which implies that the local minimum of ϕ is nearly a smooth local minimum of ϕ_j, or that L_j is becoming too large. While we cannot prove that these cases will not occur, we do not anticipate trouble here. We expect that if $\frac{currentapprox}{\|currentapprox\|} \to 0$ for some j, then whenever $\mathcal{A}_\phi(\boldsymbol{x}_k) = \{j\}$, Algorithm 10 will not be required to compute a descent direction. However, this is mere intuition and has yet to be proved.*

We gain as much insight from the converse of Proposition 4.1 as from the direct statement. In particular, if κ is chosen appropriately, then either $\phi(\boldsymbol{y}_k) < \phi(\boldsymbol{x}_k)$ or $\boldsymbol{y}_k$ lies on the opposite side of the kink from $\boldsymbol{x}_k$. In other words, we are guaranteed to either bracket the kink, and thus construct the point $\boldsymbol{z}_k$ on the kink, or else produce a point $\boldsymbol{y}_k$ such that $\phi(\boldsymbol{y}_k) < \phi(\boldsymbol{x}_k)$. When the latter applies, we can set $\boldsymbol{p}_k = \boldsymbol{y}_k - \boldsymbol{x}_k$, $\alpha_k = 1$, and immediately update $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$.

All that is needed to extend Algorithm 9 suitably is to distinguish the cases where we succeed or fail to find points on both sides of the kink. Regardless of which option applies, we have a sensible course of action. We write the completed version of Algorithm 9 in Algorithm 10.

There are three qualitatively different ways in which Algorithm 10 might terminate. Lines 11 and 23 correspond to when $\boldsymbol{y}_k$ is on the opposite side of the kink from $\boldsymbol{w}_k$, and a bracket is found thus setting up a 1D optimisation. It is when these lines apply that Algorithm 9 operates as described. The second way Algorithm 10 might terminate is through Line 18, in which we fail to find points on both sides of the kink, but instead locate a suitable value for $\boldsymbol{x}_{k+1}$.

The third case, corresponding to Lines 20 and 26 only apply when we encounter strange behaviour from ϕ. Specifically, Line 26 is only reached if Assumption 4.4 fails. While we understand that this might occur when multiple kinks are present, as we have not yet developed the BASM for such circumstances, we leave this block unwritten.

On the other hand, Line 20 refers to a strange situation where $\phi(\boldsymbol{w}_k) > \phi(\boldsymbol{x}_k) > \phi(\boldsymbol{y}_k) > \phi(\boldsymbol{v}_k)$. Since this case corresponds to when Proposition 4.1 applies, it follows that this case should only occur when either an additional kink is encountered, since $\mathcal{A}_\phi(\boldsymbol{w}_k) \cap \mathcal{A}_\phi(\boldsymbol{y}_k) = \varnothing$.

At present, we have not equipped the BASM to handle more than a single kink. As we expect Assumption 4.4 to apply under such circumstances, we terminate the BASM whenever Lines 20 or 26 are encountered within Algorithm 10.

Algorithm 10: blind active set method following a non-smooth step.

/* Initialisation: */
input : ϕ // Objective function
1 $\boldsymbol{x}_k$ // The currentiterate
2 $\tilde{\boldsymbol{g}}_i = -\boldsymbol{p}_{k-1}$ // The previous search direction
3 δ // The overstep length constant
4 κ // The test step length constant
5 h // A finite difference step length

Define $\boldsymbol{w}_k = \boldsymbol{x}_k - \delta\frac{\tilde{\boldsymbol{g}}_i}{\|\tilde{\boldsymbol{g}}_i\|}$.
Evaluate $\tilde{\boldsymbol{g}}_j = \nabla\phi(\boldsymbol{w}_k)$, an approximation for $\boldsymbol{g}_j = \nabla\phi_j(\boldsymbol{x}_k)$.
Define $\boldsymbol{y}_k = \boldsymbol{w}_k - \kappa\frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|}$.
if $\phi(\boldsymbol{y}_k) < \phi(\boldsymbol{w}_k)$ **then**
    Define $\boldsymbol{v}_k = \boldsymbol{y}_k - h\frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|}$.
    if $\phi(\boldsymbol{v}_k) > \phi(\boldsymbol{y}_k)$ **then**
        Compute β_k by applying a univariate minimiser to $\varphi(\alpha) = \phi(\boldsymbol{w}_k - \alpha\frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|})$ using the bracket $(0, \kappa, \kappa + h)$.
    else
        if $\phi(\boldsymbol{y}_k) < \phi(\boldsymbol{x}_k)$ **then**
            $\boldsymbol{p}_k = \boldsymbol{y}_k - \boldsymbol{x}_k$.
            $\alpha_k = 1$.
            $\boldsymbol{x}_{k+1} = \boldsymbol{y}_k$.
            $\boldsymbol{p}_{k+1} = -\tilde{\boldsymbol{g}}_j$.
            output: $\boldsymbol{p}_k, \alpha_k, \boldsymbol{x}_{k+1}, \boldsymbol{p}_{k+1}$
        else
            Unwritten
else
    Define $\boldsymbol{v}_k = \boldsymbol{w}_k - h\frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|}$.
    if $\phi(\boldsymbol{v}_k) < \phi(\boldsymbol{w}_k)$ **then**
        Compute β_k by applying a univariate minimiser to $\varphi(\alpha) = \phi(\boldsymbol{w}_k - \alpha\frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|})$ using the bracket $(0, h, \kappa)$.
    else
        Impossible if Assumption 4.4 applies.
Set $\boldsymbol{p}_k = -\left(\frac{\|\tilde{\boldsymbol{g}}_i\|\langle\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j|\tilde{\boldsymbol{g}}_j\rangle}{\delta\|\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j\|^2}\right)\left(\boldsymbol{w}_k - \beta_k\frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|} - \boldsymbol{x}_k\right)$.
output: $\boldsymbol{p}_k$

4.3.1.3 Full construction - smooth case

All that remains to define the BASM is to specify more rigorously how the BASM should behave when $\boldsymbol{x}_k$ is not known to be on a kink (that is α_{k-1} was not a NS local minimum of the ELS). This we write in Algorithm 11.

Algorithm 11: blind active set method following a smooth step.

```
   /* Initialisation:                                                    */
   input : φ // Objective function
         1 x_k // The currentiterate
         2 g̃_i = −p_{k−1} // The previous search direction
         3 h // A finite difference step length
 4 p_k = −∇φ(x_k).
 5 Define v_k = x_k + h p_k/‖p_k‖
 6 if φ(v_k) < φ(x_k) then
   |  output: p_k
 7 else
 8 |  if φ(x_k + h p_{k−1}/‖p_{k−1}‖) < φ(x_k) then
   |   |  output: p_{k−1}
 9 |  else if φ(x_k − h p_{k−1}/‖p_{k−1}‖) < φ(x_k) then
   |   |  output: −p_{k−1}
10 |  else
11 |   |  Unwritten.
```

Observe that whenever ϕ is locally smooth within a ball of radius h around $\boldsymbol{x}_k$, then Algorithm 11 should terminate at Line 6. The other options only ever apply if $-\nabla\phi(\boldsymbol{x}_k)$ is not a descent direction from $\boldsymbol{x}_k$. This is most likely to occur when $\boldsymbol{p}_{k-1}$ was generated using Algorithm 10, and is nearly parallel to the kink, resulting in $\boldsymbol{x}_k$ also lying on the kink (or at least close to it). When this applies, assuming α_{k-1} was not the outcome of ELS, then we trial both $\boldsymbol{p}_{k-1}$ and $-\boldsymbol{p}_{k-1}$ as descent directions, given that there are nearly parallel to the kink. If this does not work, it either means that we are close to a minimiser, or more likely that ϕ has a shape near $\boldsymbol{x}_k$ which we have not yet taken into account.

4.3.2 BASM convergence results

Having presented Algorithms 9, 10 and 11, in this section we prove that when there is only one kink in a neighbourhood containing of $\boldsymbol{x}_{k-1}, \boldsymbol{x}_k, \boldsymbol{w}_k, \boldsymbol{y}_k$ and $\boldsymbol{z}_k$, the cone of descent directions is wide enough and the constants δ and κ are chosen appropriately, then Algorithm 10 produces a search direction contained within $\mathcal{D}_\epsilon(\phi, \boldsymbol{x}_k)$ (see Theorem 10).

In order to state how we intend to go about this, we require the following notation:

Notation 4.3. *Let $\boldsymbol{x}_k$ be a point such that $\mathcal{A}_\phi(\boldsymbol{x}_k) = \{i, j\}$ (as in the context of Algorithm*

9). We denote by $\boldsymbol{g}_{\{i,j\}}$ *the minimal normed element of* $\partial_C \phi(\boldsymbol{x}_k)$*, which in this context can be computed analytically to be:*

$$\boldsymbol{g}_{\{i,j\}} := \frac{\langle \boldsymbol{g}_j | \boldsymbol{g}_j - \boldsymbol{g}_i \rangle}{\|\boldsymbol{g}_j - \boldsymbol{g}_i\|^2} \boldsymbol{g}_i + \frac{\langle \boldsymbol{g}_i | \boldsymbol{g}_i - \boldsymbol{g}_j \rangle}{\|\boldsymbol{g}_j - \boldsymbol{g}_i\|^2} \boldsymbol{g}_j. \tag{4.9}$$

For this vector, it holds that:

$$\left\langle \boldsymbol{g}_i \middle| \boldsymbol{g}_{\{i,j\}} \right\rangle = \left\langle \boldsymbol{g}_j \middle| \boldsymbol{g}_{\{i,j\}} \right\rangle = \left\| \boldsymbol{g}_{\{i,j\}} \right\|^2 = \frac{\|\boldsymbol{g}_i\| \|\boldsymbol{g}_j\| \sin \omega(\boldsymbol{g}_i, \boldsymbol{g}_j)}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|} \tag{4.10}$$

Remark. *It is important to recognise that* $\boldsymbol{g}_{\{i,j\}}$ *is not the same as* $\boldsymbol{v}_\epsilon$*, given that* $\boldsymbol{g}_{\{i,j\}}$ *is defined with respect to the Clarke Generalised Gradient, while* $\boldsymbol{v}_\epsilon$ *depends on the Goldstein Generalised Gradient. However if* $\mathcal{A}_\phi(\boldsymbol{y}) \subseteq \{i,j\},\ \forall \boldsymbol{y} \in \mathbb{B}_\epsilon(\boldsymbol{x}_k)$ *(meaning no other kinks are found nearby), then* $\left\| \boldsymbol{g}_{\{i,j\}} - \boldsymbol{v}_\epsilon \right\| \sim \mathcal{O}(\epsilon)$*.*

For the remainder of this section, we work towards proving that $\boldsymbol{p}_k$ as calculated within Algorithm 9 approximates $-\boldsymbol{g}_{\{i,j\}}$ and bounding the error. We do so for the situation where the following assumption applies:

Assumption 4.5. *The point* $\boldsymbol{z}_k$ *as constructed in Algorithm 9 satisfies:* $\mathcal{A}_\phi(\boldsymbol{z}_k) = \mathcal{A}_\phi(\boldsymbol{x}_k)$*.*

Assumption 4.6. *The vectors* $\tilde{\boldsymbol{g}}_i, \boldsymbol{g}_i, \tilde{\boldsymbol{g}}_j$ *and* $\boldsymbol{g}_j$ *satisfy:* $\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle \geq 0$*,* $\left\langle \boldsymbol{g}_i - \boldsymbol{g}_j \middle| \tilde{\boldsymbol{g}}_j \right\rangle \leq 0$ *and* $\left\langle \tilde{\boldsymbol{g}}_i \middle| \tilde{\boldsymbol{g}}_j \right\rangle < 0$*.*

The calculations which follow rely on $\tilde{\boldsymbol{g}}_i$ and $\tilde{\boldsymbol{g}}_j$ approximating $\boldsymbol{g}_i$ and $\boldsymbol{g}_j$. Thus before continuing, we bound the approximation error in both cases.

Lemma 11. *Let* $\boldsymbol{g}_i, \boldsymbol{g}_j, \tilde{\boldsymbol{g}}_i$*, and* $\tilde{\boldsymbol{g}}_j$ *be obtained from applying Algorithm 9 to the piecewise-smooth function* ϕ*. If* ϕ *satisfies Assumption 4.1 (within each set* Ω_i*,* $\phi = \phi_i$ *has a Lipschitz Continuous gradient with Lipschitz constant* L_i*), then:*

$$\|\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i\| \leq \alpha_{k-1} \|\tilde{\boldsymbol{g}}_i\| L_i, \tag{4.11}$$

$$\left\|\boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j\right\| \leq \delta L_j. \tag{4.12}$$

Proof. From the definition of $\boldsymbol{g}_i$ and $\tilde{\boldsymbol{g}}_i$, we have:

$$\|\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i\| = \|\nabla\phi_i(\boldsymbol{x}_k) - \nabla\phi_i(\boldsymbol{x}_{k-1})\| \leq L_i\|\boldsymbol{x}_k - \boldsymbol{x}_{k-1}\| = L_i\|\alpha_{k-1}\nabla\phi_i(\boldsymbol{x}_{k-1})\| = \alpha_{k-1}L_i\|\tilde{\boldsymbol{g}}_i\|.$$

Furthermore

$$\left\|\boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j\right\| = \|\nabla\phi_j(\boldsymbol{x}_k) - \nabla\phi_j(\boldsymbol{w}_k)\| \leq L_j\|\boldsymbol{x}_k - \boldsymbol{w}_k\| = L_j\left\|\delta\frac{\tilde{\boldsymbol{g}}_i}{\|\tilde{\boldsymbol{g}}_i\|}\right\| = \delta L_j,$$

which completes the proof. □

The direction $\boldsymbol{p}_k$ as computed in Algorithm 9 is parallel to $\boldsymbol{z}_k - \boldsymbol{x}_k$. If we are to assess $\boldsymbol{p}_k$ as a descent direction, the first step is to express β_k analytically. This in turn requires an expression for β_k.

Lemma 12. *Let β_k be the value computed in Line 6 of Algorithm 9. If Assumptions 4.3, 4.5 and 4.6 apply, β_k is computed using ELS, and δ is sufficiently small to ensure:*

$$\delta \leq \frac{1}{3(1+\mu)}\frac{\left|\left\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \right\rangle\right|}{\left\|\tilde{\boldsymbol{g}}_j\right\|(L_i + L_j)}, \tag{4.13}$$

then it holds that:

$$|\beta_k + \delta\mu| \leq \frac{3\sqrt{3}}{2}\delta^2\frac{\left\|\tilde{\boldsymbol{g}}_j\right\|(L_i + L_j)}{\left|\left\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \right\rangle\right|}(1+\mu)^2. \tag{4.14}$$

where

$$\mu = \frac{\left\|\tilde{\boldsymbol{g}}_j\right\|}{\|\tilde{\boldsymbol{g}}_i\|}\left|\frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\left\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \right\rangle}\right|.$$

Proof. Since Assumption 4.5 is taken to be true, $\boldsymbol{x}_k$ and $\boldsymbol{z}_k$ have the same active set: $\{i, j\}$, meaning that they are both contained within $\Omega_i \cap \Omega_j$. By Definition 4.3, it follows that:

$$(\phi_i - \phi_j)(\boldsymbol{z}_k) = (\phi_i - \phi_j)(\boldsymbol{x}_k) = 0. \tag{4.15}$$

Taylors Theorem reveals that:

$$\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{z}_k - \boldsymbol{x}_k \rangle = -\frac{1}{2}(\boldsymbol{z}_k - \boldsymbol{x}_k)^T\boldsymbol{H}(\boldsymbol{z}_k - \boldsymbol{x}_k), \tag{4.16}$$

where $\boldsymbol{H}$ is the value of $\nabla^2(\phi_i - \phi_j)$ at some point between $\boldsymbol{z}_k$ and $\boldsymbol{x}_k$. Using the fact that $\boldsymbol{z}_k - \boldsymbol{x}_k = \delta \frac{\boldsymbol{p}_{k-1}}{\|\boldsymbol{p}_{k-1}\|} - \beta_k \frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|} = -\left(\delta \frac{\tilde{\boldsymbol{g}}_i}{\|\tilde{\boldsymbol{g}}_i\|} + \beta_k \frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|}\right)$, we now compute

$$
\begin{aligned}
&\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{z}_k - \boldsymbol{x}_k \rangle = -\frac{1}{2}(\boldsymbol{z}_k - \boldsymbol{x}_k)^T \boldsymbol{H} (\boldsymbol{z}_k - \boldsymbol{x}_k), \\
\Leftrightarrow &\left\langle \boldsymbol{g}_i - \boldsymbol{g}_j \middle| \delta \frac{\tilde{\boldsymbol{g}}_i}{\|\tilde{\boldsymbol{g}}_i\|} + \beta_k \frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|} \right\rangle = \frac{1}{2} \left(\delta \frac{\tilde{\boldsymbol{g}}_i}{\|\tilde{\boldsymbol{g}}_i\|} + \beta_k \frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|}\right)^T \boldsymbol{H} \left(\delta \frac{\tilde{\boldsymbol{g}}_i}{\|\tilde{\boldsymbol{g}}_i\|} + \beta_k \frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|}\right), \\
\Leftrightarrow &\beta_k^2 \left(\frac{1}{2\|\tilde{\boldsymbol{g}}_j\|^2} \tilde{\boldsymbol{g}}_j^T \boldsymbol{H} \tilde{\boldsymbol{g}}_j\right) + \beta_k \left(\delta \frac{\tilde{\boldsymbol{g}}_i^T \boldsymbol{H} \tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_i\|} - \langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle\right) \\
&+ \frac{1}{2}\delta^2 \frac{\tilde{\boldsymbol{g}}_i^T \boldsymbol{H} \tilde{\boldsymbol{g}}_i}{\|\tilde{\boldsymbol{g}}_i\|^2} - \delta \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\|\tilde{\boldsymbol{g}}_i\|} = 0.
\end{aligned}
$$

In the workings above, we have computed a quadratic equation which relates β_k to δ. In order to bound $|\beta_k - \mu\delta|$, we will need to solve this quadratic with respect to to β_k. To simplify the calculations which follow, we introduce the following notation which is used only in the context of this proof.

$$
\begin{aligned}
a &= \frac{1}{2\|\tilde{\boldsymbol{g}}_j\|^2} \tilde{\boldsymbol{g}}_j^T \boldsymbol{H} \tilde{\boldsymbol{g}}_j, & b &= \frac{1}{\|\tilde{\boldsymbol{g}}_i\| \|\tilde{\boldsymbol{g}}_j\|} \tilde{\boldsymbol{g}}_i^T \boldsymbol{H} \tilde{\boldsymbol{g}}_j, & c &= \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle}{\|\tilde{\boldsymbol{g}}_j\|}, \\
d &= \frac{1}{2\|\tilde{\boldsymbol{g}}_i\|^2} \tilde{\boldsymbol{g}}_i^T \boldsymbol{H} \tilde{\boldsymbol{g}}_i, & e &= \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\|\tilde{\boldsymbol{g}}_i\|}, & \mu &= \frac{\|\tilde{\boldsymbol{g}}_j\|}{\|\tilde{\boldsymbol{g}}_i\|} \left| \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \right| = \left| \frac{e}{c} \right|.
\end{aligned}
$$

Note that Assumption 4.6 is equivalent to $c \leq 0$ and $e \geq 0$. Moreover, recall $\boldsymbol{H}$ is $\nabla^2(\phi_i - \phi_j)$ evaluated at some point. Since ϕ is piecewise-smooth function satisfying Assumption 4.1, it follows that:

$$
|a|, |d| \leq \frac{1}{2}(L_i + L_j), \quad \text{and} \quad |b| \leq L_i + L_j. \tag{4.17}
$$

We now can write the quadratic which we need to solve in the form:

$$
a\beta_k^2 + (b\delta - c)\beta_k + d\delta^2 - e\delta = 0. \tag{4.18}
$$

When solving Equation (4.18) for β_k,we do so with the understanding that δ should be 'small'. Unfortunately, we can't solve Equation (4.18) directly given how much things

change if $a = 0$. Therefore, we consider two cases, that where $a = 0$ and $a \neq 0$. Beginning with the first case, we find:

$$\beta_k = -\delta\frac{e-\delta d}{c-\delta b} = -\delta\frac{e}{c} - \delta^2\frac{be-dc}{c(c-\delta b)} \;\Rightarrow\; |\beta_k - \delta\mu| = \delta^2\left|\frac{be-dc}{c(c-\delta b)}\right|. \tag{4.19}$$

In order to simplify Equation (4.19) further, we apply Equation (4.13) which was assumed to be true in the statement of the lemma. Within the context of our transformed variables, Equation (4.13) becomes:

$$\begin{aligned}
&\frac{(L_i+L_j)\|\tilde{\boldsymbol{g}}_j\|}{\left|\langle \boldsymbol{g}_i - \boldsymbol{g}_j|\tilde{\boldsymbol{g}}_j\rangle\right|}\delta \leq \frac{1}{3(1+\mu)},\\
\Rightarrow &\frac{L_i+L_j}{\|c\|}\delta \leq \frac{1}{3(1+\mu)},\\
\Rightarrow &\|b\|\delta \leq (L_i+L_j)\delta < \frac{1}{3(1+\mu)}\|c\|,\\
\Rightarrow &\|c\| - \|b\|\delta \geq \left(1 - \frac{1}{3(1+\mu)}\right)\|c\|,\\
\Rightarrow &\|c-b\delta\| \geq \frac{2+3\mu}{3+3\mu}\|c\| \geq \frac{2}{3}\|c\|.
\end{aligned}$$

We condense the following result from the workings above for future reference.

$$\|c-b\delta\| \geq \|c\| - \|b\|\delta \geq \left(1 - \frac{1}{3(1+\mu)}\right)\|c\| \geq \frac{2}{3}\|c\|. \tag{4.20}$$

Inserting this outcome along with Equation (4.17) into Equation (4.19), we find:

$$|\beta_k - \delta\mu| = \delta^2\left|\frac{be-dc}{c(c-\delta b)}\right| \leq \frac{3\delta^2}{2}\frac{|be|+|dc|}{c^2} \leq \frac{3\delta^2}{4}\frac{L_i+L_j}{|c|}(1+2\mu) \leq \frac{3\delta^2}{4}\frac{L_i+L_j}{|c|}(1+\mu)^2.$$

When reversing the substitution, this becomes

$$|\beta_k + \delta\mu| \leq \frac{3\delta^2}{4}\left(\frac{\|\tilde{\boldsymbol{g}}_j\|(L_i+L_j)}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j|\tilde{\boldsymbol{g}}_j\rangle}\right)(1+\mu)^2 < \frac{1}{2}\left(\sqrt{\frac{9}{5}}\right)^3\delta^2\left(\frac{\|\tilde{\boldsymbol{g}}_j\|(L_i+L_j)}{\left|\langle \boldsymbol{g}_i - \boldsymbol{g}_j|\tilde{\boldsymbol{g}}_j\rangle\right|}\right)(1+\mu)^2. \tag{4.21}$$

Next we move our attention to the case in which $a \neq 0$. In order to proceed, we are

forced to apply the quadratic equation, giving us:

$$\beta_k = \frac{-(b\delta - c) \pm \sqrt{(b\delta - c)^2 - 4a\delta(d\delta - e)}}{2a} \tag{4.22}$$

While the quadratic equation yields two solutions, only one is of interest to us. In the context of this calculation we assume $\delta \ll 1$ and expect β_k to be of a similar size. Given that $c < 0$ and $e > 0$, the root we are interested in corresponds to the $\pm$ term being a $+$.

In order to proceed we need to apply Taylors Theorem to remove the square root term. Defining $g(\delta) := \sqrt{(b\delta - c)^2 - 4a\delta(d\delta - e)}$, there exists $\eta \in [0, \delta]$ such that:

$$g(\delta) = g(0) + \delta g'(0) + \frac{\delta^2}{2} g''(\eta). \tag{4.23}$$

Using Mathematica to evaluate $g(0)$ and $g'(0)$, we find:

$$\begin{gathered} g(\delta) = |c| + \delta \frac{2ae - bc}{|c|} + \frac{\delta^2}{2} g''(\eta), \\ \Leftrightarrow \beta_k = \frac{c - b\delta + g(\delta)}{2a} = \frac{1}{2a}\left(c - b\delta + |c| + \delta \frac{2ae - bc}{|c|} + \frac{\delta^2}{2a} g''(\eta)\right). \end{gathered}$$

Since $c < 0$, it follows that:

$$\beta_k = \frac{c - b\delta + g(\delta)}{2a} = \frac{1}{2a}\left(-b\delta - \delta \frac{2ae - bc}{c} + \frac{\delta^2}{2} g''(\eta)\right) = -\delta \frac{e}{c} + \frac{\delta^2}{4} g''(\eta) = -\delta\mu + \frac{\delta^2}{4} g''(\eta). \tag{4.24}$$

Hence, in order to bound $|\beta_k + \delta\mu|$, all that we need to do is bound $|g''(\eta)|$. Once again using Mathematica, we obtain:

$$g''(\delta) = -\frac{4a(c^2 d - bce + ae^2)}{\sqrt{(b\delta - c)^2 - 4a\delta(d\delta - e)}^3}. \tag{4.25}$$

$$|\beta_k + \delta\mu| = \delta^2 \left| \frac{c^2 d - bce + ae^2}{\sqrt{(b\eta - c)^2 - 4a\eta(d\eta - e)}^3} \right|. \tag{4.26}$$

In order to bound the error term above, we need a bound for the denominator. Starting

from Equation (4.13) and using Equation (4.20), we compute:

$$\begin{aligned}
\left|(b\eta - c)^2 - 4a\eta(d\eta - e)\right| &\geq \left||b\eta - c|^2 - 4|a\eta(d\eta - e)|\right| \geq \left||b\delta - c|^2 - 4|a\delta(d\eta - e)|\right|, \\
&\geq \left|\left(1 - \frac{1}{3(1+\mu)}\right)^2 |c|^2 - 4\delta^2|ad| - 4\delta|ae|\right|, \\
&\geq |c|^2\left|\left(1 - \frac{1}{3(1+\mu)}\right)^2 - 4\left|\frac{a\delta}{c}\right|\left|\frac{d\delta}{c}\right| - 4\left|\frac{a\delta}{c}\right|\left|\frac{e}{c}\right|\right|, \\
&\geq |c|^2\left|\left(1 - \frac{1}{3(1+\mu)}\right)^2 - \left(\frac{1}{3(1+\mu)}\right)^2 - 2\mu\left(\frac{1}{3(1+\mu)}\right)\right|, \\
&\geq |c|^2\left|1 - \frac{2(1+\mu)}{3(1+\mu)}\right| = \frac{1}{3}|c|^2.
\end{aligned}$$

With the denominator now bounded from below, we revisit Equation (4.24) to find:

$$|\beta_k + \delta\mu| \leq 3\sqrt{3}\delta^2\left|\frac{c^2 d - bce + ae^2}{|c|^3}\right| \leq 3\sqrt{3}\delta^2\frac{1}{|c|^3}\left(\left|c^2 d\right| + |bce| + \left|ae^2\right|\right). \tag{4.27}$$

Finally we reverse the substitutions to find:

$$\begin{aligned}
|\beta_k + \delta\mu| &\leq 3\sqrt{3}\delta^2\frac{|c^2 d| + |bce| + |ae^2|}{|c|^3}, \\
&\leq 3\sqrt{3}\delta^2\left(\left|\frac{d}{c}\right| + \left|\frac{be}{c^2}\right| + \left|\frac{ae^2}{c^3}\right|\right), \\
&\leq \frac{3\sqrt{3}}{2}\delta^2\frac{L_i + L_j}{|c|}\left(1 + 2\mu + \mu^2\right), \\
&= \frac{3\sqrt{3}}{2}\delta^2\frac{\left\|\tilde{\boldsymbol{g}}_j\right\|(L_i + L_j)}{\left\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \right\rangle}(1+\mu)^2.
\end{aligned}$$

Combining this result with Equation (4.21) completes the proof. □

Remark. *Lemma 12 shows that $\beta_k \sim \mathcal{O}(\delta)$ when the bound RHS of Equation (4.14) is not too large. This bound gets large when either $L_i + L_j$ is large, $\left\|\tilde{\boldsymbol{g}}_j\right\|$ is large, or the vector $\tilde{\boldsymbol{g}}_j$ is nearly perpendicular to $\boldsymbol{g}_i - \boldsymbol{g}_j$. This outcome is intuitive when Figure 4.3.2, the illustration of Algorithm 9 is taken into account, since if $\tilde{\boldsymbol{g}}_j$ is perpendicular to the normal vector of the kink, then of course β_k will be large, and our approximation will be inaccurate. If however the kink is having a sufficient effect as to render smooth optimisation methods ineffective, then we don't expect this to happen.*

Lemma 13. *Let the points $\boldsymbol{x}_{k-1}, \boldsymbol{x}_k, \boldsymbol{w}_k$ and β_k be obtained from Algorithm 9, μ be as defined in Lemma 12, and $\tilde{\beta}_k = \delta\mu$ be an approximation of $\boldsymbol{z}_k$. Then $\boldsymbol{p}_k$ as defined in Algorithm 9 may be written in the form:*

$$\tau \boldsymbol{p}_k = -\left(\boldsymbol{g}_{\{i,j\}} + \boldsymbol{E}_i + \boldsymbol{E}_j\right), \tag{4.28}$$

where

$$\tau = \frac{\left\|\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j\right\|^2}{\left\|\boldsymbol{g}_i - \boldsymbol{g}_j\right\|^2} \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle}{\left\langle \tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j | \tilde{\boldsymbol{g}}_j \right\rangle}, \tag{4.29}$$

$$\boldsymbol{E}_i = \frac{1}{\left\|\boldsymbol{g}_i - \boldsymbol{g}_j\right\|^2} \left(\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle \left(\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i\right) + \langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i \rangle \tilde{\boldsymbol{g}}_j \right), \tag{4.30}$$

$$\boldsymbol{E}_j = \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle}{\left\|\boldsymbol{g}_i - \boldsymbol{g}_j\right\|^2} \left(\frac{\tilde{\beta}_k - \beta_k}{\delta} \frac{\|\tilde{\boldsymbol{g}}_i\|}{\left\|\tilde{\boldsymbol{g}}_j\right\|} \tilde{\boldsymbol{g}}_j - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} (\boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j) - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\left\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \right\rangle} \frac{\left\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j \right\rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \tilde{\boldsymbol{g}}_j \right). \tag{4.31}$$

Moreover, the error terms: $\boldsymbol{E}_i$ and $\boldsymbol{E}_j$ may be bounded by:

$$\|\boldsymbol{E}_i\| \leq \frac{\alpha_{k-1} \|\tilde{\boldsymbol{g}}_i\| \left\|\tilde{\boldsymbol{g}}_j\right\| (L_i + L_j)}{\left\|\boldsymbol{g}_i - \boldsymbol{g}_j\right\|} \left(\frac{\|\boldsymbol{g}_j\|}{\left\|\tilde{\boldsymbol{g}}_j\right\|} + 1 \right), \tag{4.32}$$

$$\|\boldsymbol{E}_j\| \leq \frac{\delta \|\tilde{\boldsymbol{g}}_i\| \left\|\tilde{\boldsymbol{g}}_j\right\| (L_i + L_j)}{\left|\left\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \right\rangle\right|} \left(\frac{3\sqrt{3}}{2} (1+\mu)^2 + \frac{\|\boldsymbol{g}_i\|}{\|\tilde{\boldsymbol{g}}_i\|} + 1 \right), \tag{4.33}$$

where μ is as defined in Lemma 12.

Proof. From the definition of $\boldsymbol{p}_k$ in Algorithm 9, we have

$$\begin{aligned}
\boldsymbol{z}_k - \boldsymbol{x}_k &= \delta \frac{\boldsymbol{p}_{k-1}}{\|\boldsymbol{p}_{k-1}\|} - \beta_k \frac{\nabla \phi_j(\boldsymbol{w}_k)}{\|\nabla \phi_j(\boldsymbol{w}_k)\|} = -\delta \frac{\tilde{\boldsymbol{g}}_i}{\|\tilde{\boldsymbol{g}}_i\|} - \beta_k \frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|}, \\
&= -\delta \frac{\tilde{\boldsymbol{g}}_i}{\|\tilde{\boldsymbol{g}}_i\|} - \tilde{\beta}_k \frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|} + (\tilde{\beta}_k - \beta_k) \frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|}, \\
&= -\frac{\delta}{\|\tilde{\boldsymbol{g}}_i\|} \left(\tilde{\boldsymbol{g}}_i - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \tilde{\boldsymbol{g}}_j \right) + (\tilde{\beta}_k - \beta_k) \frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|}, \\
&= -\frac{\delta}{\|\tilde{\boldsymbol{g}}_i\|} \left(\boldsymbol{g}_i - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \boldsymbol{g}_j \right) + (\tilde{\beta}_k - \beta_k) \frac{\tilde{\boldsymbol{g}}_j}{\|\tilde{\boldsymbol{g}}_j\|} \\
&\quad + \frac{\delta}{\|\tilde{\boldsymbol{g}}_i\|} (\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i) - \frac{\delta}{\|\tilde{\boldsymbol{g}}_i\|} \left(\frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \boldsymbol{g}_j - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \tilde{\boldsymbol{g}}_j \right).
\end{aligned}$$

Next, we observe that the expression $\boldsymbol{g}_i - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \boldsymbol{g}_j$ may be scaled to match $\boldsymbol{g}_{\{i,j\}}$. Indeed:

$$\begin{aligned}
\boldsymbol{p}_k &= -\left(\frac{\|\tilde{\boldsymbol{g}}_i\| \langle \tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j | \tilde{\boldsymbol{g}}_j \rangle}{\delta \|\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j\|^2} \right) (\boldsymbol{z}_k - \boldsymbol{x}_k), \\
&= -\frac{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2}{\|\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j\|^2} \frac{\langle \tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j | \tilde{\boldsymbol{g}}_j \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \left(\frac{\langle \boldsymbol{g}_i | \boldsymbol{g}_i - \boldsymbol{g}_j \rangle \boldsymbol{g}_j + \langle \boldsymbol{g}_j | \boldsymbol{g}_j - \boldsymbol{g}_i \rangle \boldsymbol{g}_i}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2} \right) - \frac{\tilde{\beta}_k - \beta_k}{\delta} \frac{\|\tilde{\boldsymbol{g}}_i\|}{\|\tilde{\boldsymbol{g}}_j\|} \frac{\langle \tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j | \tilde{\boldsymbol{g}}_j \rangle}{\|\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j\|^2} \tilde{\boldsymbol{g}}_j \\
&\quad - \frac{\langle \tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j | \tilde{\boldsymbol{g}}_j \rangle}{\|\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j\|^2} (\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i) + \frac{\langle \tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j | \tilde{\boldsymbol{g}}_j \rangle}{\|\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j\|^2} \left(\frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \boldsymbol{g}_j - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \tilde{\boldsymbol{g}}_j \right), \\
&= -\frac{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2}{\|\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j\|^2} \frac{\langle \tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j | \tilde{\boldsymbol{g}}_j \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \boldsymbol{g}_{\{i,j\}} - \frac{\tilde{\beta}_k - \beta_k}{\delta} \frac{\|\tilde{\boldsymbol{g}}_i\|}{\|\tilde{\boldsymbol{g}}_j\|} \frac{\langle \tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j | \tilde{\boldsymbol{g}}_j \rangle}{\|\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j\|^2} \tilde{\boldsymbol{g}}_j \\
&\quad - \frac{\langle \tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j | \tilde{\boldsymbol{g}}_j \rangle}{\|\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j\|^2} (\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i) + \frac{\langle \tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j | \tilde{\boldsymbol{g}}_j \rangle}{\|\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j\|^2} \left(\frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \boldsymbol{g}_j - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \tilde{\boldsymbol{g}}_j \right) \right), \\
&= -\frac{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2}{\|\tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j\|^2} \frac{\langle \tilde{\boldsymbol{g}}_i - \tilde{\boldsymbol{g}}_j | \tilde{\boldsymbol{g}}_j \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \left(\boldsymbol{g}_{\{i,j\}} + \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2} \left(\frac{\tilde{\beta}_k - \beta_k}{\delta} \frac{\|\tilde{\boldsymbol{g}}_i\|}{\|\tilde{\boldsymbol{g}}_j\|} \tilde{\boldsymbol{g}}_j \right. \right. \\
&\quad \left. \left. + (\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i) - \left(\frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \boldsymbol{g}_j - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \tilde{\boldsymbol{g}}_j \right) \right) \right).
\end{aligned}$$

We proceed by scaling by τ.

$$\begin{aligned}
\tau \boldsymbol{p}_k &= -\left(\boldsymbol{g}_{\{i,j\}} + \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2} \left(\frac{\tilde{\beta}_k - \beta_k}{\delta} \frac{\|\tilde{\boldsymbol{g}}_i\|}{\|\tilde{\boldsymbol{g}}_j\|} \tilde{\boldsymbol{g}}_j + (\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i) \right.\right. \\
&\quad \left.\left. - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} (\boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j) - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle \langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle - \langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle \langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle \langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \tilde{\boldsymbol{g}}_j \right)\right), \\
&= -\left(\boldsymbol{g}_{\{i,j\}} + \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2} \left(\frac{\tilde{\beta}_k - \beta_k}{\delta} \frac{\|\tilde{\boldsymbol{g}}_i\|}{\|\tilde{\boldsymbol{g}}_j\|} \tilde{\boldsymbol{g}}_j + (\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i) \right.\right. \\
&\quad \left.\left. - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} (\boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j) - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i \rangle \langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle - \langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle \langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle \langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \tilde{\boldsymbol{g}}_j \right)\right), \\
&= -\left(\boldsymbol{g}_{\{i,j\}} + \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2} \left(\frac{\tilde{\beta}_k - \beta_k}{\delta} \frac{\|\tilde{\boldsymbol{g}}_i\|}{\|\tilde{\boldsymbol{g}}_j\|} \tilde{\boldsymbol{g}}_j + (\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i) \right.\right. \\
&\quad \left.\left. - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} (\boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j) - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \tilde{\boldsymbol{g}}_j + \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \tilde{\boldsymbol{g}}_j \right)\right).
\end{aligned}$$

Finally, we collect all terms containing the $(\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i)$ expression to be $\boldsymbol{E}_i$, and the terms containing $(\boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j)$ or $(\beta_k - \tilde{\beta}_k)$ to be $\boldsymbol{E}_j$.

$$\begin{aligned}
\tau \boldsymbol{p}_k &= -\left(\boldsymbol{g}_{\{i,j\}} + \frac{1}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2} \left(\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle (\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i) - \langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i \rangle \tilde{\boldsymbol{g}}_j \right) \right. \\
&\quad \left. + \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2} \left(\frac{\tilde{\beta}_k - \beta_k}{\delta} \frac{\|\tilde{\boldsymbol{g}}_i\|}{\|\tilde{\boldsymbol{g}}_j\|} \tilde{\boldsymbol{g}}_j - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} (\boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j) + \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \tilde{\boldsymbol{g}}_j \right)\right), \\
&= -\left(\boldsymbol{g}_{\{i,j\}} + \boldsymbol{E}_i + \boldsymbol{E}_j \right).
\end{aligned}$$

To complete this proof, we must bound $\|\boldsymbol{E}_i\|$ and $\|\boldsymbol{E}_j\|$.

$$\begin{aligned}
\|\boldsymbol{E}_i\| &= \frac{1}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2} \left\| \langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle (\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i) - \langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i \rangle \tilde{\boldsymbol{g}}_j \right\|, \\
&\leq \frac{1}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2} \left(\|\boldsymbol{g}_i - \boldsymbol{g}_j\| \|\boldsymbol{g}_j\| \|\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i\| + \|\boldsymbol{g}_i - \boldsymbol{g}_j\| \|\tilde{\boldsymbol{g}}_j\| \|\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i\| \right), \\
&= \frac{\|\boldsymbol{g}_i - \tilde{\boldsymbol{g}}_i\|}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|} (\|\boldsymbol{g}_j\| + \|\tilde{\boldsymbol{g}}_j\|) \leq \frac{\alpha_{k-1} \|\tilde{\boldsymbol{g}}_i\| (L_i + L_j)}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|} \left(\|\boldsymbol{g}_j\| + \|\tilde{\boldsymbol{g}}_j\| \right).
\end{aligned}$$

Last of all, we have:

$$
\begin{aligned}
\|\boldsymbol{E}_j\| &= \frac{|\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle|}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2} \left\| \frac{\tilde{\beta}_k - \beta_k}{\delta} \frac{\|\tilde{\boldsymbol{g}}_i\|}{\|\tilde{\boldsymbol{g}}_j\|} \tilde{\boldsymbol{g}}_j - \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} (\boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j) + \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \tilde{\boldsymbol{g}}_j \right\|, \\
&\leq \frac{|\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle|}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2} \left(\frac{|\tilde{\beta}_k - \beta_k|}{\delta} \|\tilde{\boldsymbol{g}}_i\| + \left| \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \right| \|\boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j\| + \left| \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \right| \left| \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \right| \|\tilde{\boldsymbol{g}}_j\| \right), \\
&\leq \frac{|\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle|}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2} \left(\frac{3\sqrt{3}}{2} \delta \frac{\|\tilde{\boldsymbol{g}}_i\| \|\tilde{\boldsymbol{g}}_j\| (L_i + L_j)}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} (1+\mu)^2 \right. \\
&+ \left. \left| \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \right| \|\boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j\| + \left| \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \right| \left| \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle} \right| \|\tilde{\boldsymbol{g}}_j\| \right), \\
&\leq \frac{1}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2} \left(\frac{3\sqrt{3}}{2} \delta \left| \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_j \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \right| \|\tilde{\boldsymbol{g}}_i\| \|\tilde{\boldsymbol{g}}_j\| (L_i + L_j) (1+\mu)^2 \right. \\
&+ \left. \|\boldsymbol{g}_j - \tilde{\boldsymbol{g}}_j\| \left(|\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \boldsymbol{g}_i \rangle| + \left| \frac{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_i \rangle}{\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle} \right| \|\boldsymbol{g}_i - \boldsymbol{g}_j\| \|\tilde{\boldsymbol{g}}_j\| \right) \right), \\
&\leq \frac{\delta (L_i + L_j)}{|\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle| \|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2} \left(\frac{3\sqrt{3}}{2} \|\boldsymbol{g}_i - \boldsymbol{g}_j\| \|\boldsymbol{g}_j\| \|\tilde{\boldsymbol{g}}_i\| \|\tilde{\boldsymbol{g}}_j\| (1+\mu)^2 \right. \\
&+ \left. \|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2 \|\tilde{\boldsymbol{g}}_j\| \|\boldsymbol{g}_i\| + \|\boldsymbol{g}_i - \boldsymbol{g}_j\|^2 \|\tilde{\boldsymbol{g}}_i\| \|\tilde{\boldsymbol{g}}_j\| \right), \\
&\leq \frac{\delta \|\tilde{\boldsymbol{g}}_i\| \|\tilde{\boldsymbol{g}}_j\| (L_i + L_j)}{|\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle|} \left(\frac{3\sqrt{3}}{2} \frac{\|\boldsymbol{g}_j\|}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|} (1+\mu)^2 + \frac{\|\boldsymbol{g}_i\|}{\|\tilde{\boldsymbol{g}}_i\|} + 1 \right), \\
&\leq \frac{\delta \|\tilde{\boldsymbol{g}}_i\| \|\tilde{\boldsymbol{g}}_j\| (L_i + L_j)}{|\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle|} \left(\frac{3\sqrt{3}}{2} (1+\mu)^2 + \frac{\|\boldsymbol{g}_i\|}{\|\tilde{\boldsymbol{g}}_i\|} + 1 \right).
\end{aligned}
$$

□

At this point we have related $\boldsymbol{p}_k$ as obtained from Algorithm 9 to $-\boldsymbol{g}_{\{i,j\}}$, the direction of steepest descent. In order for this relation to imply that $\boldsymbol{p}_k$ is a descent direction, we need to show that $\boldsymbol{p}_k \in \mathcal{D}_\epsilon(\phi, \boldsymbol{x}_k)$. In the following results, we show that this is achieved if the following conditions are satisfied:

Condition 4.1. *There exits* $\kappa_1 \in (0, \frac{1}{4})$ *such that*

$$
\|\boldsymbol{E}_i\| < \kappa_1 \frac{\|\boldsymbol{g}_{\{i,j\}}\|^2}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|}. \tag{4.34}
$$

Condition 4.2. *There exits* $\kappa_2 \in (0, \frac{1}{4})$ *such that*

$$\|\boldsymbol{E}_j\| < \kappa_2 \frac{\left\|\boldsymbol{g}_{\{i,j\}}\right\|^2}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|}. \tag{4.35}$$

Corollary 4.2. *If* α_{k-1} *and* δ *are sufficiently small to ensure Conditions 4.1 and 4.2 are satisfied, then it holds that* $\left\|\boldsymbol{g}_{\{i,j\}}\right\|(1-\kappa_1-\kappa_2) < \|\tau \boldsymbol{p}_k\| < \left\|\boldsymbol{g}_{\{i,j\}}\right\|(1+\kappa_1+\kappa_2)$.

Proof. Starting from Equation (4.28), we get

$$\left|\frac{\|\tau \boldsymbol{p}_k\| - \left\|\boldsymbol{g}_{\{i,j\}}\right\|}{\left\|\boldsymbol{g}_{\{i,j\}}\right\|}\right| = \left|\frac{\left\|\boldsymbol{g}_{\{i,j\}} + \boldsymbol{E}_i + \boldsymbol{E}_j\right\| - \left\|\boldsymbol{g}_{\{i,j\}}\right\|}{\left\|\boldsymbol{g}_{\{i,j\}}\right\|}\right| \leq \frac{\|\boldsymbol{E}_i\| + \|\boldsymbol{E}_j\|}{\left\|\boldsymbol{g}_{\{i,j\}}\right\|}. \tag{4.36}$$

Since Conditions 4.1 and 4.2 are satisfied, this becomes:

$$\left|\frac{\|\tau \boldsymbol{p}_k\| - \left\|\boldsymbol{g}_{\{i,j\}}\right\|}{\left\|\boldsymbol{g}_{\{i,j\}}\right\|}\right| \leq \frac{\left\|\boldsymbol{g}_{\{i,j\}}\right\|}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|}(\kappa_1+\kappa_2) \leq (\kappa_1+\kappa_2), \tag{4.37}$$

which is equivalent to $\left\|\boldsymbol{g}_{\{i,j\}}\right\|(1-\kappa_1-\kappa_2) < \|\tau \boldsymbol{p}_k\| < \left\|\boldsymbol{g}_{\{i,j\}}\right\|(1+\kappa_1+\kappa_2)$. □

Theorem 14. *Let the* $\boldsymbol{p}_k$ *be the search direction obtained from Algorithm 9. If* α_{k-1} *and* δ *are sufficiently small so as to ensure that Conditions 4.1 and 4.2 are satisfied, and* $\kappa_1 + \kappa_2 \leq \frac{1}{3}$, *then it holds that:* $\boldsymbol{p}_k \in \lim_{\epsilon \to 0} \mathcal{D}_\epsilon(\phi, \boldsymbol{x}_k)$.

Proof. First note that in the limiting case where $\lim_{\epsilon\to 0} \boldsymbol{v}_\epsilon = \boldsymbol{g}_{\{i,j\}}$ (see Definition 4.2)then the statement of this theorem is equivalent to:

$$\min_{\ell \in \{i,j\}} \frac{\langle \nabla \phi_\ell(\boldsymbol{x}_k) | \boldsymbol{p}_k \rangle}{\|\boldsymbol{p}_k\|} = \min_{\boldsymbol{y} \in \mathbb{B}_\epsilon(\boldsymbol{x}_k)} \min_{\ell \in \{i,j\}} \frac{\langle \nabla \phi_\ell(\boldsymbol{y}) | \boldsymbol{p}_k \rangle}{\|\boldsymbol{p}_k\|} \geq \frac{1}{2} \lim_{\epsilon \to 0} \|\boldsymbol{v}_\epsilon\| = \frac{1}{2}\left\|\boldsymbol{g}_{\{i,j\}}\right\|. \tag{4.38}$$

We proceed by invoking Lemma 13 and Corollary 4.2:

$$\begin{aligned}\frac{|\langle \boldsymbol{g}_\ell | \tau \boldsymbol{p}_k \rangle|}{\|\tau \boldsymbol{p}_k\|} &\geq \frac{\left|\left\langle \boldsymbol{g}_\ell \middle| \boldsymbol{g}_{\{i,j\}} + \boldsymbol{E}_i + \boldsymbol{E}_j \right\rangle\right|}{(1+\kappa_1+\kappa_2)\left\|\boldsymbol{g}_{\{i,j\}}\right\|} = \frac{\left|\left\|\boldsymbol{g}_{\{i,j\}}\right\|^2 + \langle \boldsymbol{g}_\ell | \boldsymbol{E}_i + \boldsymbol{E}_j \rangle\right|}{(1+\kappa_1+\kappa_2)\left\|\boldsymbol{g}_{\{i,j\}}\right\|}, \\ &\geq \frac{\left|\left\|\boldsymbol{g}_{\{i,j\}}\right\|^2 - \|\boldsymbol{g}_\ell\|(\|\boldsymbol{E}_i\| + \|\boldsymbol{E}_j\|)\right|}{(1+\kappa_1+\kappa_2)\left\|\boldsymbol{g}_{\{i,j\}}\right\|} \geq \frac{\left|\left\|\boldsymbol{g}_{\{i,j\}}\right\|^2 - \dfrac{\|\boldsymbol{g}_\ell\|}{\|\boldsymbol{g}_i - \boldsymbol{g}_j\|}\left\|\boldsymbol{g}_{\{i,j\}}\right\|^2 (\kappa_1+\kappa_2)\right|}{(1+\kappa_1+\kappa_2)\left\|\boldsymbol{g}_{\{i,j\}}\right\|}, \\ &\geq \frac{1}{2}\left\|\boldsymbol{g}_{\{i,j\}}\right\|,\end{aligned}$$

since $\kappa_1 + \kappa_2 \leq \frac{1}{3}$ (which was assumed in the statement of the theorem). □

When the conditions for Theorem 14 hold, Theorem 10 applies which in turn shows convergence for the BASM.

Remark. *We can't ensure Conditions 4.1 and 4.2 are satisfied, and consequentially that Theorem 14 applies, as* $\left\|\boldsymbol{g}_{\{i,j\}}\right\| \to 0$. *However, if* $\|\boldsymbol{g}_i - \boldsymbol{g}_j\| \not\to 0$, *then* $\left\|\boldsymbol{g}_{\{i,j\}}\right\| \to 0$ *will only occur when the cone of descent directions collapses, meaning that we can only construct a descent direction when this cone is not too narrow. This should not be a surprise as it was the conclusion of Section 4.2.*

4.3.3 Limitations of the analysis

In Section 4.3.2, we proved at length that under certain conditions, $\boldsymbol{p}_k$ as constructed by Algorithm 9 would satisfy Theorem 10. This is not quite the same as showing that Theorem 10 is applicable to an algorithm which constructs $\boldsymbol{p}_k$ using Algorithm 10. Here we list a number of holes in our theory, which may be viewed as a list of what needs to be proved in order show that the BASM converges.

1. All of the results contained within Section 4.3.2 rely on the assumptions that the active sets at $\boldsymbol{x}_{k-1}, \boldsymbol{w}_k$ and $\boldsymbol{x}_k$ are of the form $\{i\}$, $\{j\}$ and $\{i,j\}$ and that Assumption 4.4 holds.

2. The bound in Lemma 12 (one of the foundational results in Section 4.3.2) breaks down if $\langle \boldsymbol{g}_i - \boldsymbol{g}_j | \tilde{\boldsymbol{g}}_j \rangle$ approaches 0. While we hope that Algorithm 10 will not apply an ELS in this situation, we have not proven this.

3. While we have taken inaccuracy of gradients into account, we have not included the LS error. This comes from the fact that ELS never solves the univariate problem exactly, but only to a tolerance.

These are all points which either we have not had the time to attempt, or have been unsuccessful in addressing.

4.3.4 Performance of the BASM

Despite the fact that we have not established convergence for the BASM in general, we still test it against DGM with BLS and HLS as we would a heuristic. On this occasion, recognising that the BASM is not complete, and thus not suitable for high dimensional problems, we only run the experiment for $n = 2, 5$.

Before we present the full results in tabular form, we have selected a few select examples which illustrate how the BASM may perform, both when it performs well, and otherwise. First, in Figure 4.3.3 we see an example where BASM keeps pace with the other two algorithms, before ultimately converging both first, and to the best minimum found. On the other hand, in Figure 4.3.4 we see a case where the BASM gets stuck, and it takes a long time to do this. Finally Figures 4.3.5 and 4.3.6 show examples where the BASM performs quite well early, but then gets stuck.

From our experience with the BASM, the main reason why it so often gets stuck early is that we have not formulated a proper termination condition. As is, we instruct the BASM to stop whenever Assumption 4.4 does not apply. Aside from this, the BASM also gets stuck when $\tilde{\boldsymbol{g}}_j$ is nearly parallel (but pointing in the opposite direction) to $\tilde{\boldsymbol{g}}_i$. While sometimes a fascinating consequence of BASM enables $\boldsymbol{p}_k$ to be constructed even here (see Section 4.3.5), in general Algorithm 10 will struggle at such times.

We show the formal experiment results in Tables 4.3.1 and 4.3.2. From Table 4.3.1,

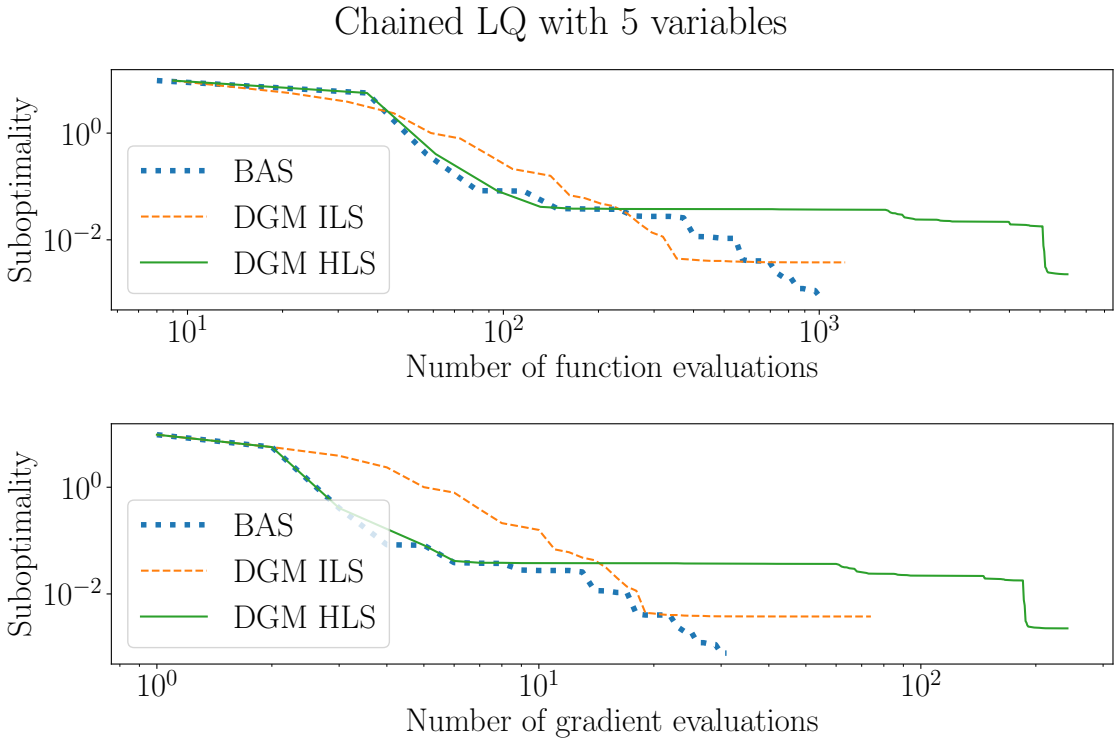

Figure 4.3.3 – We compare the DGM with both LS algorithms against the BASM on Problem A.3 with $n = 5$. Note that we only count the cost of function/gradient evaluations at the end of each iteration. Since the data for the first coordinates were harvested after the first (as opposed to zero) iteration, despite the fact that each algorithm was initialised from the same starting point, our plotted lines do not originate from the same coordinate since each algorithm required a different cost for the first iteration.

the general trend we get is that BASM is very fast when it works (though not as fast as DGM with HLS), but doesn't always converge. This is what is also observed from Table 4.3.2. However, since DGM with HLS is no longer as effective, BASM has the best performance for some test functions.

The fact that BASM improves with respect to DGM with HLS as the number of variables increase is promising, given that it was for this purpose that we derived the algorithm. However, on the whole the fact that BASM is incomplete is abundantly clear. Before any statement can be made as to its effectiveness, we need to extend its ability to construct descent directions.

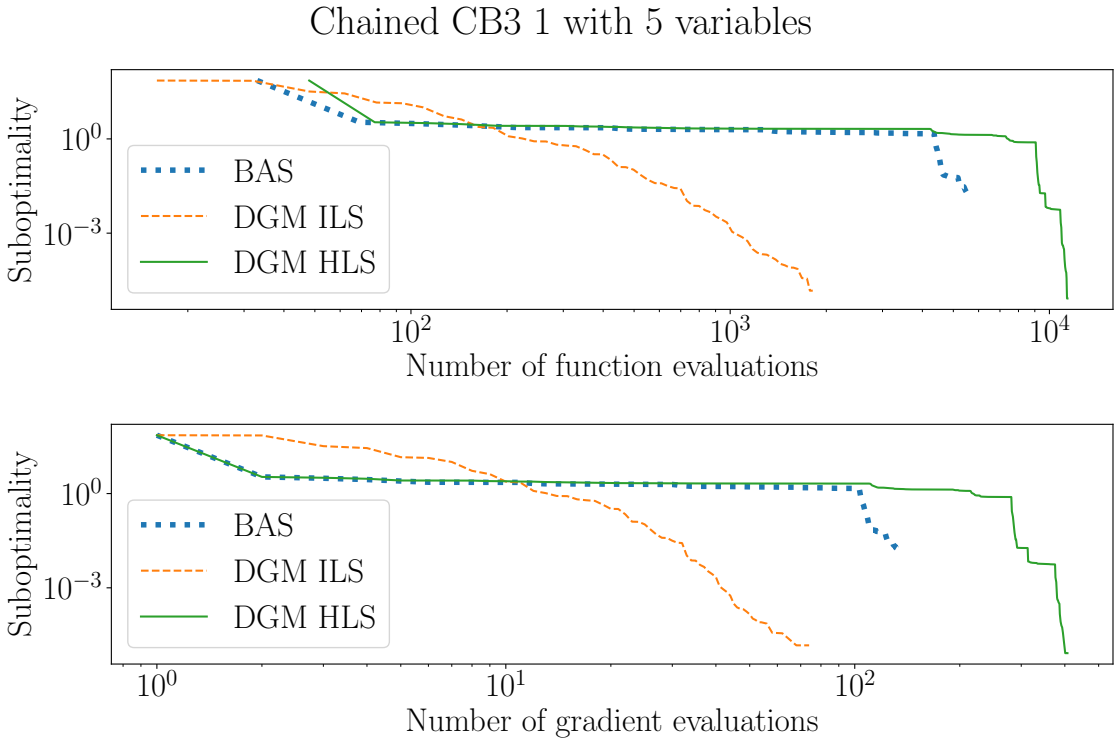

Figure 4.3.4 – We compare the DGM with both LS algorithms against the BASM on Problem A.4 with $n = 5$. Note that we only count the cost of function/gradient evaluations at the end of each iteration. Since the data for the first coordinates were harvested after the first (as opposed to zero) iteration, despite the fact that each algorithm was initialised from the same starting point, our plotted lines do not originate from the same coordinate since each algorithm required a different cost for the first iteration.

func	Alg	fmin	#f	Alg	fmin	#f	Alg	fmin	#f
A.1	BLS	2.38e-12	264	HLS	8.73e-12	259	BAS	8.92e-15	273
A.2	BLS	1.68e-06	6214	HLS	3.18e-08	270	BAS	5.41e-08	792
A.3	BLS	5.15e-10	830	HLS	3.24e-10	62	BAS	9.8e-10	107
A.4	BLS	4.4e-06	2787	HLS	2.35e-07	118	BAS	6.45e-07	375
A.5	BLS	6.98e-06	2444	HLS	2.36e-07	118	BAS	6.45e-07	375
A.6	BLS	6.31e-06	842	HLS	8.18e-07	67	BAS	5.59e-09	122
A.7	BLS	3.61e-06	844	HLS	6.82e-15	94	BAS	9.85e-07	305
A.9	BLS	1.82e-06	856	HLS	7.86e-09	625	BAS	0.000258	302
A.10	BLS	1.6e-08	1160	HLS	7.66e-09	650	BAS	0.000258	302
A.17	BLS	0.00444	32637	HLS	2.26e-08	8157	BAS	0.0551	67
A.18	BLS	2.13e-06	16752	HLS	1.53e-08	634	BAS	0.285	23994

Table 4.3.1 – A comparison of the performances of DGM equipped with both BLS and HLS against BASM. The test functions used here are all functions of two variables.

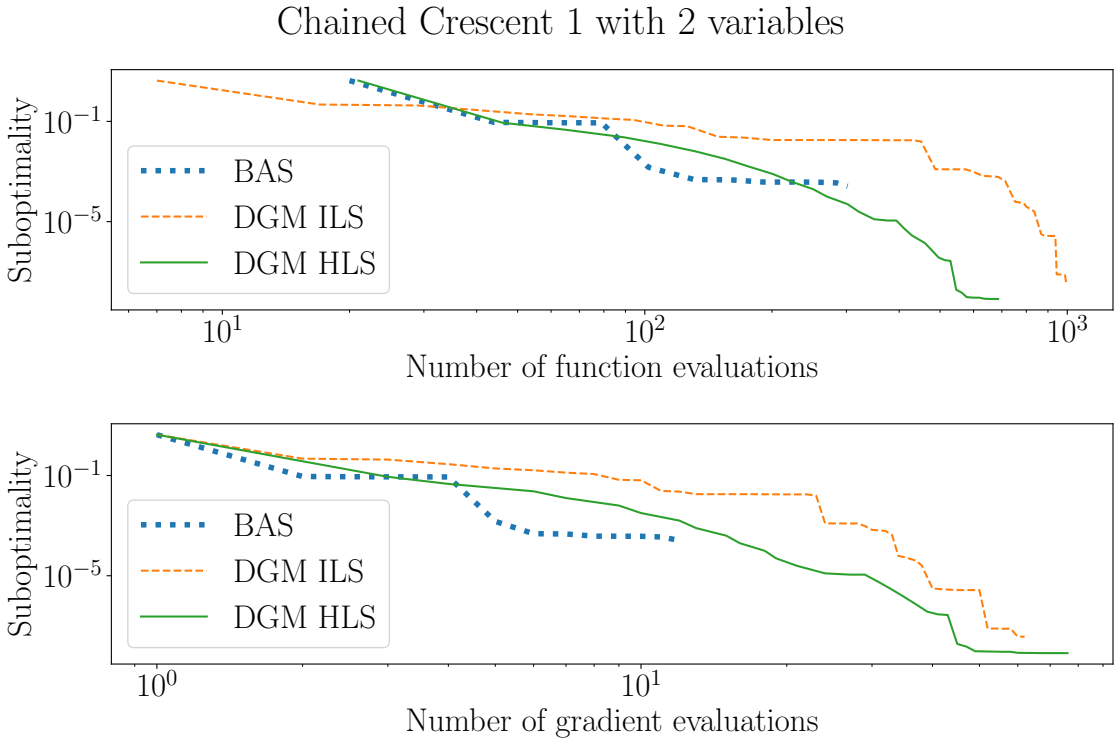

Figure 4.3.5 – We compare the DGM with both LS algorithms against the BASM on Problem A.9 with $n = 5$. Note that we only count the cost of function/gradient evaluations at the end of each iteration. Since the data for the first coordinates were harvested after the first (as opposed to zero) iteration, despite the fact that each algorithm was initialised from the same starting point, our plotted lines do not originate from the same coordinate since each algorithm required a different cost for the first iteration.

func	Alg	fmin	#f	Alg	fmin	#f	Alg	fmin	#f
A.1	BLS	3.24e-11	905	HLS	1.92e-11	553	BAS	1.85e-14	835
A.2	BLS	8.84e-06	25013	HLS	1.87e-05	2628	BAS	0.00142	836
A.3	BLS	0.00015	22920	HLS	0.00114	4826	BAS	0.000778	1038
A.4	BLS	1.91e-05	1711	HLS	2.24e-05	8042	BAS	0.018	5602
A.5	BLS	4.59e-09	20023	HLS	6.08e-09	1487	BAS	0.182	486
A.6	BLS	8.45e-06	909	HLS	3.14e-06	129	BAS	2.03e-08	131
A.7	BLS	1.16e-05	1327	HLS	9.51e-06	6898	BAS	0.0328	38405
A.9	BLS	7.89e-10	4686	HLS	1.31e-09	1184	BAS	7.7e-06	972
A.10	BLS	6.48e-06	2310	HLS	0.000331	16486	BAS	0.112	1885
A.17	BLS	0.584	11875	HLS	9.81e-05	3382	BAS	0.114	1579
A.18	BLS	0.0453	2327	HLS	0.0452	3344	BAS	0.0454	1821

Table 4.3.2 – A comparison of the performances of DGM equipped with both BLS and HLS against BASM. The test functions used here are all functions of five variables.

4.3.5 An interesting side effect of ELS inaccuracy

In our experience of experimenting with the BASM, we have observed a fascinating phenomenon occur once the cone of descent directions begins to narrow. In this section,

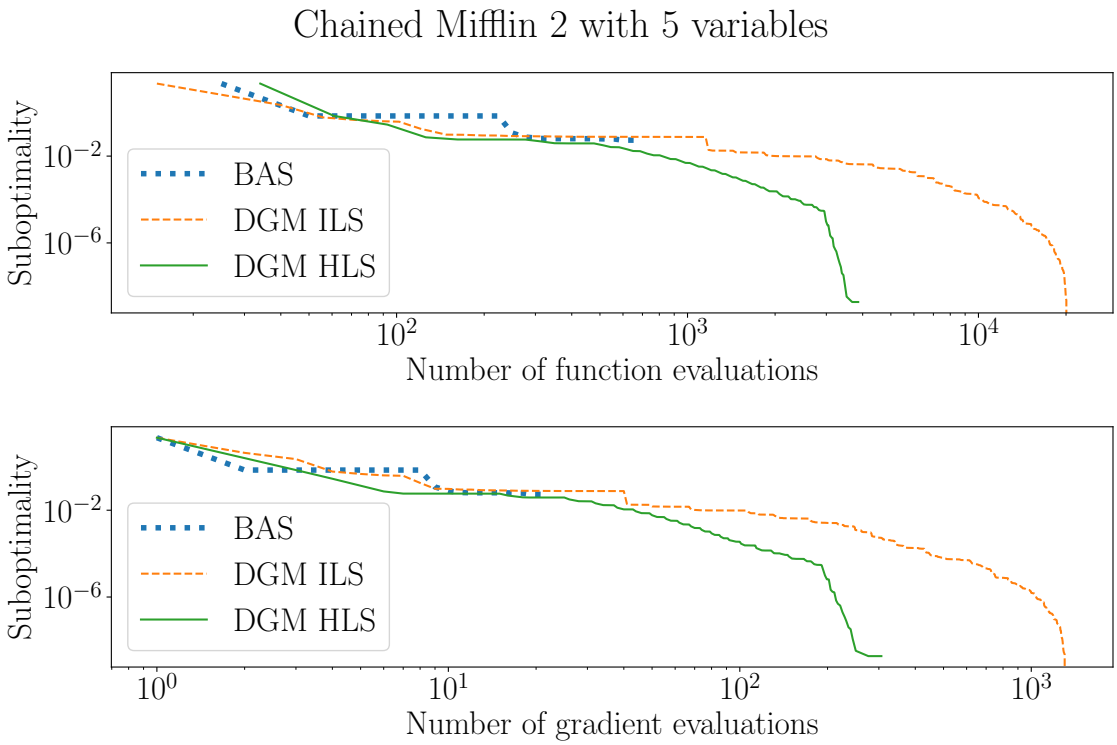

Figure 4.3.6 – We compare the DGM with both LS algorithms against the BASM on Problem A.8 with $n = 5$. Note that we only count the cost of function/gradient evaluations at the end of each iteration. Since the data for the first coordinates were harvested after the first (as opposed to zero) iteration, despite the fact that each algorithm was initialised from the same starting point, our plotted lines do not originate from the same coordinate since each algorithm required a different cost for the first iteration.

we describe our observations and suggest possible explanations for why it has occurred. Let it be understood that we have no rigorous justification for the content of this section, but we consider it sufficiently entertaining to mention anyway.

As the BASM runs, we have observed that $\cos \omega(\boldsymbol{p}_{k-1}, \boldsymbol{q}_k) \to -1$, where $\boldsymbol{q}_k = -\nabla\phi(\boldsymbol{w}_k)$ as it begins to converge. When this happens, it usually follows that $\|\boldsymbol{z}_k - \boldsymbol{x}_k\| \to 0$. Since the search direction produced by Algorithm 10 is a scaled version of $\boldsymbol{z}_k - \boldsymbol{x}_k$, this means that as $\|\boldsymbol{z}_k - \boldsymbol{x}_k\| \to 0$ then our descent direction is increasingly influenced by noise.

If we are working with infinite precision, then we wouldn't expect this to have any effect, but when working with machine precision, this means that any LS error dominates the approximation of $\boldsymbol{p}_k$. The result of this, as illustrated in Figure 4.3.8, is that $\boldsymbol{p}_k$ is a poor estimate of a search direction, resulting in an infinitesimal step length to $\boldsymbol{x}_{k+1}$.

At this point, even though $\boldsymbol{p}_k$ was a poor approximation for a descent direction,

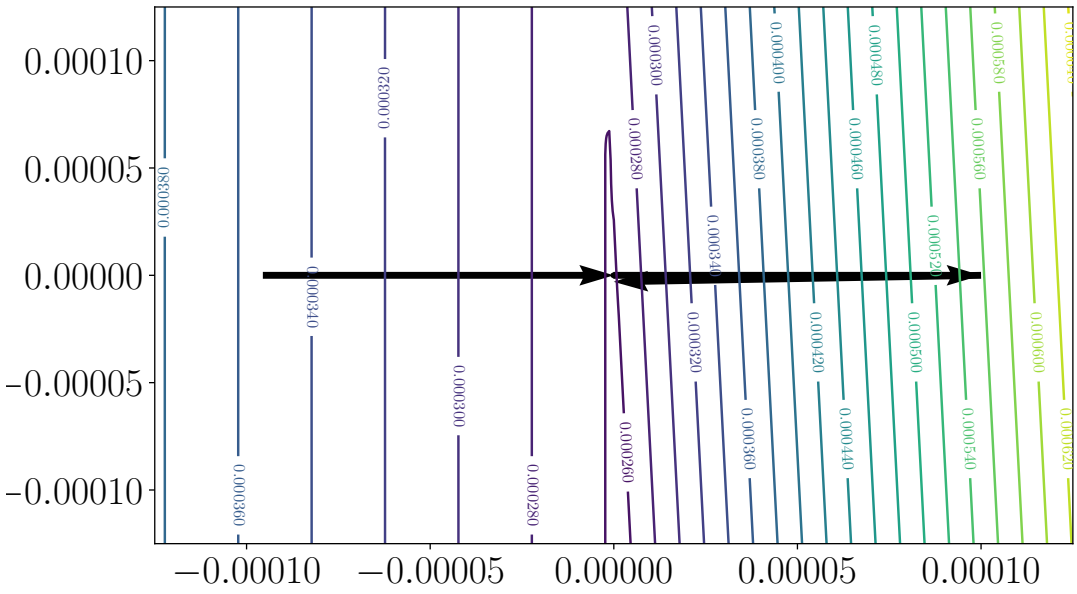

Figure 4.3.7 – When applied to the Problem A.8, we observe the BASM getting stuck due to $\langle \tilde{\boldsymbol{g}}_i | \tilde{\boldsymbol{g}}_j \rangle \to -1$.

unlike $\boldsymbol{p}_{k-1}$ we do not expect it to be nearly perpendicular to the kink. As a result, Algorithm 9 may be repeated, this time with $\cos\omega(\boldsymbol{p}_k, \boldsymbol{q}_{k+1}) \gg -1$. This allows $\boldsymbol{q}_{k+1}$ to be constructed significantly further away from $\boldsymbol{x}_{k+1}$, meaning that the LS errors become insignificant.

In our testing, we have noticed that as the BASM begins to converge (whether to a local minimum or not), we see this double iteration phenomenon occur more and more often. In fact, this general process can be repeated more than two times. The only limit on this approach occurs once $\boldsymbol{w}_k$ is within a distance ϵ of the kink. At this point, Assumption 4.4 will no longer apply and Algorithm 10 will terminate.

Remark. *It is possible that the phenomenon we have described in this section is merely an artefact of low dimensions, or a particular type of test function. We however have chosen to share it given that we had not planned for this phenomenon to occur, and in the experiments we conducted this phenomenon resulting in being able to construct descent directions within regions where we had not expected the BASM to be able to navigate.*

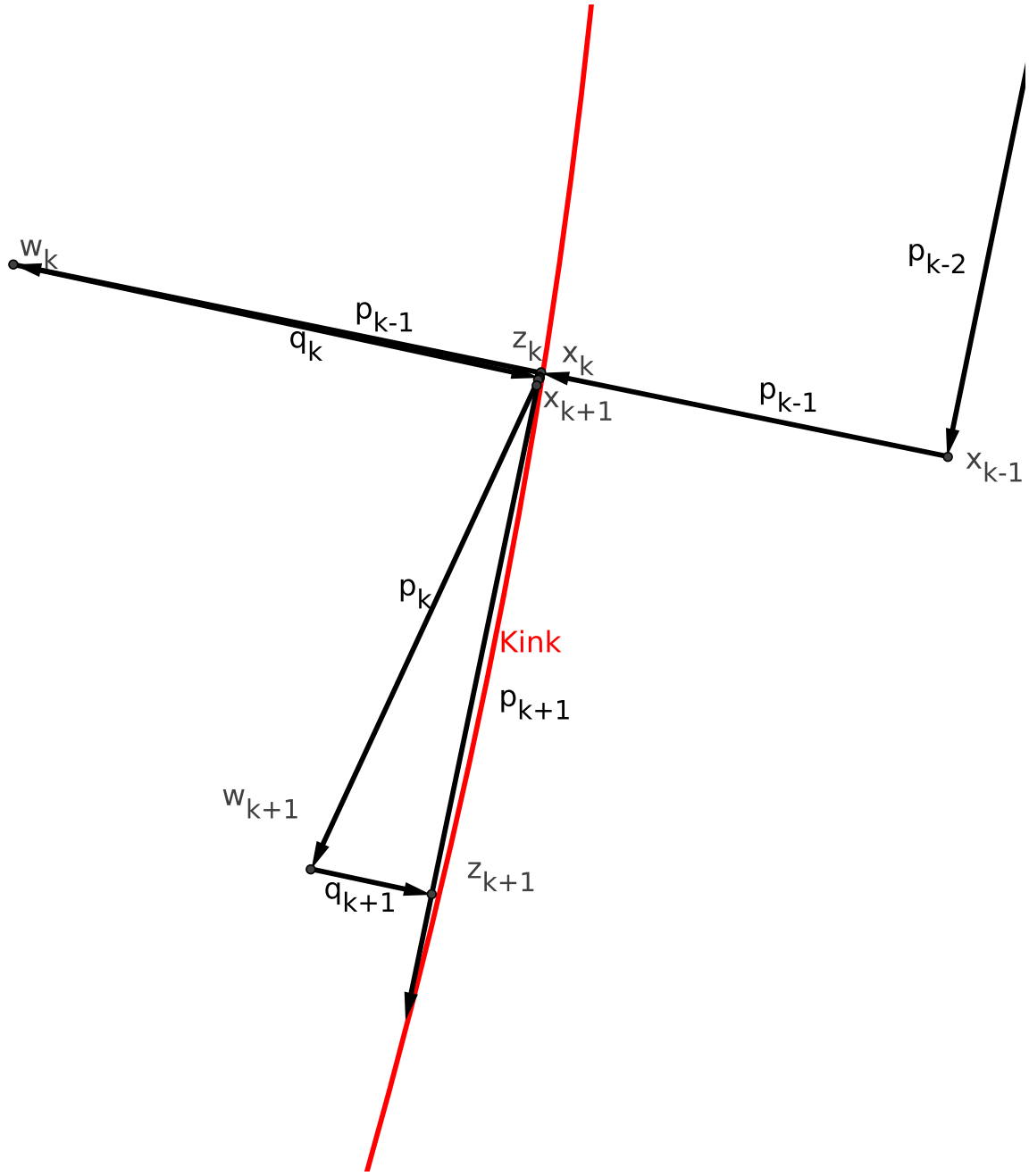

Figure 4.3.8 – Once $\langle \boldsymbol{p}_{k-1} | \boldsymbol{q}_k \rangle \to 0$, the LS errors begin to dominate the approximation of $\boldsymbol{p}_k$. At such times, we often observe a negligible step to $\boldsymbol{x}_{k+1}$, followed by a repeat of Algorithm 9 which yields a superior descent direction on the second occasion.

4.4 Conclusions and future work

In this chapter we have explored the possibility of using ELS for NSO. Our justification for this is that the ILS methods more commonly used are themselves computationally expensive, meaning that the traditional objection to ELS does not apply.

To mitigate the remaining weaknesses of ELS, we have designed the HLS to combine the desirable features of both ILS and ELS. In Section 4.1.2, we have inserted the HLS into the DGM, a particular generalised gradient descent method, and demonstrated that when equipped with HLS, the DGM consistently converges faster when applied to low dimensional problems.

Unfortunately, this trend does not continue when larger dimensions are taken into account. We can think of two possible reasons for this:

- The reasoning which justifies the HLS for small problems extends only to low dimensional problem and does not apply to higher dimensional problems.
- The step lengths produced by ELS are less useful (even ignoring cost) then those produced by BLS in the context of a method like DGM.

While acknowledging the possibility of the first options, we have chosen to suppose the second. Specifically, we have attempted to construct a new algorithm which doesn't merely employ HLS as a subroutine, but is specifically designed to make use of the additional information which HLS provides.

We have only managed to construct this new algorithm, the BASM, to be well defined in the context of low dimensional problems. In this setting, we have observed it to perform competitively with DGM when it does not get stuck prematurely. Therefore, while we yet cannot say whether the BASM might ultimately be a valid option for NSO, we can identify a series of questions concerning BASM, which if answered might make it more clear what its potential might be.

- Is there a computationally affordable construction which can produce a pair of points $(\boldsymbol{x}_k, \boldsymbol{z}_k)$ on a intersection of multiple kinks in the same way we have done for a single kink?

- Is it possible to use information from past iterations to approximate any second order information along the kink itself?
- What is a sensible termination condition to use for the BASM?
- Does the phenomenon observed in Section 4.3.5 only occur in lower dimensions or in general. If it occurs in general, might it be useful for convergence theory?

Chapter 5

Robust Univariate Optimisation

5.1 Existing methods

In this chapter, we derive an algorithm suitable for minimising a univariate piecewise smooth function f within the interval $[a, b]$. Since we will use this algorithm as the univariate optimiser in Algorithm 8, we need it to be first robust and then fast. We begin by exploring existing algorithms and assess whether a suitable one exists for solving the following problem:

Problem 5.1. *Given is a function $f : \mathbb{R} \to \mathbb{R}$ of the form $f = \max_{i=1,\dots,k} f_i(x)$ where $f_i : \mathbb{R} \to \mathbb{R}$ are smooth functions, an oracle for calling function values of $f(x)$, and an interval $[a, b]$. Design an optimisation method for finding a local minimiser of f in $[a, b]$ which makes use of only oracle calls for f.*

One category of solutions is that of global Lipschitzian methods such as of Evtushenko [69], Piyavskii [70], and Shubert [71], These methods use the Lipschitz constant to construct a function which equals the objective function at known points, but underestimates the objective everywhere else. The objective function is then evaluated at a sequence of points where a global minimum may possibly exist, thus either finding a new candidate for the global minimum, or else demonstrating that the the global minimum is not found near to the points evaluated. These algorithms would work very well for our purposes given that they are designed to find the a 1D global minimum

robustly. However, our aim is to find a 1D optimiser to use in for the HLS. In this context, we do not know if it is useful to find the 1D global minimum as opposed to any other local minimum. In this chapter, we chose to investigate methods for finding a local minimum with as few iterations as possible. Therefore, while we take inspiration from the approaches taken by these Lipschitzian methods (and underestimate the objective function away from known points in a similar way), we have decided not to use them.

Another class of algorithms which solves Problem 5.1 is that of the global optimisers which use interval analysis. The idea of these methods is to bound the objective function from below by bounding each individual term in the objective and them combining these bounds using interval arithmetic (for details see [72–75]). Like the Lipschitzian methods, algorithms from this class will find a global, not local minimiser, and will achieve this without needing a Lipschitz constant to be provided. These do not require the objective function to be known analytically, but do need to be able to decompose the objective function into a series of simple arithmetic operations $(+, -, \times, /)$. However, our objective function, which comes from 3dv1d, is a black box function, meaning that we cannot guarantee that all of the operations which go into the calculation of the objective function are acceptable (we suspect that some internal calculations are conducted iteratively). Hence, we have not attempted to use these methods for our research.

When choosing a local univariate solver, one faces a trade off between speed and robustness. The basic methods are Golden Section [76, 77] which converges Q-linearly for any continuous function, and interpolating methods [77, 78] which, when successful, converge super linearly. The latter's speed depends on their ability to construct a polynomial which approximates the objective function well locally. If the approximation does not fit the objective function well, then these algorithms may fail.

In practice, the most effective methods are bracketing interpolation hybrid methods such as Brent's Method [79, 80] and Hager's cubic method [81] which combine the speed of interpolation methods with the stability of Golden Section by making use of a fall back option when the interpolation is not working. While convergence is guaranteed

for such hybrid methods, they will still need their interpolations to match the objective function well if super linear convergence is to be obtained. Therefore, we can't expect them to converge quickly for an objective function of the form stated in Problem 5.1.

There exist other bracketing-interpolation hybrid methods for non differentiable functions of the form $f(x) = \max_i f_i(x)$ [64, 82]. However, these algorithms assume that we can compute each f_i (from which f is computed) separately. Therefore, these are unsuitable for Problem 5.1 given that the oracle is defined to only return the value of $f(x)$.

The only algorithm (to our knowledge) which is theoretically suited is the five point method of Mifflin and Strodiot [83], which from now on we refer to as the Mifflin-Strodiot algorithm (MSA). This algorithm is designed to converge rapidly to the local minimum x^* even when the objective function is non-differentiable at x^*.

The remainder of this chapter is structured as follows: In Section 5.2, we define the notion of a *bracket* rigorously and describe the existing methods which may be applicable to our problem, and demonstrate their use. Section 5.3 focuses on the derivation, and theory of a new univariate optimiser of our design, for which we present three variations. While it is the final of these variations which we suggest as an viable method to solve Problem 5.1, the other two are critical for its derivation.

5.2 Bracketing methods

Consider the problem of minimising a function of the form $f \in C^0[a, b]$, where $[a, b] \subset \mathbb{R}$. A common approach is to use a **bracketing** method. These are a class of 1D optimisation methods whose strength is their robustness.

Definition 5.1 (Bracket)**.** *Let f be a function, and $x^L, x^M, x^R \in \mathbb{R}$. We call the trio of points x^L, x^M, x^R a* bracket *of f if the following properties are satisfied:*

1. $x^L < x^M < x^R$,
2. $f(x^L) \geq f(x^M) \leq f(x^R)$.

Definition 5.2 (Set of Brackets)**.** *We define $\mathcal{B}_f$ to be the set of all brackets for the function f.*

Definition 5.3 (Bracket Length)**.** *Let $\mathcal{X} \in \mathcal{B}_f$. We define the length of the bracket $\mathcal{X}$ to be $b(\mathcal{X}) = x^R - x^L$.*

For convenience, we will frequently use the following notation (or generalisations of it) for the remainder of this document.

- Denote by $\mathcal{X} \in \mathcal{B}_f$ a bracket of f. We further define by x^L, x^M, x^R the individual elements of the bracket such that $x^L < x^M < x^R$.
- Denote by $b(\mathcal{X}) = x^R - x^L$, the length of the bracket.

Having defined a bracket, we now show their importance with the following result.

Lemma 15. *Let f be a continuous function and $(x^L, x^M, x^R) \in \mathcal{B}_f$ be a bracket of f. There exists a point $x^* \in (x^L, x^R)$ such that x^* is a minimum of f.*

Proof. Since f is continuous, we apply the Extreme Value Theorem [84, Theorem 4.4.3] to note that $\exists x^* \in [x^L, x^R]$ such that x^* is a minimum of f. As $(x^L, x^M, x^R) \in \mathcal{B}_f$ is a bracket of f, it follows that $f(x^L) \geq f(x^M) \leq f(x^R)$. If $x^* = \operatorname{argmin}_{x \in [x^L, x^R]} f(x) = f(x^M)$, then $x^* = x^M \in (x^L, x^R)$ is the point referred to by the proposition. Otherwise, if $x^* = \operatorname{argmin}_{x \in [x^L, x^R]} f(x) < f(x^M)$, x^* can't be equal to x^L or x_1^R since $f(x^L) = f(x^M) = f(x^R)$. Therefore, $x^* \in (x^L, x^R)$. □

Bracketing Algorithms are the set of algorithms which exploit Lemma 15 in order to guarantee global convergence to a local minimiser. In general they may be written in the form of Algorithm 12. Note that in order to function, they need two input functions, the step function $\sigma(\mathcal{X})$ and the update function $U(\tilde{x}, \mathcal{X})$.

In order for Algorithm 12 to converge, the functions σ and U must satisfy the

Algorithm 12: General Bracketing Method [85]

```
/* Initialisation:                                          */
input : f // Objective function
        1 σ // Step Function
        2 U // Update Function
        3 X_0 ∈ B_f // Initial Bracket
        4 ε > 0 // Tolerance
5 i = 0;
6 while b(X_i) > 2ε do
7     x̃_i = σ(X_i).
8     X_{i+1} = U(x̃_i, X_i).
9     i = i + 1;
  output: X_i.
```

following conditions.

$$\mathcal{X} \in \mathcal{B}_f \Rightarrow \sigma(\mathcal{X}) \in (x^L, x^R), \tag{5.1a}$$

$$\mathcal{X} \in \mathcal{B}_f \Rightarrow U(\sigma(\mathcal{X}), \mathcal{X}) \in \mathcal{B}_f, \tag{5.1b}$$

$$\exists C < 1 \text{ such that } b(U(\sigma(\mathcal{X}), \mathcal{X})) < Cb(\mathcal{X}). \tag{5.1c}$$

Once we define functions σ and U which satisfy Conditions 5.1a to 5.1c, then Algorithm 12 is guaranteed to converge to a bracket which contains a local minimum.

However, robustness is not enough as we also want an algorithm to be fast. Proving that a bracketing method converges quickly in general is not easy to do. In practice, we prove super-linear convergence only for certain special cases, while demonstrating numerical efficiency experimentally. The special case we use when proving super-linear convergence is that of *unimodal functions*.

Definition 5.4. *A function $f : [a, b] \to \mathbb{R}$ is called unimodal if there exists $\zeta \in (a, b)$ such that f is monotonically decreasing (increasing) on (a, ζ) and monotonically increasing (decreasing) on (ζ, b). If a function is not unimodal, then it is multi-modal.*

In the following sections, we present 3 particular Bracketing Methods: Golden Section, Brent's method and the Mifflin-Strodiot algorithm (MSA). Then in Section 5.2.4 we demonstrate both the capabilities and limitations of these algorithms, ultimately

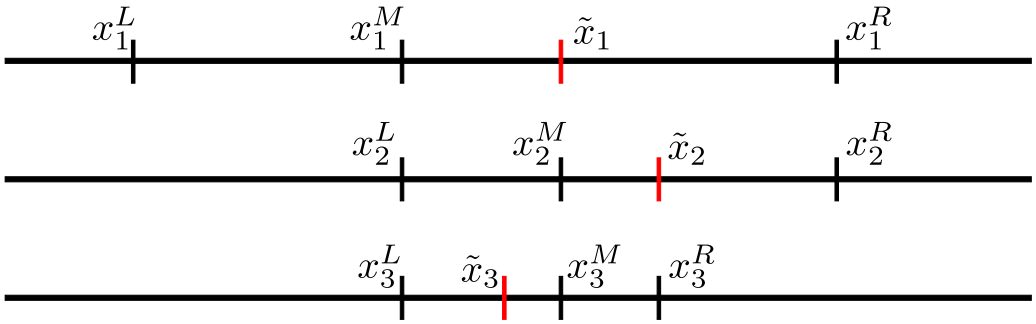

Figure 5.2.1 – Given a starting bracket $\mathcal{X}_1 = (x_1^L, x_1^M, x_1^R)$, we show three iterations of Golden Section. For each iteration i, the position of $\tilde{x}_i$ is highlighted in red.

motivating the construction of our new algorithm which we discuss in Section 5.3.

5.2.1 Golden Section

The Golden Section algorithm is a simple yet effective method which does not require any assumptions to be made about the objective function. It follows the outline of Algorithm 12 exactly using the particular step σ_G and update U_G functions corresponding to Golden Section as the input functions σ and U . In this section, we define explicitly σ_G and U_G, starting with U_G.

Suppose we have a bracket $\mathcal{X}$, and an additional point $\tilde{x}$ such that $\tilde{x} \in (x^L, x^R)$. How might we construct a new bracket contained within the previous one using $\tilde{x}$? This depends on two things: the size of $\tilde{x}$ compared to x^M and the size of $f(\tilde{x})$ compared to $f(x^M)$. There are four cases to consider, which we write out explicitly below.

$$U_G(\tilde{x}, \mathcal{X}) = \begin{cases} (x^L, \tilde{x}, x^M) & \text{if } \tilde{x} < x^M \text{ and } f(\tilde{x}) < f(x^M) \\ (\tilde{x}, x^M, x^R) & \text{if } \tilde{x} < x^M \text{ and } f(\tilde{x}) > f(x^M) \\ (x^M, \tilde{x}, x^R) & \text{if } \tilde{x} > x^M \text{ and } f(\tilde{x}) < f(x^M) \\ (x^L, x^M, \tilde{x}) & \text{if } \tilde{x} > x^M \text{ and } f(\tilde{x}) > f(x^M) \end{cases} . \tag{5.2}$$

Remark. *Note that U as defined in Equation* (5.2) *satisfies Conditions 5.1c and 5.1b provided that $\tilde{x}$ satisfies Condition 5.1a.*

Next we define σ_G to be:

$$\sigma_G(\mathcal{X}) := \begin{cases} \text{if } x^M < \frac{1}{2}(x^L + x^R) & x^M + \phi(x^R - x^M) \\ \text{else} & x^M + \phi(x^L - x^M) \end{cases}, \tag{5.3}$$

where $\phi = \frac{1}{2}(3 - \sqrt{5})$.

What properties does this choice of σ_G have? First note that σ_G will always select a point in the larger of the following intervals: (x^L, x^M) or (x^M, x^R). Next consider how this choice of ϕ affects the size of the new bracket.

Assume without loss of generality that $x^M < \frac{1}{2}(x^L + x^R)$. Depending on the sign of $f(x^M) - f(\sigma_G(\mathcal{X}))$, the length of the new bracket: $b(U_G(\sigma_G(\mathcal{X}), \mathcal{X}))$ will have one of two values: $x^R - x^M$ or $\sigma_G(\mathcal{X}) - x^L$.

Define w to be the ratio $(x^M - x^L)/(x^R - x^L)$. Given this, we compute the ratio of the length of the new bracket to the length of the old one for both cases. These are

$$\frac{x^R - x^M}{x^R - x^L} = 1 - \frac{x^M - x^L}{x^R - x^L} = 1 - w,$$

and

$$\begin{aligned} \frac{\sigma_G(\mathcal{X}) - x^L}{x^R - x^L} &= \frac{x^M + \phi(x^R - x^M) - x^L}{x^R - x^L}, \\ &= \frac{\phi(x^R - x^L) + (1 - \phi)(x^M - x^L)}{x^R - x^L} = \phi + (1 - \phi)w. \end{aligned}$$

For Golden Section, we choose $w = \phi$. This means that the relative position of $\sigma_G(\mathcal{X})$ with respect to x^R and x^M will be the same as that of x^M with respect to x^R and x^L, that is:

$$\frac{\sigma_G(\mathcal{X}) - x^M}{x^R - x^M} = \frac{x^M - x^L}{x^R - x^L}.$$

Having chosen w this way, the ratio of the length of the new bracket to that of the old will either be $1 - \phi$ or $\phi(2 - \phi)$. The value $\phi = \frac{1}{2}(3 - \sqrt{5})$ corresponds to the unique number such that $1 - \phi = \phi(2 - \phi)$ and $\phi < 1$.

In short, this choice of ϕ ensures that the next bracket is smaller than the pre-

vious one. Furthermore, the reduction of the bracket does not depend on whether $(x^L, x^M, \sigma_G(\mathcal{X}))$ or $(x^M, \sigma_G(\mathcal{X}), x^R)$ is the next bracket chosen. Finally the relative spacing of the bracket points is preserved.

As a result of this, Golden Section has the property that $b(\mathcal{X}_{i+1})/b(\mathcal{X}_i) = 1 - \phi$ for every iteration $i \geq 0$ (assuming that $(x_0^M - x_0^L)/(x_0^R - x_0^L) = \phi$).

Remark. *If $(x_0^M - x_0^L)/(x_0^R - x_0^L) \neq \phi$ then one may observe that $b(\mathcal{X}_{i+1})/b(\mathcal{X}_i) \neq 1 - \phi$ for a few iterations. In practice however, one soon encounters an iteration i_0 such that $b(\mathcal{X}_{i+1})/b(\mathcal{X}_i) = 1 - \phi$ for every $i \geq i_0$.*

Not only have we defined the functions σ_G and U_G for Golden Section, but we have also derived its convergence properties in the process. Specifically, we have shown that $b(\mathcal{X}_{i+1})/b(\mathcal{X}_i) = 1 - \phi = \frac{1}{2}(-1 + \sqrt{5}) \approx 0.618$ for all iterations i. In other words Golden Section converges linearly with a factor ≈ 0.618.

5.2.2 Brent's Method

While Golden Section is both robust and generally applicable, we often wish to optimise a function about which more is known. If $f \in C^k[a, b]$, with $k \geq 1$, then we may take advantage of the function's curvature properties in order to achieve faster convergence. One method which does this is Brent's Method. In this section, we present a simplified version of Brent's Method. It is simplified in the sense that we omit the features which ensure numerical stability, instead focusing on how the method would operate given exact arithmetic. A practical version of Brent's method, as translated from [85], is given in Appendix B.1 Algorithm 13.

Brent's method [79, 85] combines ideas from Bracketing Methods, and Interpolation Methods. It makes use of a bracket to ensure convergence while simultaneously using interpolation to improve speed. While it mostly conforms to the form of Algorithm 12, some generalisations are required. We label the step function and update function used for Brent's Method as σ_B and U_B.

In order to proceed, we extend our bracket notation from earlier. Now we define:

$$\mathcal{X} = \Big(\; x^L, x^M, x^R, y, z, \delta_1, \delta_2 \; \Big).$$

We need $\mathcal{X}$ to contain these four additional values for sake of the step function σ_B. When selecting our next test point, we interpolate the 3 values of x found thus far with the smallest values of f. The best value will by definition be x^M. However the other two need not be x^L or x^R. Therefore, we update y and z such that y_i, the value of y during the i-th iteration of Brent's method, is equal to the second best value of x found. Meanwhile z is defined as the previous value of y.

Another feature of Brent's method is that it keeps track of the convergence rate. This way it can revert to Golden Section if gets stuck, or converges too slowly. We define δ_1 to be the previous value of $\tilde{x} - x^M$, and δ_2 to be the previous value of δ_1.

With the definition of $\mathcal{X}$ extended, we now define U_B:

$$U_B(\tilde{x}, \mathcal{X}) \coloneqq \begin{cases} \text{if } f(\tilde{x}) < f(x^M) & (U_G(\tilde{x}, (x^L, x^M, x^R)), x^M, y, \tilde{x} - x^M, \delta_1) \\ \text{else if } f(\tilde{x}) < f(y) & (U_G(\tilde{x}, (x^L, x^M, x^R)), \tilde{x}, y, \tilde{x} - x^M, \delta_1) \\ \text{else} & (U_G(\tilde{x}, (x^L, x^M, x^R)), y, z, \tilde{x} - x^M, \delta_1) \end{cases} \tag{5.4}$$

Note that Equation (5.4) extends Equation (5.2) only in the sense of bookkeeping. It merely adds the feature of updating y, z, δ_1 and δ_2 appropriately.

Next we discuss how σ_B works. For each iteration, Brent's method determines two candidates for $\tilde{x}$, and selects one of them depending on certain conditions. One of the candidates is simply σ_G (see Equation (5.3)), the same step used in Golden Section.

The alternative is the point $\tilde{x}$ which satisfies $q'(\tilde{x}) = 0$ where q is the quadratic polynomial which interpolates x^M, y and z. We illustrate this option in Figure 5.2.2. While this process can be proven to converge super-linearly under ideal conditions (we state the theorem from [80] in Appendix B.1.2, Theorem 30), there exist circumstances under which it fails to converge at all. This is why the golden section step σ_G is

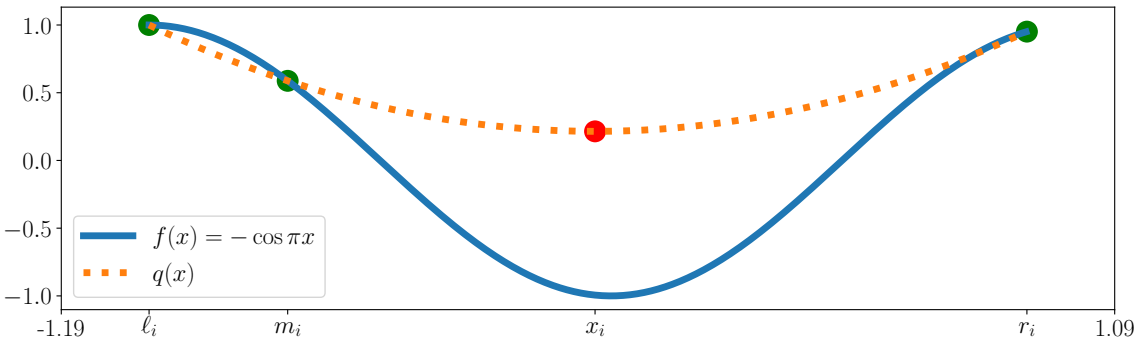

Figure 5.2.2 – Given the three points in green, Brent's method constructs the quadratic $q(x)$. Since the unique minimum of this quadratic is between x_i^L and x_i^R, we accept this step and use $\tilde{x}_i$ as the next test point.

sometimes used, both to speed up convergence, and prevent the algorithm from stalling.

Formally we state:

$$\sigma_B(\mathcal{X}) := \begin{cases} \text{if } \arg_x(q'(x) = 0) \notin (x^L, x^R) & \sigma_G(x^L, x^M, x^R) \\ \text{else if } |\arg_x(q'(x) = 0) - x^M| > \frac{1}{2}|\delta_2| & \sigma_G(x^L, x^M, x^R) \\ \text{else} & \arg_x(q'(x) = 0) \end{cases} \tag{5.5}$$

In short, there are two circumstances under which the local minimum of the quadratic q will be rejected. The first is when the local minimum lies outside the bracket (or doesn't exist at all). In this case, it is clear that σ_G is a more sensible option. The second case in which we select σ_G is when the alternative option does not appear to be converging fast enough.

As a consequence of this, Brent's method usually converges super-linearly and hence finds a local minimum far faster than golden section. In the event that Brent's method fails to converge super-linearly, it in practice performs no worse than Golden Section.

5.2.3 The Mifflin-Strodiot algorithm

To our knowledge, the five point method of Mifflin and Stodiot [83] is the only existing univariate optimiser with is both suitable for and tailored to Problem 5.1. Like Brent's method, the Mifflin-Strodiot algorithm (MSA) approximates f using interpolation. However, the MSA makes use of a larger pool of points to construct more than one

interpolation in anticipation of the possibility of a NS local optimum. In this section, we define the Mifflin-Strodiot algorithm using a consistent notation to that which we have employed thus far.

As stated, the MSA employs a larger set of points than merely a bracket. However, if we were to replace a bracket with this extended set of points, then the MSA would match the form of Algorithm 12. This motivates the following definition.

Definition 5.5 (Mifflin-Strodiot Extended Bracket)**.** *Given the function f, we call $\mathcal{X} \in \mathbb{R}^5$ a Mifflin-Strodiot extended bracket written in the following form*

$$\mathcal{X} = \begin{pmatrix} x_2^L & x_1^L & x^M & x_1^R & x_2^R \end{pmatrix}^T, \tag{5.6}$$

if the following conditions apply:

$$x_2^L < x_1^L < x^M < x_1^R < x_2^R \tag{5.7a}$$

$$f(x_1^L) \geq f(x^M) \leq f(x_1^R). \tag{5.7b}$$

Having extended the bracket, we now define the update function U_M, which is the natural extension of U_G to MS extended brackets.

$$U_M(\tilde{x}; \mathcal{X}, \alpha) := \begin{cases} \begin{pmatrix} x_2^L & x_1^L & \tilde{x} & x^M & x_1^R \end{pmatrix}^T & \text{if} \quad \tilde{x} < x^M \text{ and } f(\tilde{x}) < f(x^M) \\ \begin{pmatrix} x_1^L & x^M & \tilde{x} & x_1^R & x_2^R \end{pmatrix}^T & \text{if} \quad \tilde{x} > x^M \text{ and } f(\tilde{x}) < f(x^M) \\ \begin{pmatrix} x_2^L & x_1^L & x^M & \tilde{x} & x_1^R \end{pmatrix}^T & \text{if} \quad \tilde{x} > x^M \text{ and } f(\tilde{x}) > f(x^M) \\ \begin{pmatrix} x_1^L & \tilde{x} & x^M & x_1^R & x_2^R \end{pmatrix}^T & \text{if} \quad \tilde{x} < x^M \text{ and } f(\tilde{x}) > f(x^M) \end{cases}, \tag{5.8}$$

Having specified U_M, all that remains to define the MSA is to write out σ_M. Intuitively, the MSA will both construct the pair of linear functions which interpolate x_2^L, x_1^L and x_1^R, x_2^R respectively, in addition to the quadratic which interpolates x_1^L, x^M, x_1^R. The MSA will first attempt to select the point which minimises the maximum of these linear functions, but will resort to the minimum of the quadratic function if needed.

In order to be more rigorous, we must introduce a considerable amount of notation.

We begin by defining the following pair of polynomials which interpolate the points on the left and right respectively:

$$q_L(x) = \tfrac{1}{2}a_L x^2 + b_L x + c_L, \quad \text{and} \quad q_R(x) = \tfrac{1}{2}a_R x^2 + b_R x + c_R \tag{5.9}$$

Next we have:

$$\begin{aligned}
g_L &= q_L'(x^M) = a_L x^M + b_L, & g_R &= q_R'(x^M) = a_R x^M + b_R, \\
\hat{g}_L &= \frac{f(x_1^L) - f(x_2^L)}{x_1^L - x_2^L}, & \hat{g}_R &= \frac{f(x_1^R) - f(x_2^R)}{x_1^R - x_2^R}, \\
e_L &= f(x^M) - f(x_1^L) - \hat{g}_L(x^M - x_1^L), & e_R &= f(x^M) - f(x_1^R) - \hat{g}_R(x^M - x_1^R).
\end{aligned}$$

Using this notation, we define S^*, the new trial point to evaluate before modification:

$$S^* = \begin{cases} x^M + (e_R - e_L)/(\hat{g}_R - \hat{g}_L) & \text{if } \hat{g}_R > 0 \text{ and } \hat{g}_L < 0 \\ Q^* & \text{otherwise} \end{cases}, \tag{5.10}$$

where Q^* is the minimiser of the polynomial which interpolates x_1^L, x^M, x_1^R. In short, before applying modifications, the MSA will check if the line interpolating x_1^L and x_2^L has negative slope and the line interpolating x_1^R and x_2^R has positive slope. If so, then the default trial point is the intersection of these lines. Otherwise, the minimum of the polynomial which interpolates x_1^L, x^M, x_1^R is selected.

Next we apply modifications. This phase involves constructing an interval which we'd like our trial point to be contained in, and by doing so protect ourselves from extreme circumstances. We begin by computing Q_L and Q_R, which are defined to be the local minima of q_L and q_R respectively which are constrained to the interval $[x_1^L, x_1^R]$. If Q_L and Q_R are close together, this implies that the quadratic functions q_L and q_R agree with each other, which in turn suggests that f may be smooth. Ultimately we modify S^* by projecting it onto $[Q_R, Q_L]$. This has the effect that when f is NS, the piecewise model is used, but when f is smooth, this is overridden.

However, there is still more to be done, we define:

$$P_L = \begin{cases} x^M + e_R/(\hat{g}_R - g_L) & \text{if } Q_L > x^M,\ Q_R \geq x^M, \text{ and } e_R < (\hat{g}_R - g_L)(Q_L - x^M) \\ Q_L & \text{otherwise} \end{cases}, \tag{5.11}$$

and

$$P_R = \begin{cases} x^M - e_L/(g_R - \hat{g}_L) & \text{if } Q_L \leq x^M,\ Q_R < x^M, \text{ and } e_L < (g_R - \hat{g}_L)(x^M - Q_R) \\ Q_R & \text{otherwise} \end{cases}. \tag{5.12}$$

Finally we define $L = \min(P_L, P_R)$, $U = \max(P_L, P_R)$, and

$$\sigma_M(\mathcal{X}) := proj(S^*, [L, U]), \tag{5.13}$$

where $proj(x, S)$ refers to the projection of the point x onto the set S.

Remark. *There exists an additional step in [83] for computing the next test point. This step is a numerical stabilising step which ensures that no two points in the bracket are too close together. We have omitted this step for consistency, as we have not discussed such details previously.*

5.2.4 Are these methods suitable for Problem 5.1?

With Golden Section, Brent's method and the Mifflin-Strodiot algorithm defined, we now demonstrate their use. Before we do, we identify the class of functions which these algorithms are designed for.

Golden Section is perfectly suitable for any continuous function, while Brent's is most applicable to twice differentiable unimodal functions. Meanwhile, the MSA is theoretically suited to non-smooth functions. In this section, we illustrate the performance of these three algorithms on a selection of test functions which are chosen to demonstrate both their capabilities and weaknesses. We begin by demonstrating all three algorithms on a smooth function, for which they all apply, and show the outcome in Figure 5.2.4.

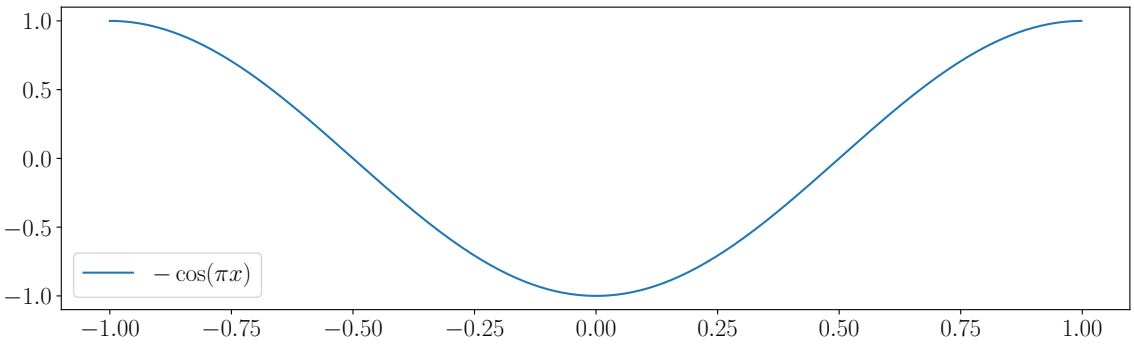

Figure 5.2.3 – A plot of the objective function $f(x) = -\cos \pi x$.

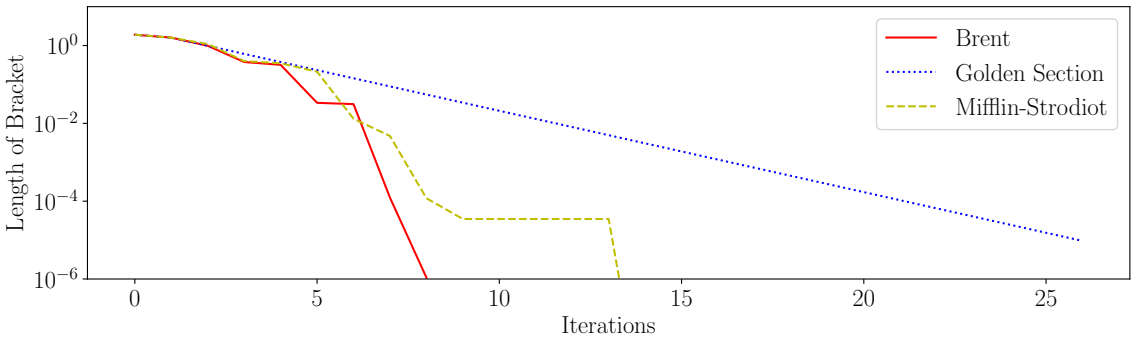

Figure 5.2.4 – We plot the convergence of Brent's method, Golden Section and the MSA when applied to the same objective function $f(x) = -\cos \pi x$, with the same starting bracket.

As we see in Figure 5.2.4, Golden Sections converges linearly and is clearly the slowest option. Brent's method is the fastest, which is to be expected given it is designed for such purposes. Meanwhile, the MSA performs much faster that Golden Section, but notably slower than Brent's method.

However, the goal we are working towards is finding a univariate optimiser for NS functions. In Figure 5.2.6, we run all three algorithms on a function which is NS, but convex.

What we note from Figure 5.2.6 is that Brent's method performs pretty much exactly the same as Golden Section. This implies that it requires its fall back option at literally every iteration. Meanwhile, the MSA reasonably quickly here, demonstrating its capabilities for such functions.

It might be tempting to conclude that the MSA is exactly what we need for Algo-

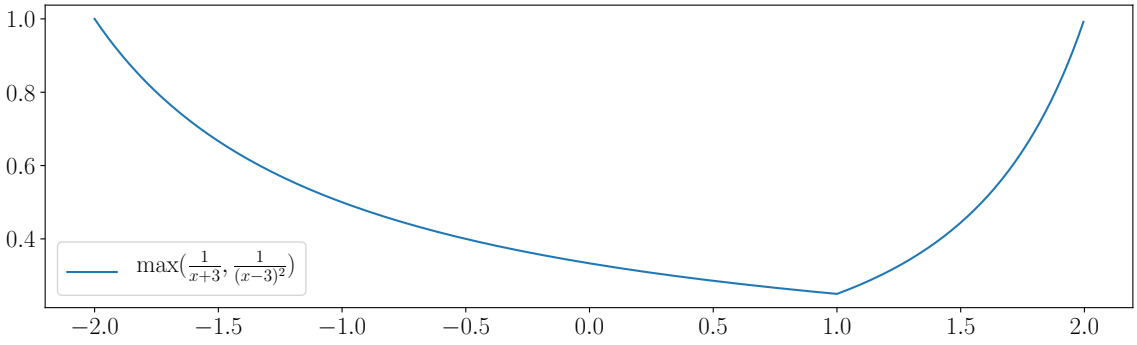

Figure 5.2.5 – A plot of the objective function $f(x) = \max((x+3)^{-1}, (x-3)^{-2})$.

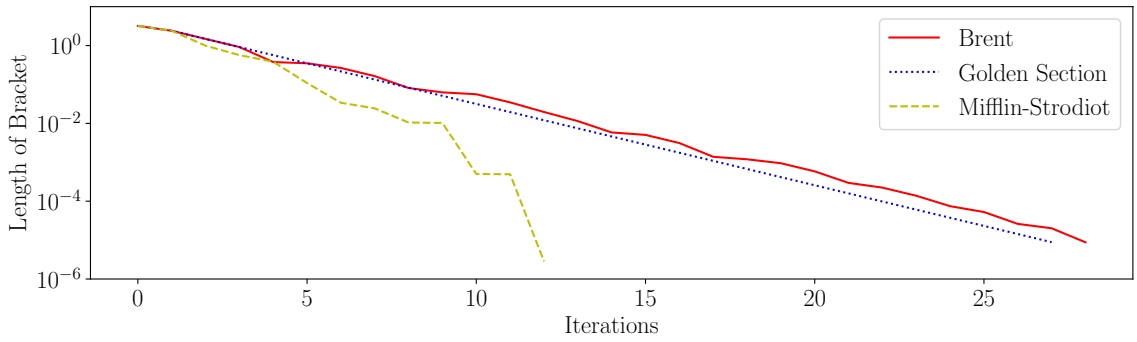

Figure 5.2.6 – We plot the convergence of Brent's method, Golden Section and the MSA when applied to the same objective function $f(x) = \max((x+3)^{-1}, (x-3)^{-2})$, with the same starting bracket.

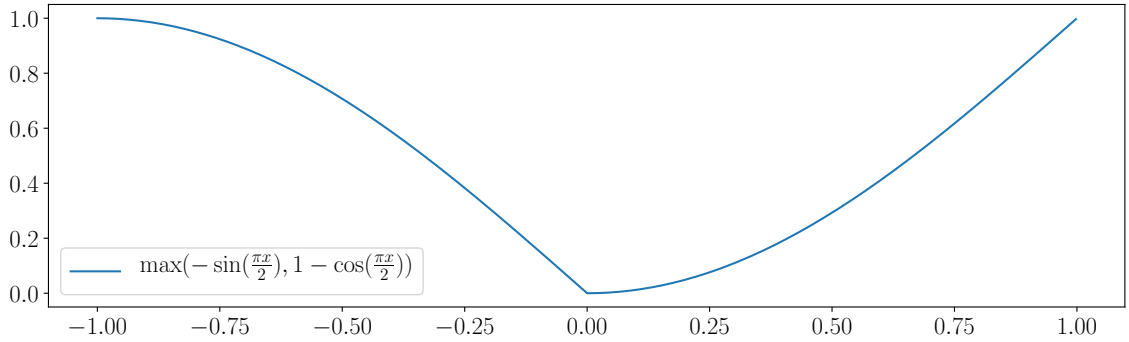

Figure 5.2.7 – A plot of the objective function $f(x) = \max\left(-\sin\frac{1}{2}\pi x, 1-\cos\frac{1}{2}\pi x\right)$.

rithm 8. However, in Figure 5.2.8 we show one additional example, this time running all three algorithms on a function which is NS and non-convex.

The main point to note in Figure 5.2.8 is that the MSA fails utterly. Not only does it converge slowly, but it makes no apparent convergence at all within 60 iterations.

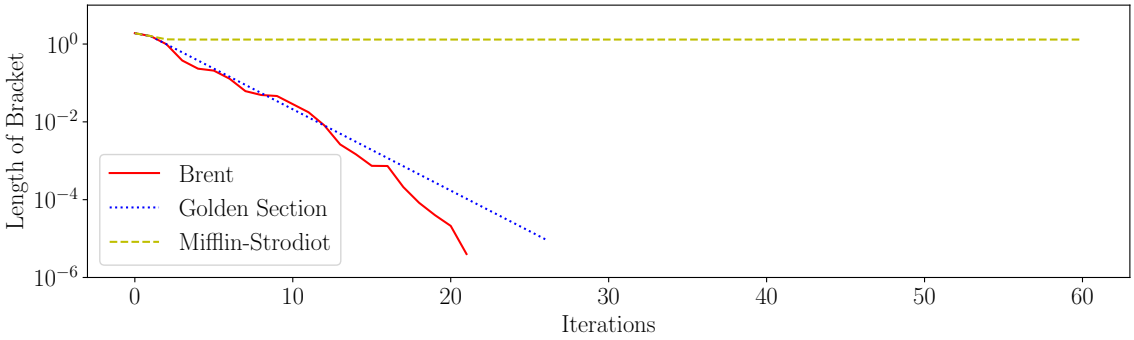

Figure 5.2.8 – We plot the convergence of Brent's method, Golden Section and the MSA when applied to the same objective function $f(x) = \max\left(-\sin\frac{1}{2}\pi x, 1-\cos\frac{1}{2}\pi x\right)$, with the same starting bracket.

Meanwhile Brent's method performs only slightly better than Golden Section.

In short, Golden Section always converges linearly regardless of the function. Brent's method converges quickly for smooth functions, but slowly yet robustly for NS functions. Finally, while the MSA converges quickly for smooth functions and some NS functions, it is not robust given that certain functions such as the one plotted in Figure 5.2.8 cause it to stall effectively indefinitely. Motivated by this, in Section 5.3 we derive a new algorithm designed to be both robust, and fast in as many contexts as possible.

5.3 The underestimating polynomial method

The idea of bracketing methods such as Brent's method [79, 80], and Hager's Cubic method [81] is that the polynomial which interpolates the bracket points approximates the objective function well locally. Therefore the local minimum of the quadratic is a sensible location to evaluate next. The error of this approximation can be quantified and bounded, yielding super-linear convergence guarantees [80, Theorem 4.1]. While this analysis works well for smooth functions, our problem is a piecewise smooth (recall Definition 4.1), black box function.

The theory behind algorithms such as Brent's method collapses due to the existence of kinks, because interpolating across a kink has no meaning and yields no meaningful error bound. Therefore, if the local minimum of a function f is a kink, we expect

algorithms like Brent's method to converge to it slowly. The premise of the UPM is inspired by the approach used in the Mifflin-Strodiot method [86]. We approximate the objective function with two polynomials, one on each side of the bracket. If the local minimum is a kink, then the combination of these polynomials may be valid approximations and useful for locating the minimum. To construct these polynomials, we need more than the three points contained in a bracket and therefore extend the definition of a bracket to include seven points:

Definition 5.6 (Extended Bracket)**.** *Given the function f, we call $\mathcal{X} \in \mathbb{R}^7$ an extended bracket written in the following form*

$$\mathcal{X} = \begin{pmatrix} x_3^L & x_2^L & x_1^L & x^M & x_1^R & x_2^R & x_3^R \end{pmatrix}^T, \tag{5.14}$$

if the following conditions apply:

$$x_3^L < x_2^L < x_1^L < x^M < x_1^R < x_2^R < x_3^R \tag{5.15a}$$

$$f(x_1^L) \geq f(x^M) \leq f(x_1^R). \tag{5.15b}$$

Remark. *If $\mathcal{X}$ is an extended bracket, then (x_1^L, x^M, x_1^R) form a bracket.*

For expressing an extended bracket's size, we define the following functions.

Definition 5.7 (Extended Bracket Length)**.** *Let $\mathcal{X}$ be an extended bracket of f. We define: $diam(\mathcal{X}) = diam(Conv\{\mathcal{X}\})$, and $b(\mathcal{X}) = x_1^R - x_1^L$.*

Remark. *Definition 5.7 is the natural extension of the function $b(\mathcal{X})$ from Section 5.2 where $\mathcal{X}$ is merely a bracket as opposed to an extended bracket.*

The subsections below are structured in the following way. In Section 5.3.1, we define the core method and in the process introduce the notation we use in later sections. Afterwards in Sections 5.3.2 to 5.3.4 we explore three variations of this core underestimating polynomial method, the order of which is motivated by the derivation process.

5.3.1 UPM core method

The UPM mostly conforms to the form of Algorithm 12. The main difference, apart from replacing the bracket with an extended bracket, is the fact that the UPM also depends on an input parameter α. This parameter is used by the step function σ when constructing the next point to evaluate. The three variations of the UPM presented in this chapter differ in how they treat α.

The static underestimating polynomial method (SUPM) requires an initial value of α to be supplied by the user, which is then kept constant for the duration of the algorithm. For the remainder of this section, we denote the step function σ and update function U which apply to the SUPM by σ_S and U_S respectively.

Given the objective function f, our strategy is to use the points x_1^L, x_2^L, x_3^L and x_1^R, x_2^R, x_3^R to construct the model functions $q^L(x; \mathcal{X}, \alpha)$, and $q^R(x; \mathcal{X}, \alpha)$ which approximate f and f to the left and right of x^M respectively. We will employ Newtons Divided Difference notation which we summarize below.

Definition 5.8. *Given $a, b, c \in \mathbb{R}$, and $f : \mathbb{R} \to \mathbb{R}$, the 1st and 2nd divided differences are defined by:*

$$f[a,b] = \frac{f(a) - f(b)}{a - b}, \qquad \text{and} \qquad f[a,b,c] = \frac{f[a,b] - f[a,c]}{b - c}.$$

The reason we use this notation is that the 1st and 2nd divided differences are natural approximations for the 1st and 2nd derivatives of f respectively. We will state this result in a later section when more rigour is needed (see Theorem 17).

Now we define our model functions q^L and q^R as the following:

$$q^k(x; \mathcal{X}, \alpha) = f(x_1^k) + f[x_1^k, x_2^k](x - x_1^k) + (f[x_1^k, x_2^k, x_3^k] - \alpha h(\mathcal{X}))(x - x_1^k)(x - x_2^k), \quad (5.16)$$

where $k \in \{L, R\}$, $\alpha > 0$ is a constant and $h(\mathcal{X})$ is a scaling function with the property that $h(\mathcal{X}) \to 0$ as $diam(\mathcal{X}) \to 0$. Note that q^k has the form of the 2nd order Newton Interpolating Polynomial combined with the adjustment term $\alpha h(\mathcal{X})$. Finally we express

σ_S in terms of q^L and q^R:

$$\sigma_S(\mathcal{X};\alpha) = \underset{x\in[x_1^L,x_1^R]}{\operatorname{argmin}} \max(q^L(x;\mathcal{X},\alpha), q^R(x;\mathcal{X},\alpha)). \tag{5.17}$$

Remark. *The model functions q^L and q^R are designed to underestimate f to the left and right of x^M respectively, within $[x_1^L, x_1^R]$. We will show how we achieve this in Lemma 18.*

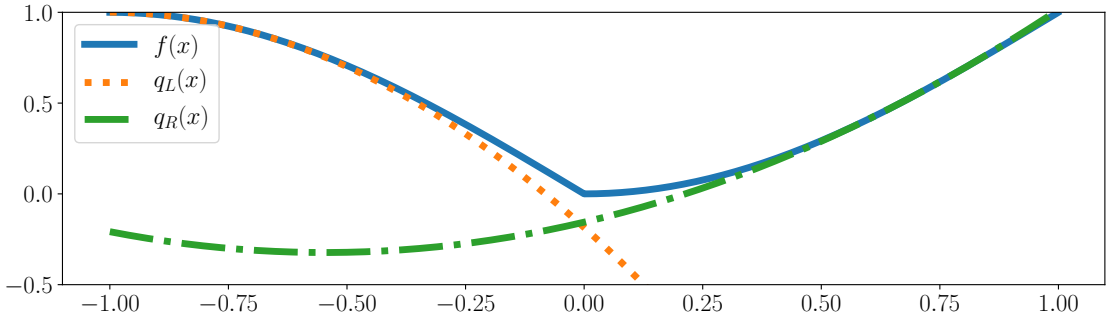

Figure 5.3.1 – We plot the objective function in addition to the quadratic function approximating the left and right. These two quadratics intersect at $x = 0.005\ldots$.

Example 5.1. *Consider the function $f(x) = \max\left(\sin\left(\frac{1}{2}x\pi\right), 1-\cos\left(\frac{1}{2}x\pi\right)\right)$. This function is piecewise smooth and unimodal in the interval $[-1,1]$ with a local minimum at 0. At this local minimum, f is not differentiable.*

Now suppose that $\{x_i^L\}_{i=1}^3 = \{-0.75, -0.9, -1\}$ and $\{x_i^R\}_{i=1}^3 = \{0.6,\ 0.8,\ 0.95\}$. We construct q^L and q^R by interpolating $\{(x_i^k, f(x_i^k))\}_{i=1}^3$ for $k = L, R$. These two quadratics intersect at $x \approx 0.005$ as shown in Figure 5.3.1. This value is taken to be the next point at which we evaluate f.

All that remains is for us to define the function U_S. With each iteration there are exactly 4 different ways in which we might update $\mathcal{X}_n$. The one we choose depends on

the values of $\tilde{x} = \sigma_S(\mathcal{X}; \alpha)$ and $f(\tilde{x})$ compared to $f(x^M)$. In particular we define:

$$U_S(\tilde{x}; \mathcal{X}, \alpha) := \begin{cases} U_1(\tilde{x}; \mathcal{X}) & \text{if} \quad \tilde{x} < x^M \text{ and } f(\tilde{x}) < f(x^M) \\ U_2(\tilde{x}; \mathcal{X}) & \text{if} \quad \tilde{x} > x^M \text{ and } f(\tilde{x}) < f(x^M) \\ U_3(\tilde{x}; \mathcal{X}) & \text{if} \quad \tilde{x} > x^M \text{ and } f(\tilde{x}) > f(x^M) \\ U_4(\tilde{x}; \mathcal{X}) & \text{if} \quad \tilde{x} < x^M \text{ and } f(\tilde{x}) > f(x^M) \end{cases}, \tag{5.18}$$

where

$$U_1(\tilde{x}; \mathcal{X}) := \begin{pmatrix} x_3^L & x_2^L & x_1^L & \tilde{x} & x^M & x_1^R & x_2^R \end{pmatrix}^T, \tag{5.19a}$$

$$U_2(\tilde{x}; \mathcal{X}) := \begin{pmatrix} x_2^L & x_1^L & x^M & \tilde{x} & x_1^R & x_2^R & x_3^R \end{pmatrix}^T, \tag{5.19b}$$

$$U_3(\tilde{x}; \mathcal{X}) := \begin{pmatrix} x_3^L & x_2^L & x_1^L & x^M & \tilde{x} & x_1^R & x_2^R \end{pmatrix}^T, \tag{5.19c}$$

$$U_4(\tilde{x}; \mathcal{X}) := \begin{pmatrix} x_2^L & x_1^L & \tilde{x} & x^M & x_1^R & x_2^R & x_3^R \end{pmatrix}^T. \tag{5.19d}$$

Remark. *The update function $\sigma_S(\mathcal{X})$ is the natural extension of the one used by the Mifflin-Strodiot method [83] to 7 points.*

Lemma 16. *Let $\mathcal{X}_1$ be an extended bracket. If $\tilde{x} = \sigma_S(\mathcal{X}_1) \in (x_1^L, x^M) \cup (x^M, x_1^R)$, then $\mathcal{X}_2 = U_S(\tilde{x}; \mathcal{X}_1, \alpha)$ is an extended bracket and $b(\mathcal{X}_2) < b(\mathcal{X}_1)$.*

Proof. If $\tilde{x} \in (x_1^L, x^M)$, then the update function U applies either U_1 or U_4 to $\mathcal{X}_1$. In both cases, $\tilde{x}$ is placed between x_1^L and x^M which implies that $\mathcal{X}_2$ satisfies Equation (5.15a).

Given that $\mathcal{X}_1$ satisfies Equation (5.15b), it follows that x^M is the location within $[x_1^L, x_1^R]$ where the smallest value of f has been evaluated. If $f(\tilde{x}) < f(x^M)$, then U_S applies U_1, which implies $\mathcal{X}_2$ satisfies Equation (5.15b). Alternatively if $f(\tilde{x}) > f(x^M)$, then U_S applies U_4, in which case $\mathcal{X}_2$ still satisfies Equation (5.15b). Since $\mathcal{X}_2$ satisfies Equations (5.15a) and (5.15b), it is an extended bracket.

The equivalent argument may be applied to the case where $\tilde{x} \in (x^M, x_1^R)$, exchanging U_1 and U_4 for U_2 and U_3 respectively. □

5.3.2 Static underestimating polynomial method

The first variation of the UPM which we discuss is the static underestimating polynomial method (SUPM). Here the word static refers specifically to the treatment of the parameter α. The SUPM requires a user defined value of α which then remains constant. In this section, we provide intuition as to the meaning of the parameter α and show the effect of α on the convergence of the UPM.

5.3.2.1 SUPM convergence theory

In the previous section, we defined U_S, the update function for the SUPM, and showed in Lemma 16 that it maps an extended bracket which satisfies Equations (5.15a) and (5.15b) to a new extended bracket which satisfies the same conditions by inserting the point $\sigma_S(\mathcal{X})$. However, an assumption made in the statement of Lemma 16 was that $\sigma_S(\mathcal{X}) \in (x_1^L, x^M) \cup (x^M, x_1^R)$. Ultimately we will require a stability perturbation if $\sigma_S(\mathcal{X}) = x^M$. However, we would like $\sigma_S(\mathcal{X}) \in (x_1^L, x_1^R)$ to hold naturally and without the need for intervention. As we work towards this, we introduce the following definition:

Definition 5.9. *Let f be a piecewise smooth function. We call f* ***locally unimodal*** *if for any local minimum x^* of f, there exists an open set $(a, b) \ni x^*$ such that f is unimodal on (a, b).*

When constructing convergence results, there are three levels of assumptions we might make. First is the case where f is a piece-wise smooth function, $\mathcal{X}$ is a bracket, and nothing more is assumed. In this general context, we can do nothing beyond defining a minimum distance between points δ. If $|\sigma_S(\mathcal{X}) - x^M| < \delta$, we replace $\sigma_S(\mathcal{X})$ with $\operatorname{argmin}_x |x - \sigma_S(\mathcal{X})|$ such that $|x - x^M| \geq \delta$, $x_1^R - x \geq \delta$ and $x - x_1^L \geq \delta$. If $\sigma_S(\mathcal{X}) = x^M$, then there exists two equally valid solutions: $x^M + \delta$ and $x^M - \delta$. When this applies, we arbitrarily set $\sigma_S(\mathcal{X}) = x^M - \delta$. By doing this, we are sure to converge eventually, even if slowly.

For the next level of assumptions, we additionally require f to be in the form $f(x) = \max(f_L(x), f_R(x))$, and $\mathcal{X}$ satisfy $f(x_j^L) = f_L(x_j^L)$, $j = 1, 2, 3$ and $f(x_j^R) = f_L(x_j^R)$, $j =$

$1, 2, 3$. Given these assumptions, we show in Lemma 18 and Corollary 5.1 that $\sigma_S(\mathcal{X}; \alpha) \in (x_1^L, x_1^R)$ given a sufficiently large choice of α. In short, when there are not too many kinks in one place, then the UPM selects sensible points to evaluate f at.

Finally, we add the assumptions that f is unimodal and $diam(\mathcal{X})$ is sufficiently small. Under these circumstances, we show in Lemma 19 and Theorem 20 that the distance between $\sigma_S(\mathcal{X})$ and the true solution x^* can be bounded.

Taken in the context of locally unimodal functions, these results will imply that the SUPM is stable when the function is not unimodal, and fast when it is. This is sufficient for our purposes given that a stable algorithm applied to a locally unimodal function will eventually converge to a bracket in which the function is unimodal.

For the analysis that follows, we require the following external result:

Theorem 17 ([80])**.** *Suppose that $k, n \geq 0$; $f \in C^{n+k}[a, b]$; $\zeta \in [a, b]$; and $x_0, \ldots, x_n$ are distinct points in $[a, b]$. Then*

$$\begin{aligned} f[x_0, \ldots, x_n] = \frac{f^{(n)}(\zeta)}{n!} + \left(\sum_{0 \leq r_1 \leq n} (x_{r_1} - \zeta) \right) \frac{f^{(n+1)}(\zeta)}{(n+1)!} \\ + \ldots + \left(\sum_{0 \leq r_1 \leq \ldots \leq r_k \leq n} \prod_{j=1}^{k} (x_{r_j} - \zeta) \right) \frac{f^{(n+k)}(\zeta)}{(n+k)!} + E \end{aligned} \tag{5.20}$$

where

$$E = \frac{1}{(n+k)!} \left(\sum_{0 \leq r_1 \leq \ldots \leq r_k \leq n} \left(\prod_{j=1}^{k} (x_{r_j} - \zeta) \right) [f^{(n+k)}(\xi_{r_1, \ldots, r_k}) - f^{(n+k)}(\zeta)] \right), \tag{5.21}$$

and $\xi_{r_1, \ldots, r_k}$ are points in the interval spanned by $x_{r_1}, \ldots, x_{r_k}$ and ζ.

We prove the existence of a constant α and $h(\mathcal{X})$ such that $\sigma_S(\mathcal{X}) \in (x_1^L, x_1^R)$.

Lemma 18. *Let f be a function of the form $f = \max(f_L, f_R)$ such that f_L'' and f_R'' are both Lipschitz continuous with Lipschitz constants M_L and M_R respectively, and $\mathcal{X}$ be an extended bracket of f where $f(x_j^k) = f_k(x_j^k)$ for $j = 1, 2, 3$ and $k \in \{L, R\}$. If $\alpha \geq \frac{1}{2} max(M_L, M_R)$ and $h(\mathcal{X}) = \max(x_3^R - x_1^L, x_1^R - x_3^L)$, then $q^R(x; \mathcal{X}, \alpha) < f_R(x)$ $\forall x \in [x_1^L, x_1^R)$ and $q^L(x; \mathcal{X}, \alpha) < f_L(x)$ $\forall x \in (x_1^L, x_1^R]$.*

Proof. Suppose $y \in (x_1^L, x_1^R]$. Since $f(x_j^L) = f_L(x_j^L)$ for $j = 1, 2, 3$, we have:

$$
\begin{aligned}
& f_L(y) > q^L(y; \mathcal{X}, \alpha), \\
\Leftrightarrow & f_L(y) > f_L(x_1^L) + f_L[x_1^L, x_2^L](y - x_1^L) + (f_L[x_1^L, x_2^L, x_3^L] - \alpha h(\mathcal{X}))(y - x_1^L)(y - x_2^L), \\
\Leftrightarrow & f_L[y, x_1^L] > f_L[x_1^L, x_2^L] + (f_L[x_1^L, x_2^L, x_3^L] - \alpha h(\mathcal{X}))(y - x_2^L), \\
\Leftrightarrow & f_L[y, x_1^L, x_2^L] > f_L[x_1^L, x_2^L, x_3^L] - \alpha h(\mathcal{X}), \\
\Leftrightarrow & \alpha h(\mathcal{X}) > f_L[x_1^L, x_2^L, x_3^L] - f_L[y, x_1^L, x_2^L].
\end{aligned}
$$

Analogously, for $y \in [x_1^L, x_1^R)$ we have:

$$\alpha h(\mathcal{X}) > f_R[x_1^R, x_2^R, x_3^R] - f_R[y, x_1^R, x_2^R].$$

From Theorem 17, we know that there exists $\xi_1^R \in [x_1^R, x_3^R]$ such that $\frac{1}{2} f_R''(\xi_1^R) = f_R[x_1^R, x_2^R, x_3^R]$ and $\xi_2^R \in [y, x_2^R]$ such that $\frac{1}{2} f_R''(\xi_2^R) = f_R[y, x_1^R, x_2^R]$. Equivalently, the points ξ_1^L and ξ_2^L exist for $f_L[x_1^L, x_2^L, x_3^L]$ and $f_L[y, x_1^L, x_2^L]$ respectively. Therefore:

$$
\begin{aligned}
f_L[x_1^L, x_2^L, x_3^L] - f_L[y, x_1^L, x_2^L] &= \tfrac{1}{2}\left(f_L''(\xi_1^L) - f_L''(\xi_2^L)\right), \\
&\leq \tfrac{1}{2} M_L |\xi_1^L - \xi_2^L|, \\
&\leq \tfrac{1}{2} M_L |\max(y, x_1^L) - x_3^L|.
\end{aligned}
$$

The bound on the RHS depends on the value of y, and is itself bounded by $\frac{1}{2} M_L |x_1^R - x_3^L|$ when considering $y \in [x_1^L, x_3^R]$. Therefore, a sufficient condition for $f_L(y) > q^L(y; \mathcal{X}, \alpha)\ \forall y \in (x_1^L, x_1^R]$ is that $\alpha h(\mathcal{X}) > \frac{1}{2} M_L |x_1^R - x_3^L|$. Similarly, a sufficient condition for $f_R(y) > q^R(y; \mathcal{X}, \alpha)\ \forall y \in [x_1^L, x_1^R)$ is $\alpha h(\mathcal{X}) > \frac{1}{2} M_R |x_3^R - x_1^L|$.

Choosing $h(\mathcal{X}) = \max(x_1^R - x_3^L, x_3^R - x_1^L)$, and $2\alpha > \max(M_L, M_R)$ ensures that both of these are satisfied. □

Corollary 5.1. *Let f be a function of the form $f = \max(f_L, f_R)$ such that f_L'' and f_R'' are both Lipschitz continuous with Lipschitz constants M_L and M_R respectively, and $\mathcal{X}$ be a an extended bracket of f where $f(x_j^k) = f_k(x_j^k)$ for $j = 1, 2, 3$ and $k \in \{L, R\}$. If*

$\alpha \geq \frac{1}{2}\max(M_L, M_R)$ *and* $h(\mathcal{X}) = \max(x_1^R - x_3^L, x_3^R - x_1^L)$*, then* $\sigma_S(\mathcal{X}) \in (x_1^L, x_1^R)$.

Proof. As Lemma 18 is applicable, we know that $q^R(x) < f_R(x)\ \forall x \in [x_1^L, x_1^R)$ and $q^L(x) < f_L(x)\ \forall x \in (x_1^L, x_1^R]$. Since $f = \max(f_L, f_R)$, it follows that $q^L(x^M) < f(x^M)$ and $q_R(x^M) < f(x^M)$. For convenience, write $q(x) = \max(q^L(x; \mathcal{X}, \alpha), q^R(x; \mathcal{X}, \alpha))$. Starting from Equation (5.17), we obtain:

$$q(\sigma_S(\mathcal{X}; \alpha)) \leq q(x^M) < f(x^M) \leq \min(f(x_1^L), f(x_1^R)) = \min(q^L(x_1^L), q^R(x_1^R)).$$

But $q(x_1^L) = q^L(x_1^L)$ and $q(x_1^R) = q^R(x_1^R)$. Therefore $q(\sigma_S(\mathcal{X}; \alpha)) < \min(q(x_1^L), q(x_1^R))$, which implies that $\sigma_S(\mathcal{X}; \alpha) \neq x_1^L$ or x_1^R. Since $\sigma_S(\mathcal{X}; \alpha) \in [x_1^L, x_1^R]$ by definition (see Equation (5.17)), it follows that $\sigma_S(\mathcal{X}; \alpha) \in (x_1^L, x_1^R)$. □

Remark. *Between Corollary 5.1 and the requirement that the minimum distance between any two points in* $\mathcal{X}$ *is* δ*, we are assured that the conditions for Lemma 16 will be satisfied.*

For the remainder of this section, we work towards bounding the convergence of the SUPM when f is piecewise-smooth and unimodal. We define x^* to be the unique local minimum of f in (x_1^L, x_1^R). When assessing the SUPM's convergence rate, we consider two cases based on the values of $f_L'(x^*)$ and $f_R'(x^*)$.

If $f_R'(x^*) = 0$ or $f_L'(x^*) = 0$ holds, then we expect the SUPM to behave similarly to Brent's Method. For example, if if $f_R'(x^*) = 0$ and $f_L'(x^*) < 0$, then we expect q^R to behave similarly to the interpolated polynomial from Brent's Method. The SUPM will be slower however for two reasons.

1. When the SUPM computes a new point, that value may be stored in x^M which does not impact the next computation but only the one after that.

2. As only q^R is converging, any point which is used to construct q^L will play little to no role in determining $\sigma_S(\mathcal{X})$, and is therefore useless.

We focus our attention primarily on the case where $f_L'(x^*) < 0$ and $f_R'(x^*) > 0$. In order to bound $|\sigma_S(\mathcal{X})|$, we need to understand what type of point $\sigma_S(\mathcal{X})$ returns.

We address this in Lemma 19. First however, we introduce a shorthand notation for Divided Differences which we use for the remainder of this section.

$$f_1^k = f(x_1^k),\ k \in \{L, R\},$$
$$f_2^k = f[x_1^k, x_2^k],\ k \in \{L, R\},$$
$$f_3^k = f[x_1^k, x_2^k, x_3^k],\ k \in \{L, R\}.$$

Lemma 19. *Let f be a piecewise smooth unimodal function of the form $f = \max(f_L, f_R)$ such that f_L'' and f_R'' are both Lipschitz continuous with Lipschitz constants M_L and M_R respectively, and $\mathcal{X}$ be an extended bracket of f. Further let α be given such that $2\alpha \geq \max(M_L, M_R)$. If $f_L'(0) < 0$ and $f_R'(0) > 0$, then there exists $\epsilon > 0$ such that if $\mathcal{X} \in (-\epsilon, \epsilon)^7$, then the model functions q^L and q^R intersect exactly once within (x_1^L, x_1^R) and $\sigma_S(\mathcal{X})$ returns this unique intersection point.*

Proof. Since $f = \max(f_L, f_R)$ is unimodal, and $\mathcal{X}$ is an extended bracket, it follows that $f(x_j^k) = f_k(x_j^k)$ for $j = 1, 2, 3$ and $k \in \{L, R\}$. We use this along with Lemma 18 to show: $q^L(x_1^L) = f(x_1^L) = f_L(x_1^L) \geq f_R(x_1^L) > q^R(x_1^L)$, and similarly $q^R(x_1^R) > q^L(x_1^L)$. By the Intermediate Value Theorem [61, Theorem 4.23], it follows that q^L intersects with q^R at least once in (x_1^L, x_1^R). Moreover, since q^L and q^R are both quadratic functions, they must intersect exactly once.

A sufficient but not necessary condition for the result to follow would be $\exists \epsilon > 0$ such that q^L and q^R are monotonically decreasing and increasing respectively. Once this holds, $\sigma_S(\mathcal{X}) = \operatorname{argmin}_x \max(q^L(x), q^R(x))$ must refer to the intersection of q^L and q^R.

To show that q^L is monotonically decreasing, it is sufficient to prove that $(q^L)'(x_1^L) \leq 0$ and $(q^L)'(x_1^R) < 0$. Once this is shown, we are done since $(q^L)'$ is a linear function of x. We find :

$$(q^L)'(x_1^L) = f_2^L = \frac{f(x_1^L) - f(x_2^L)}{x_1^L - x_2^L} \leq 0,$$

since f is unimodal and $\mathcal{X}$ is an extended bracket of f (recall Definition 5.6). For

$(q^L)'(x_1^R)$ we begin with

$$(q^L)'(x_1^R) = f_2^L + (f_3^L - \alpha h(\mathcal{X}))(2x_1^R - x_1^L - x_2^L). \tag{5.22}$$

Next we apply Theorem 17 to conclude:

$$\exists \xi_1^L, \xi_2^L \in [x_2^L, x_1^L] \text{ such that } f_2^L = f_L'(0) + \tfrac{1}{2}(x_1^L f_L''(\xi_1^L) + x_2^L f_L''(\xi_2^L)),$$
$$\exists \xi_3^L \in [x_3^L, x_1^L] \text{ such that } f_3^L = \tfrac{1}{2} f_L''(\xi_3^L).$$

From this and Equation (5.22) it follows that

$$\begin{aligned}(q^L)'(x_1^R) = f_L'(0) &+ \tfrac{1}{2}\left(x_1^L f_L''(\xi_1^L) + x_2^L f_L''(\xi_2^L)\right) \\ &+ \tfrac{1}{2}(2x_1^R - x_1^L - x_2^L)\left(f_L''(\xi_3^L) - 2\alpha h(\mathcal{X})\right).\end{aligned}$$

Since f_L'' and f_R'' are Lipschitz continuous, we know that they are bounded on $[x_3^L, x_3^R]$. Let M be constant which bounds f_L'' and f_R''. We use this along with the fact that $\mathcal{X} \in (-\epsilon, \epsilon)^7$ to conclude:

$$\begin{aligned}(q^L)'(x_1^R) &\leq f_L'(0) + \tfrac{1}{2}\left(-x_1^L M - x_2^L M + (2x_1^R - x_1^L - x_2^L)(M - 2\alpha h(\mathcal{X})\right), \\ (q^L)'(x_1^R) &\leq f_L'(0) + \tfrac{1}{2}\left(2\epsilon M + 4\epsilon(M - 2\alpha h(\mathcal{X})\right), \\ &\leq f_L'(0) + \epsilon\left(3M - 4\alpha h(\mathcal{X})\right) \leq f_L'(0) + 3\epsilon M,\end{aligned}$$

since $0 < h(\mathcal{X}) < 2\epsilon$ and α is constant. Therefore a sufficient condition for $(q^L)'(x_1^R) < 0$ to hold is that $\epsilon < |f_L'(0)|/3M$. That q^R is monotonically increasing given sufficiently small ϵ follows from the equivalent argument. □

Having established the conditions under which we can express $\sigma_S(\mathcal{X})$ analytically, we now bound the distance between this point and the true minimiser.

Theorem 20. *Let $f : [a, b] \to \mathbb{R}$ be a piecewise smooth unimodal function of the form $f = \max(f_L, f_R)$ such that $x^* \in [a, b]$ is a minimiser of f, $f_L'(x^*) < 0$, $f_R'(x^*) > 0$, and $\mathcal{X}$*

be an extended bracket of f. If the functions f_L'' and f_R'' are both Lipschitz continuous (with constant M_L and M_R) and $\alpha > \frac{1}{2}\max(M_L, M_R)$, then there exists $\epsilon > 0$ such that

$$|\sigma_S(\mathcal{X})| \leq \frac{\sqrt{2}}{f_R'(x^*) - f_L'(x^*)} \big(|x_1^L - x^*||x_2^L - x^*|(|x_1^L - x^*| + |x_3^L - x^*|)M_L \\ + |x_1^R - x^*||x_2^R - x^*|(|x_1^R - x^*| + |x_3^R - x^*|)M_R\big),$$

when $\mathcal{X} \in (-\epsilon, \epsilon)^7$.

Proof. Since Lemma 19 applies, we know that $\sigma_S(\mathcal{X})$ returns the unique point of intersection between q^L and q^R in (x_1^L, x_1^R). The intersection points of these quadratics are the roots to the polynomial $ax^2 + bx + c = 0$, where:

$$a = f_3^R - f_3^L,$$
$$b = f_2^R - f_2^L - (x_1^R + x_2^R)(f_3^R - \alpha h) + (x_1^L + x_2^L)(f_3^L - \alpha h),$$
$$c = f_1^R - f_1^L - x_1^R f_2^R + x_1^L f_2^L + x_1^R x_2^R (f_3^R - \alpha h) - x_1^L x_2^L (f_3^L - \alpha h).$$

Assuming without loss of generality $x^* = 0$, we invoke Theorem 17 to make the following substitutions for $\ell \in \{L, R\}$.

$$f_1^\ell = f_\ell(0) + x_1^\ell f_\ell'(0) + \tfrac{1}{2}(x_1^\ell)^2 f_\ell''(\xi_1^\ell), \qquad \xi_1^\ell \in [0, x_1^\ell],$$
$$f_2^\ell = f_\ell'(0) + \tfrac{1}{2}\left(x_1^\ell f_\ell''(\xi_2^\ell) + x_2^\ell f_\ell''(\xi_3^\ell)\right), \qquad \xi_2^\ell \in [0, x_1^\ell],\ \xi_3^\ell \in [0, x_2^\ell],$$
$$f_3^\ell = \tfrac{1}{2} f_\ell''(\xi_4^\ell), \qquad \xi_4^\ell \in [0, x_3^\ell].$$

By inserting the substitutions above into the definitions of a, b and c, we get:

$$a = \tfrac{1}{2}\left(f_R''(0) - f_L''(0)\right) + K_a^R - K_a^L,$$
$$b = f_R'(0) - f_L'(0) + K_b^R - K_b^L,$$
$$c = K_c^R - K_c^L,$$

where K_a^ℓ, K_b^ℓ and K_c^ℓ are in turn defined for $\ell \in \{L, R\}$ by:

$$
\begin{aligned}
K_a^\ell &= \tfrac{1}{2}\left(f_\ell''(\xi_4^\ell) - f_\ell''(0)\right), \\
K_b^\ell &= \tfrac{1}{2}\left(x_1^\ell(f_\ell''(\xi_2^\ell) - f_\ell''(\xi_4^\ell)) + x_2^\ell(f_\ell''(\xi_3^\ell) - f_\ell''(\xi_4^\ell))\right) + (x_1^\ell + x_2^\ell)\alpha h(\mathcal{X}), \\
K_c^\ell &= \tfrac{1}{2}x_1^\ell\left(x_1^\ell(f_\ell''(\xi_1^\ell) - f_\ell''(\xi_2^\ell)) + x_2^\ell(f_\ell''(\xi_4^\ell) - f_\ell''(\xi_3^\ell))\right) - x_1^\ell x_2^\ell \alpha h(\mathcal{X}).
\end{aligned}
$$

Writing a, b and c in terms of these K values allows us to give bounds using the assumption that f_L'' and f_R'' are Lipschitz Continuous .

$$|K_a^\ell| \le \tfrac{1}{2}M_\ell|x_3^\ell| \le \tfrac{1}{2}M_\ell\epsilon, \tag{5.23a}$$

$$|K_b^\ell| \le \tfrac{1}{2}(|x_1^\ell| + |x_2^\ell|)\left(M_\ell|x_3^\ell| + \alpha h(\mathcal{X})\right) \le \epsilon^2(M_\ell + 2\alpha), \tag{5.23b}$$

$$|K_c^\ell| \le \tfrac{1}{2}|x_1^\ell||x_2^\ell|M_\ell\left(|x_1^\ell| + |x_3^\ell|\right) \le M_\ell\epsilon^3. \tag{5.23c}$$

Choose an ϵ_1 small enough to ensure $\frac{1}{2}(f_R'(0) - f_L'(0)) > K_b^L - K_b^R$ (note that $b \to f_R'(0) - f_L'(0)$ as $\epsilon \to 0$). If $\epsilon \le \epsilon_1$, then it follows that $b > 0$ which in turn implies:

$$\sigma_S(\mathcal{X}) = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \tag{5.24}$$

because the other root lies outside $[x_1^L, x_1^R]$ as $\epsilon \to 0$. We apply Taylor's Theorem to conclude $\exists \xi \in [-|4ac|, |4ac|]$ such that $\sqrt{b^2 - 4ac} = b + 2ac/\sqrt{b^2 - \xi}$. Since $ac \to 0$ as $\epsilon \to 0$, there exists ϵ_2 such that $|4ac| \le \frac{1}{2}b^2$ when $\epsilon < \epsilon_2$. Then if $\epsilon < \min(\epsilon_0, \epsilon_1, \epsilon_2)$, we have:

$$
\begin{aligned}
|\sigma_S(\mathcal{X})| &= \left|\frac{-b + \sqrt{b^2 - 4ac}}{2a}\right| = \frac{1}{2|a|}\left|\frac{2ac}{\sqrt{b^2 - \xi}}\right|, && \xi \in [0, 4ac], \\
&\le \frac{|c|}{\sqrt{b^2 - \xi}} \le \frac{|c|}{\sqrt{b^2 - 4|ac|}} \le \sqrt{2}\left|\frac{c}{b}\right|, \\
&= \sqrt{2}\frac{K_c^R - K_c^L}{f_R'(0) - f_L'(0) + K_b^R - K_b^L} \le 2\sqrt{2}\frac{|K_c^R| + |K_c^L|}{f_R'(0) - f_L'(0)},
\end{aligned}
$$

where the last step follows from us having previously chosen ϵ small enough to ensure $\frac{1}{2}(f_R'(0) - f_L'(0)) > K_b^L - K_b^R$. Finally we insert Equation (5.23c) and the result follows.

□

Theorem 20 is remarkably similar to [86, Theorem 4.1], the equivalent result which bounds the convergence of the MSA. Using our own notation for their result, Mifflin and Strodiot showed $|x_i^M - x^*| \leq A_i|x^* - x_{1,i}^L| + B_i|x_{1,i}^L - x^*|$ for each iteration i, where A_i and B_i are sequences which converge to 0 as $i \to \infty$. If we rewrite the result from Theorem 20 in the same form, the sequence A_i may be written explicitly in terms of $x_{1,i}^L, x_{2,i}^L$, and $x_{3,i}^L$, and similarly for B_i. Then, if $x_{1,i}^L$ and $x_{1,i}^L$ both converge to x^*, the rate at which A_i and B_i tend to zero begins to resemble quadratic convergence.

However, Theorem 20, like [86, Theorem 4.1] does not prove quadratic convergence, nor even linear convergence. This is because we cannot know a priori whether the sequences $\{x_{1,i}^L\}$ and $\{x_{1,i}^L\}$ converge at all, nor how quickly if they do. Suppose there exists an iteration i_0 such that $\forall i > i_0$, $x_{1,i}^L$ is constant. In this case, the bound given by Theorem 20 will decrease, but not converge to 0. In practice, when this occurs x_i^M will still converge to x^* but only sub-linearly, while $b(\mathcal{X}_i) \not\to 0$.

On the other hand, suppose that both $\{x_{1,i}^L\}$ and $\{x_{1,i}^L\}$ converge to x^*, but we only update $x_{1,i}^L$ once every k iterations. Then we may say that $|x_{ki}^M - x^*| \to 0$ super-linearly as $i \to \infty$, but depending on the size of k, this may not be particularly useful.

In short, there are three main weaknesses of the SUPM which lead to its failure. The first is that we need to choose α, a suitable value of which we cannot know in advance. The second is that convergence only occurs in reasonable time if both updates of $x_{1,i}^L$ and $x_{1,i}^L$ occur regularly. Finally, we require a bracket sufficiently close to the solution that f is unimodal before any of our results apply. These shortcomings of the SUPM motivate subsequent sections, in which we describe more stable variations of the UPM.

5.3.2.2 Numerical performance of the SUPM

Having discussed the theoretical properties of the SUPM, in this section we now show it in action. We will demonstrate how its performance is affected by choice of α and on the objective function. Before showing the results of our numerical experiment, we first provide specific examples to show what occurs in practice.

When testing the SUPM, we apply it to three categories of test-functions: smooth unimodal, non-smooth unimodal and smooth multimodal. To our knowledge there is not a commonly used set of univariate test functions. Therefore, we have taken some test functions from Jamil et al [87], keeping the ones which are well defined for one variable. Adding to this, we have added a few of our own design to be adversarial examples based on our observations. We scale each test function f such that the minimum Lipschitz constant for f'' lies between 0.8 and 1. For the category of smooth unimodal, we use the following test functions:

$$f_1^{SU} = -\frac{1}{\sqrt{e}} \exp\left(-\tfrac{1}{2}x^2\right), \qquad -1 \le x \le 1, \tag{5.25}$$

$$f_2^{SU} = \frac{1}{24}x^4, \qquad -1 \le x \le 1, \tag{5.26}$$

$$f_3^{SU} = \frac{1}{11}\left(-\sin\left(2x - \tfrac{1}{2}\pi\right) - 3\cos x - \tfrac{1}{2}x\right), \qquad -2.5 \le x \le 3, \tag{5.27}$$

$$f_4^{SU} = \frac{1}{2500}\left(\frac{x^2}{2} - \frac{\cos(5\pi x)}{25\pi^2} - \frac{x\sin(5\pi x)}{5\pi}\right), \qquad -10 \le x \le 10, \tag{5.28}$$

$$f_5^{SU} = -\frac{1}{250}\left(x^{\frac{2}{3}} + (1-x^2)^{\frac{1}{3}}\right), \qquad 0.1 \le x \le 0.9, \tag{5.29}$$

$$f_6^{SU} = \frac{1}{6000}\left(e^x + \frac{1}{\sqrt{x}}\right), \qquad 0.1 \le x \le 3, \tag{5.30}$$

$$f_7^{SU} = -\frac{1}{13}(16x^2 - 24x + 5)e^{-x}, \qquad 1.3 \le x \le 3.9. \tag{5.31}$$

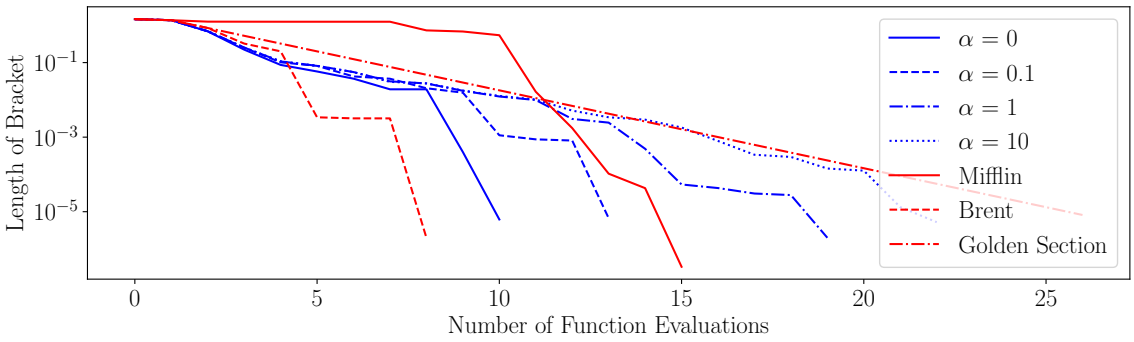

Figure 5.3.2 – We demonstrate the SUPM with various values of α when applied to $f_1^{SU}(x)$.

The results obtained from running the SUPM on $f_1^{SU}(x)$ are shown in Figure 5.3.2.

We see that Brent's method converges fastest, which is not surprising given that Brent's method is designed for such functions. Next we see the SUPM converging slightly slower than Brent's method for $\alpha = 0, 0.1, 1$. In fact, it seems that the larger the value of α, the slower convergence appears. While the MSA ultimately converges quickly, it does so in a less robust manner, making no progress initially before rapidly converging in the end.

The next class of test functions we consider is that of non-smooth unimodal:

$$f_1^{NU} = -60000 \exp\left(-\frac{|x|}{50}\right), \qquad -32 \le x \le 32, \tag{5.32}$$

$$f_2^{NU} = \frac{1}{6} \max\left(\frac{1}{x+3}, \log(x)\right), \qquad -2 \le x \le 10, \tag{5.33}$$

$$f_3^{NU} = \frac{1}{24} \max\left(\frac{1}{x+3}, \frac{1}{(x-3)^2}\right), \qquad -2 \le x \le 2, \tag{5.34}$$

$$f_4^{NU} = \frac{1}{160} \max\left(\frac{1}{x+3}, \exp(x)\right), \qquad -2 \le x \le 5, \tag{5.35}$$

$$f_5^{NU} = \frac{1}{150} \max\left(\exp(-x), \exp(x)\right), \qquad -5 \le x \le 5. \tag{5.36}$$

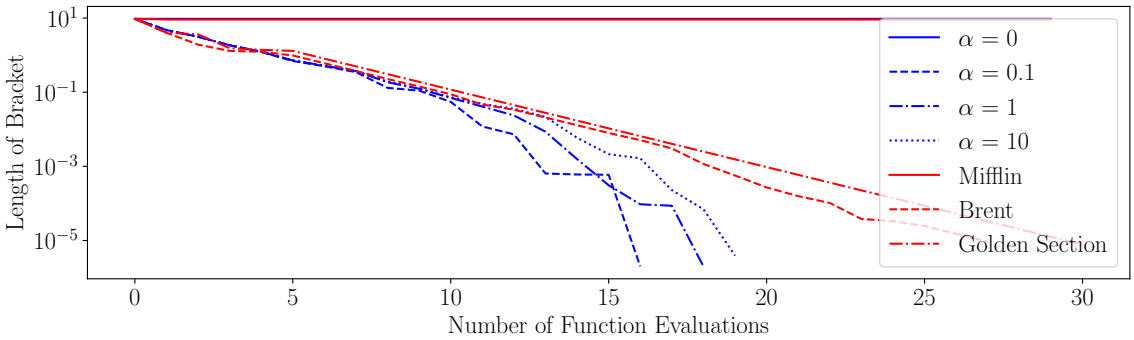

Figure 5.3.3 – We demonstrate the SUPM with various values of α on the f_2^{NU}.

From Figure 5.3.3, we observe that the SUPM fails to converge when $\alpha = 0$. Excluding $\alpha = 0$, we see that the larger α is, the slower the SUPM converges. While convergence occurs at different rates, the SUPM appears to converge super-linearly for all finite values of $\alpha > 0$. Meanwhile, the performance of Brent's method is barely distinguishable from Golden section. Last of all, the MSA fails entirely to converge. Its worth noting that f_2^{NU} is non-convex, a property which the other function which we previously showed the MSA struggles with shares.

Finally we list our smooth multimodal test functions:

$$f_1^{SM} = \frac{x^6}{300}\left(2 + \sin\left(\frac{1}{x}\right)\right), \qquad -1 \le x \le 1, \tag{5.37}$$

$$f_2^{SM} = -\frac{1}{40000}\sin\left(5\pi x\right)^6, \qquad -1 \le x \le 1, \tag{5.38}$$

$$f_3^{SM} = -\frac{1}{250000}\sin\left(5\pi(x^{\frac{3}{4}} - \frac{1}{20})\right)^6, \qquad 0.01 \le x \le 1, \tag{5.39}$$

$$f_4^{SM} = \frac{1}{5}\left(\sin\left(\frac{16}{15}x - 1\right) + \sin\left(\frac{16}{15}x - 1\right)^2\right), \qquad -1 \le x \le 1, \tag{5.40}$$

$$f_5^{SM} = \frac{x^2}{4000} - \cos x + 1, \qquad 100 \le x \le 100, \tag{5.41}$$

$$f_6^{SM} = \frac{1}{71}\left(\left(\log(x-2)\right)^2 + \left(\log(10-x)\right)^2 - x^{\frac{1}{5}}\right), \qquad 2.5 \le x \le 9.5, \tag{5.42}$$

$$f_7^{SM} = \frac{1}{40}\left(\sin(x) + \sin\left(\frac{10x}{3}\right) + \log(x) + \frac{21}{25}x\right), \qquad 0.5 \le x \le 10. \tag{5.43}$$

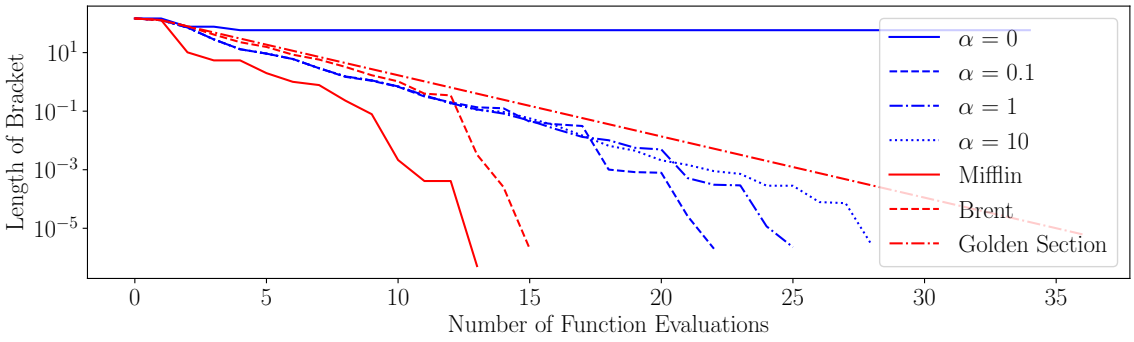

Figure 5.3.4 – We demonstrate the SUPM with various values of α on the function f_5^{SM}.

For this example, the MSA is clearly the fastest, followed closely by Brent's method. Similarly to the non-smooth example, we see the SUPM getting slower as α increases. This is of course with the exception of $\alpha = 0$ where the SUPM fails to converge. It is worth noting that even though none of the theory for the SUPM was designed for multi modal functions, it often converges anyway.

In the examples we have shown, there exist a few occasions for which the SUPM has failed to converge. In order to show why this occurs, we present a specific example below, showing what properties q^L and q^R have when this happens.

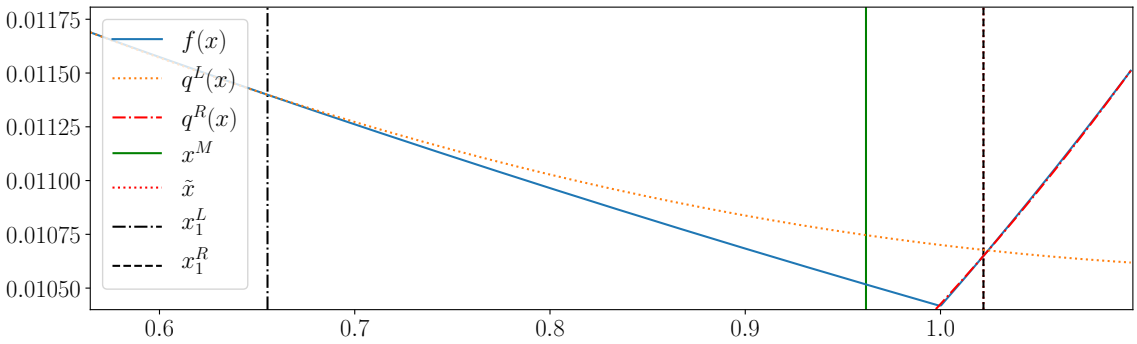

Figure 5.3.5 – When the SUPM is run for 6 iterations on f_3^{NU} from the starting bracket $\mathcal{X}_0$ (see Example 5.2), the model functions q^L and q^R are as plotted above .

Example 5.2. *Consider the objective function* f_3^{NU} *with* $\alpha = 0$ *and*

$$\mathcal{X}_0 = (-1.811263, -2.171330, -2.23927, 1.820150, 2.102197, 2.293404, 2.334091).$$

If we run the SUPM up to the 6th iteration, we find the scenario illustrated in Figure 5.3.5.

The value of $\tilde{x}$*, which is calculated to be the minimum of* $\max(q^L(x), q^R(x))$ *within* (x_1^L, x_1^R) *is evaluated to be* x_1^R*. Moreover,* $\tilde{x}$ *is determined only by* q^L *because* $q^L(x) > q^R(x)$ *for every* $x \in [x_1^L, x_1^R]$*. This is problematic given that the location of* $\tilde{x}$ *means that* q^R *will be updated for the following iteration, but* q^L *will be left the same. Thus the same thing will happen again, resulting the SUPM stalling.*

Our takeaway from this example is that the UPM process may fail if the model functions q^L and q^R do not underestimate the objective function f. This is a consequence of α not being large enough. In fact, this failure to underestimate f is the same problem which affects the MSA. Not only this, but the MSA is more vulnerable to this problem then the UPM given that it relies on linear approximations and not quadratic ones. Thus, for any non convex function, the MSA the linear approximation constructed by the MSA is guaranteed to overestimate f in places.

Having presented some specific examples which illustrate the performance of the SUPM, we now conduct a more rigorous experiment. For each test function, we run

Brent's Method, the MSA and the SUPM with α values $0, 0.1, 1, 10$ for randomly generated starting brackets.

Each test function has an interval $[a, b]$ on which it is intended to be used. To construct our random starting brackets, we sample 4 points uniformly from both $[a, a + \frac{1}{5}(b-a)]$ and $[b - \frac{1}{2}(b-a), b]$. After evaluating the function at these 8 points, we select the point where f is minimised to be x^M; usually this will be one of the points closer to $\frac{1}{2}(b-a)$. Finally we order the remaining points, and select the 3 closest on the left and right to be $x_3^L, x_2^L, \ldots, x_3^R$.

After generating the bracket, we run each algorithm and compute the average convergence rate over 1000 different starting brackets. If the algorithm converged for a particular starting bracket, then its average convergence rate is by definition between 0 and 1, the smaller the better. If an algorithm did not converge, we assign ∞ instead.

We do this because the purpose of the SUPM is to be a robust algorithm. Since we are constructing the underestimating polynomial method with line search in mind, we anticipate needing to call the UPM many times within the context of a line search method. If any proposed 1D optimisation method is not capable of converging robustly, then there is not point in not using Golden Section instead. Therefore we are not interested in it only having a probability of success. Either it always converges or it does not. We present the outcome of this experiment in Tables 5.3.1 to 5.3.3.

Functions	$\alpha = 0$	$\alpha = 0.1$	$\alpha = 1$	$\alpha = 10$	Brent	Mifflin
f_1^{SU}	0.3268	0.4273	0.4912	0.5472	0.2159	0.1477
f_2^{SU}	0.68	0.6164	0.6183	0.6187	0.3282	0.6614
f_3^{SU}	∞	0.4848	0.5438	0.5882	0.3329	∞
f_4^{SU}	∞	0.6105	0.6104	0.6104	0.5626	∞
f_5^{SU}	∞	0.5469	0.5925	0.6233	0.2756	0.2581
f_6^{SU}	∞	0.5871	0.615	0.6246	0.2812	∞
f_7^{SU}	0.35	0.4966	0.5521	0.5944	0.2575	∞

Table 5.3.1 – Average convergence rate of different algorithms on the set of smooth unimodal test functions.

From these tables, we see justification for our observations made earlier. In particular, when the SUPM converges, it usually converges faster for small values of α.

Functions	$\alpha = 0$	$\alpha = 0.1$	$\alpha = 1$	$\alpha = 10$	Brent	Mifflin
f_1^{NU}	∞	0.1868	0.2684	0.3272	0.5115	∞
f_2^{NU}	∞	0.3999	0.4445	0.4806	0.6337	∞
f_3^{NU}	∞	0.4538	0.4931	0.5255	0.6457	0.3259
f_4^{NU}	∞	0.4153	0.4665	0.5006	0.5934	0.3378
f_5^{NU}	∞	0.4204	0.4645	0.4974	0.4903	0.3592

Table 5.3.2 – Average convergence rate of different algorithms on the set of non-smooth unimodal test functions. We write ∞ when the algorithm fails to converge at least once.

Functions	$\alpha = 0$	$\alpha = 0.1$	$\alpha = 1$	$\alpha = 10$	Brent	Mifflin
f_1^{SM}	∞	0.6146	0.6138	0.6144	0.4828	∞
f_2^{SM}	∞	0.5076	0.5579	0.5961	0.3588	∞
f_3^{SM}	∞	0.5481	0.5918	0.6177	0.3372	∞
f_4^{SM}	∞	0.4536	0.5308	0.5789	0.2703	∞
f_5^{SM}	∞	0.4704	0.5188	0.5582	0.311	∞
f_6^{SM}	∞	0.5599	0.5962	0.6191	0.2982	∞
f_7^{SM}	∞	0.4733	0.5353	0.5776	0.273	∞

Table 5.3.3 – Average convergence rate of different algorithms on the set of smooth multimodal test functions.

However, the smaller the value of α, the less likely the SUPM will converge in the first place.

For the smooth test functions, we see that Brent's Method is usually the best, although it is occasionally beaten by the MSA. While the MSA converges very quickly at times, we see that it lacks the robustness of Brent's method, sometimes failing to converge even on smooth functions. Meanwhile for non-smooth functions, Brent's method performs little better then Golden Section. On the other hand, the SUPM performs between Brent's method and Golden Section for smooth functions and is far superior for non-smooth functions.

5.3.3 Extremal underestimating polynomial method

In Section 5.3.2.1 , we showed that if α was sufficiently large, then the σ_S is guaranteed to select a point within (x_1^L, x_1^R). Coupled with a restriction on the minimum distance between points, this is enough to ensure convergence eventually. However, for an arbitrary piecewise smooth objective function, we have no idea what a suitable value for α

might be. As there is no finite value of α which satisfies Lemma 18 for all functions, we ask what happens if we let $\alpha \to \infty$ and name the corresponding variation of the SUPM the extremal underestimating polynomial method (EUPM), denoting its step function and update function by σ_E and U_E respectively.

As $\alpha \to \infty$ we are assured by Lemma 18 that q^L and q^R will underestimate f_L and f_R respectively within $[x_1^L, x_1^R]$. Furthermore, if $h(\mathcal{X}) > 0$, then we can pick a sufficiently large α such that the coefficient of x^2 in $q^k(x; \mathcal{X}, \alpha)$ (see Equation (5.16)) will be negative. Once α satisfies both of these, then q^L and q^R will intersect at exactly one point, which is the point that $\sigma_S(\mathcal{X})$ returns. We define $\sigma_E(\mathcal{X})$ to be this point which we calculate by identifying the root of $q^R(x; \mathcal{X}, \alpha) - q^L(x; \mathcal{X}, \alpha) = 0$, which remains bounded as $\alpha \to \infty$. By starting from Equation (5.24) and noting the Taylor Series as $\alpha \to \infty$, we find:

$$\sigma_E(\mathcal{X}) = \frac{x_1^R x_2^R - x_1^L x_2^L}{x_1^R + x_2^R - x_1^L - x_2^L}. \tag{5.44}$$

Note that the step function σ_E does not depend on either x_3^L or x_3^R. Therefore, within the context of this section, we reduce the extended bracket $\mathcal{X}$ to the form:

$$\mathcal{X} = \begin{pmatrix} x_2^L & x_1^L & x^M & x_1^R & x_2^R \end{pmatrix}^T, \tag{5.45}$$

In order to construct U_E, we first define $\tilde{U}_S$ to be the update function U_S, where we remove the entries corresponding to x_3^R and x_3^L. Using this notation, we define:

$$U_E(\mathcal{X}) := \tilde{U}_S(\sigma_E(\mathcal{X}), \mathcal{X}). \tag{5.46}$$

Given σ_E and U_E, the EUPM is precisely in the form of Algorithm 12.

Remark. *Equation* (5.46) *is of the same form as the update function used by the Mifflin-Strodiot method [83].*

Remark. *We see the EUPM is qualitatively different from the SUPM in that σ_E depends only on the values of $\mathcal{X}$, and not on the function f at all. Therefore, like Golden Section,*

the EUPM is equally suited to smooth, non-smooth, unimodal and multi-modal functions and is scale invariant.

5.3.3.1 EUPM convergence theory

In this section we derive a bound for the convergence rate of the EUPM. In particular, we find an integer k and constant C such that $b(\mathcal{X}_{i+k})/b(\mathcal{X}_i) \leq C$ holds for any objective function. While there exist infinitely many possible objective functions, there are only finitely many possible values for $\mathcal{X}_{i+k}$ given $\mathcal{X}_i$. This is because σ_E does not depend on the objective function f and thus the only effect that f has on the EUPM is in the function U_E (See Equation (5.18)) by determining which update function: U_1, U_2, U_3 or U_4 to apply, none of which depend on f.

Therefore, there are only 4 possible values of $\mathcal{X}_{i+1}$ given $\mathcal{X}_i$: $U_j\mathcal{X}$ for $j = 1, 2, 3, 4$. Moreover, there are at most 4^k possible values of $\mathcal{X}_{i+k}$ given $\mathcal{X}_i$. Each possible value of $\mathcal{X}_{i+k}$ depends on the sequence of update functions which generated it.

Remark. *In this section, we will refer to the update functions U_1, U_2, U_3 and U_4 extensively. To simplify notation, we replace U_j with $\tilde{U}_j$, where $\tilde{U}_j(\mathcal{X}) \coloneqq U_j(\sigma_E(\mathcal{X}), \mathcal{X})$. For the remainder of Section 5.3.3, U_j should be read as $\tilde{U}_j$.*

Definition 5.10. *Let $\{\mathcal{X}_i\}_{i=0}^n$ be a sequence of extended brackets generated by the EUPM such that $\mathcal{X}_{i+1} = U_E(\mathcal{X}_i)$ for each i. Let $I = \{i_j\}_{j=0}^{n-1} \in \{1, 2, 3, 4\}^n$ be the sequence of numbers such that $\mathcal{X}_{j+1} = U_{i_j}(\mathcal{X}_j),\ \forall j \in \{0, 1, \ldots, n-1\}$. Then we call I the* sequence of EUPM iterations *which generate $\{\mathcal{X}_i\}_{i=0}^n$. Moreover we write:*

$$U_I \coloneqq U_{i_n} \circ \ldots U_{i_2} \circ U_{i_1},\ \textit{where } I = \{i_j\}_{j=1}^n.$$

Definition 5.11. *Define $\mathcal{I}_n$ to be the set of all length n sequences of EUPM iterations.*

Definition 5.12. *Define $Q_k(f, \mathcal{X}_0)$ to be the sequence of the first k iterations generated by the EUPM when applied to the function f with the starting bracket $\mathcal{X}_0$.*

Lemma 21. *Let f be a function, $\mathcal{X} \in \mathcal{B}_f$, and $I \in \mathcal{I}_n$ such that $I = \{I_1, I_2\} = Q_n(f, \mathcal{X}_0)$, where $I_1 \in \mathcal{I}_{n-k}$ and $I_2 \in \mathcal{I}_k$. Then it holds that: $Q_k(f, U_{I_1}\mathcal{X}_0) = I_2$.*

Proof. $I_1 = Q_{n-k}(f, \mathcal{X}_0)$ refers to the first $(n-k)$ iterations of the EUPM when applied to the function f with the starting bracket $\mathcal{X}_0$. On the other hand, I_2 refers to the next k iterations, suggesting that $I_2 = Q_k(f, \mathcal{X}_{n-k})$. But $\mathcal{X}_{n-k} = U_{I_1}\mathcal{X}_0$, thus implying $I_2 = Q_k(f, U_{I_1}\mathcal{X}_0)$. □

Using the notation from Definitions 5.10 and 5.12, it follows that $\mathcal{X}_{n+k} = U_{Q_k(f,\mathcal{X}_n)}(\mathcal{X}_n)$. Since $Q_k(f, \mathcal{X}_n) \in \mathcal{I}_k$ for every function f, it follows that:

$$\frac{b(\mathcal{X}_{n+k})}{b(\mathcal{X}_n)} \leq C \text{ for any function } f \Leftrightarrow \frac{b(U_{Q_k(f,\mathcal{X}_n)}\mathcal{X}_n)}{b(\mathcal{X}_n)} \leq C, \text{ for any function } f. \quad (5.47)$$

We'd like to say that Equation (5.47) is equivalent to saying $b(U_I\mathcal{X}_0)/b(\mathcal{X}_0) \leq C$ for every $I \in \mathcal{I}_k$, but unfortunately this is not true. This is because the set $\mathcal{I}_k$ is larger than the set of sequences containing elements of the form $Q_k(f, \mathcal{X}_0)$ for any f given a particular $\mathcal{X}_0$. This being the case, we need to define the following notation so as to be adequately rigorous later.

Definition 5.13. *Given $I \in \mathcal{I}_n$, we define:*

$$\tilde{\mathcal{B}}_I := \{\mathcal{X} \in \mathbb{R}^5 | \ \exists n > 0, \text{ and } \exists f \text{ such that } \mathcal{X} \in \mathcal{B}_f, \text{ and } Q_n(f, \mathcal{X}) = I\}.$$

This notation enables us to state our main convergence result.

Theorem 22. *The EUPM converges R-linearly. In particular:*

$$\max_{I \in \mathcal{I}_5} \max_{\mathcal{X} \in \tilde{\mathcal{B}}_I} \frac{b(U_I\mathcal{X})}{b(\mathcal{X})} \leq \frac{1}{2}.$$

Given how long the proof of Theorem 22 is, we reserve an entire section for it. The proof may be found in Section 5.3.3.2, although it may be skipped on first reading.

Theorem 22 provides a bound on the linear convergence rate for the EUPM albeit a weak one. In practice, we observe far faster convergence as will be demonstrated in Sections 5.3.3.3 and 5.3.5. The main justification for the EUPM is its robustness. In Section 5.3, we assumed that the SUPM started with a bracket that was already

sufficiently close to the solution. Once equipped with this, we might expect the SUPM to converge quickly. The EUPM is useful for producing such a bracket in the first place. In Section 5.3.4, we present the final variation of the UPM, in which we combine the initial robustness of the EUPM with the desirable properties of the SUPM.

5.3.3.2 Proof of Theorem 22

We divide this proof into two parts, which are defined by Lemmas 23 and 24. Within this section, we write particular sequences of iterations such as $I = \{4, 1, 3\}$ simply as $I = 413$.

Lemma 23. *It holds that* $b(U_I\mathcal{X})/b(\mathcal{X}) \leq \frac{1}{2}\ \forall I \in \mathcal{A}$ *where* $\mathcal{A} = \{44\ ,\ 111,\ 143,\ 422,\ 414,\ 434,\ 1411,\ 1141,\ 1423,\ 4322,\ 4314,\ 4114\}$.

Lemma 24. *If* $b(U_I\mathcal{X})/b(\mathcal{X}) \leq \frac{1}{2}\ \forall I \in \mathcal{A}$ *then* $b(U_I\mathcal{X})/b(\mathcal{X}) \leq \frac{1}{2}\ \forall I \in \mathcal{I}_5$, *where* $\mathcal{A}$ *is as defined in Lemma 23.*

Given that the proof of Lemma 23 is entirely algebraic calculations, we leave it for Appendix B.2. In this section, we work towards proving Lemma 24. Together, these two results are sufficient to prove Theorem 22.

To proceed, we need a rigorous definition of a sub-string.

Definition 5.14. *Let* $I = \{i_j\}_{j=1}^{n} \in \mathcal{I}_n$ *be a length* n *sequence of iterations of the EUPM. A sub-string of the sequence* I *is a subsequence of the form* $J = \{i_j\}_{j=k_1}^{k_2}$ *such that* $1 \leq k_1 < k_2 \leq n$. *Denote by* $\mathcal{S}(I)$ *the set of sub-strings of the sequence* I.

The first step we take towards proving Lemma 24 is to show that $b(U_I\mathcal{X})/b(\mathcal{X}) \leq \frac{1}{2}$ holds whenever $b(U_J\mathcal{X})/b(\mathcal{X}) \leq \frac{1}{2}$ for any sub-string J of I. The impact of this approach on proving Lemma 24 is that if we can show, for example, that $b(U_{44}\mathcal{X})/b(\mathcal{X}) \leq \frac{1}{2}$, then it would immediately follow that $b(U_I\mathcal{X})/b(\mathcal{X}) \leq \frac{1}{2}$ for $I = 441, 444, 144, 244, 344\ldots$.

Lemma 25. *Let* $I \in \mathcal{I}_n$ *be of the form* $I = \{I_1, I_2\}$, *where* $I_1 \in \mathcal{I}_{n-k}$ *and* $I_2 \in \mathcal{I}_k$. *Then it holds that:* $\tilde{\mathcal{B}}_I \subseteq \tilde{\mathcal{B}}_{I_1}$ *and* $U_{I_1}(\tilde{\mathcal{B}}_I) \subseteq \tilde{\mathcal{B}}_{I_2}$.

Proof. If $\mathcal{X} \in \tilde{\mathcal{B}}_I$, then there exists a function f such that $I = Q_n(f, \mathcal{X})$. Since, $I = \{I_1, I_2\}$, then $I_1 = Q_{n-k}(f, \mathcal{X})$ by definition of Q. Hence $\mathcal{X} \in \tilde{\mathcal{B}}_{I_1}$, meaning $\mathcal{X} \in \tilde{\mathcal{B}}_I \Rightarrow \mathcal{X} \in \tilde{\mathcal{B}}_{I_1}$. Moreover, $I_2 = Q_k(f, U_{I_1}\mathcal{X})$ by Lemma 21, which implies $U_{I_1}\mathcal{X} \in \tilde{\mathcal{B}}_{I_2}$. □

Lemma 26. *Let $\mathcal{X}$ be an extended bracket of f. Given $I \in \mathcal{I}_n$, and $J \in \mathcal{S}(I)$, it holds that:*

$$\max_{\mathcal{X} \in \tilde{\mathcal{B}}_I} \frac{b(U_I\mathcal{X})}{b(\mathcal{X})} \leq \max_{\mathcal{X} \in \tilde{\mathcal{B}}_J} \frac{b(U_J\mathcal{X})}{b(\mathcal{X})}.$$

Proof. Since J is a sub-string of I, it follows that we might write I in the form: $I = \{I_1, J, I_2\}$. This implies that $U_I = U_{I_2}U_JU_{I_1}$. Next we have:

$$\frac{b(U_I\mathcal{X})}{b(\mathcal{X})} = \frac{b(U_{I_2}U_JU_{I_1}\mathcal{X})}{b(\mathcal{X})} = \frac{b(U_{I_2}U_JU_{I_1}\mathcal{X})}{b(U_JU_{I_1}\mathcal{X})} \frac{b(U_JU_{I_1}\mathcal{X})}{b(U_{I_1}\mathcal{X})} \frac{b(U_{I_1}\mathcal{X})}{b(\mathcal{X})}.$$

Now we use the fact that $b(U_{I_2}U_JU_{I_1}\mathcal{X})/b(U_JU_{I_1}\mathcal{X}) \leq 1$ and $b(U_{I_1}\mathcal{X})/b(\mathcal{X}) \leq 1$ to get:

$$\max_{\mathcal{X} \in \tilde{\mathcal{B}}_I} \frac{b(U_I\mathcal{X})}{b(\mathcal{X})} \leq \max_{\mathcal{X} \in \tilde{\mathcal{B}}_I} \frac{b(U_JU_{I_1}\mathcal{X})}{b(U_{I_1}\mathcal{X})} = \max_{\boldsymbol{s} \in \tilde{\mathcal{B}}_{\{J,I_2\}}} \frac{b(U_J\boldsymbol{s})}{b(\boldsymbol{s})} \leq \max_{\boldsymbol{s} \in \tilde{\mathcal{B}}_J} \frac{b(U_J\boldsymbol{s})}{b(\boldsymbol{s})}, \tag{5.48}$$

where the last two steps follow from Lemma 25. □

The next step we take towards proving Lemma 24 is to show that $\mathcal{I}_n$ is smaller than $\{1,2,3,4\}^n$. This would greatly reduce the number of sequences which we'd need to check in order to confirm Lemma 24.

Lemma 27. *Let $I = \{i_j\}_{j=0}^{n-1} \in \mathcal{I}_n$ be a sequence of EUPM iterations. If $i_j = 1$ then $i_{j+1} \in \{1,4\}$. Similarly, if $i_j = 2$ then $i_{j+1} \in \{2,3\}$.*

Proof. If $i_j = 1$, then by definition $\mathcal{X}_{j+1} = U_1\mathcal{X}_j$. From this, we calculate:

$$\begin{aligned}
\sigma_E(\mathcal{X}_{j+1}) - x^M_{j+1} &= \frac{x^L_{1,j+1}x^L_{2,j+1} - x^L_{1,j+1}x^L_{2,j+1}}{x^L_{1,j+1} + x^L_{2,j+1} - x^L_{1,j+1} - x^L_{2,j+1}} - x^M_{j+1}, \\
&= \frac{x^L_{1,j}x^M_j - x^L_{1,j}x^L_{2,j}}{x^L_{1,j} + x^M_j - x^L_{1,j} - x^L_{2,j}} - \frac{x^L_{1,j}x^L_{2,j} - x^L_{1,j}x^L_{2,j}}{x^L_{1,j} + x^L_{2,j} - x^L_{1,j} - x^L_{2,j}}, \\
&= \frac{-(x^L_{2,j} - x^M_j)(x^L_{1,j} - x^L_{1,j})(x^L_{1,j} - x^L_{2,j})}{(x^M_j + x^L_{1,j} - x^L_{1,j} - x^L_{2,j})(x^L_{1,j} + x^L_{2,j} - x^L_{1,j} - x^L_{2,j})} < 0,
\end{aligned}$$

since each $\mathcal{X}_j$ is an extended bracket. Given that $\sigma_E(\mathcal{X}_{j+1}) - x^M_{j+1} < 0$, it follows from Equation (5.18) that $i_{j+1} \in \{1, 4\}$. The equivalent argument applies for when $i_j = 2$. □

Now we know that no sequence containing the sub-string $12, 13, 21$ or 24 can be an element of $\mathcal{I}_n$. In the process of proving this, one may notice a symmetry between the functions U_1, U_4 and U_2, U_3 respectively. In the lemmas which follow, we derive a relation between U_1, U_4 and U_2, U_3 respectively to show that if $b(U_I\mathcal{X})/b(\mathcal{X}) \leq \frac{1}{2}$ holds for some I, it will still hold after replacing U_1 and U_4 with U_2 and U_3 and vice versa.

Definition 5.15. *Let $I \in \mathcal{I}_n$ be a sequence of EUPM iterations. We define $X : \mathbb{R}^5 \to \mathbb{R}^5$ to be the function which satisfies $X(a, b, c, d, e) = -(e, d, c, b, a)$.*

We make use of the function X because of the following useful properties.

Lemma 28. *It holds that $XX\mathcal{X} = \mathcal{X}$, $\sigma_E(X\mathcal{X}) = -\sigma_E(\mathcal{X})$, $b(X\mathcal{X}) = b(\mathcal{X})$, $U_1\mathcal{X} = XU_2X\mathcal{X}$, $U_2\mathcal{X} = XU_1X\mathcal{X}$, $U_3\mathcal{X} = XU_4X\mathcal{X}$ and $U_4\mathcal{X} = XU_3X\mathcal{X}$.*

Proof. Let $\mathcal{X} = (a, b, c, d, e)$. It follows that:

$$
\begin{aligned}
XX\mathcal{X} &= XX(a, b, c, d, e) = -X(e, d, c, b, a) = (a, b, c, d, e) = \mathcal{X}, \\
\sigma_E(X\mathcal{X}) &= \sigma_E(-e, -d, -c, -b, -a) = -\frac{ed - ab}{d + e - a - b} = -\sigma_E(a, b, c, d, e) = -\sigma_E(\mathcal{X}), \\
b(X\mathcal{X}) &= b(X(a, b, c, d, e)) = b(-e, -d, -c, -b, -a) = d - b = b(\mathcal{X}).
\end{aligned}
$$

$$XU_2X\mathcal{X} = XU_2\begin{pmatrix} -e \\ -d \\ -c \\ -b \\ -a \end{pmatrix} = X\begin{pmatrix} -d \\ -c \\ \frac{ab-ed}{d+e-a-b} \\ -b \\ -a \end{pmatrix} = \begin{pmatrix} a \\ b \\ \frac{ed-ab}{d+e-a-b} \\ c \\ d \end{pmatrix} = U_1\mathcal{X},$$

$$XU_3X\mathcal{X} = XU_3\begin{pmatrix} -e \\ -d \\ -c \\ -b \\ -a \end{pmatrix} = X\begin{pmatrix} -e \\ -d \\ -c \\ -\frac{ed-ab}{e+d-a-b} \\ -b \end{pmatrix} = \begin{pmatrix} b \\ \frac{ed-ab}{e+d-a-b} \\ c \\ d \\ e \end{pmatrix} = U_4\mathcal{X}.$$

□

Having shown that the function X relates U_1 to U_4 and U_4 to U_3, we now show that this relation also holds for sequences of iterations.

Definition 5.16. *Let $I = \{i_j\}_{j=1}^n \in \mathcal{I}_n$. Define $W : \mathcal{I}_n \to \mathcal{I}_n$ to be the function: $W(I) = \{w(i_j)\}_{j=1}^n$, where $w(1) = 2, w(2) = 1, w(3) = 4$ and $w(4) = 3$.*

Lemma 29. *If f and g are functions such that $g(x) = f(-x)$ and $\mathcal{X}_0 \in \mathcal{B}_f$, then $X\mathcal{X} \in \mathcal{B}_g$. Moreover, if $\{\mathcal{X}_i^f\}_{i=0}^n$ and $\{\mathcal{X}_i^g\}_{i=0}^n$ are the sequences of extended brackets generated by the EUPM when applied to f and g starting from $\mathcal{X}_0$ and $X\mathcal{X}_0$ respectively, then it holds that $\mathcal{X}_j^f = X\mathcal{X}_j^g$ for every $j = 0, 1, \ldots, n$ and $Q_n(g, X\mathcal{X}) = W(Q_n(f, \mathcal{X}))$.*

Proof. Note that $X\mathcal{X} \in \mathcal{B}_g$ follows from Definitions 5.6 and 5.15. Let $\{\mathcal{X}_i^f\}_{i=0}^n$ and $\{\mathcal{X}_i^g\}_{i=0}^n$ be the sequences of extended brackets generated by applying the EUPM with the starting brackets $\mathcal{X}_0$ and $X\mathcal{X}_0$ to f and g respectively. Define:

$$\varphi(\mathcal{X}) = \sigma_E(\mathcal{X}) - x^M. \tag{5.49}$$

Since $\sigma_E(\mathcal{X}) = -\sigma_E(X\mathcal{X})$, it follows that $f(\sigma_E(\mathcal{X}_0)) = g(\sigma_E(X\mathcal{X}_0))$ and $\varphi(\mathcal{X}_0) < 0 \Leftrightarrow \varphi(X\mathcal{X}_0) > 0$. Given that $sign(\varphi(\mathcal{X}))$ is precisely the condition in Equation (5.18)

which determines whether to apply update functions U_1, U_4 or U_2, U_3, it follows that $\mathcal{X}_1^f = U_j \mathcal{X}_0^f = U_j \mathcal{X}_0 \Leftrightarrow \mathcal{X}_1^g = U_{w(j)} \mathcal{X}_0^g = U_{w(j)} X \mathcal{X}_0$.

Applying Lemma 28, we note that $U_{w(j)} X \mathcal{X}_0 = X U_j \mathcal{X}_0$ which in turn means $\mathcal{X}_1^g = X \mathcal{X}_1^f$. From this it follows inductively that $\mathcal{X}_j^f = X \mathcal{X}_j^g$ for all $j = 1, 2, \ldots, n$. Moreover, if we write $Q_n(f, \mathcal{X})$ and $Q_n(g, X\mathcal{X})$ in the forms $\{i_j^f\}_{j=0}^{n-1}$ and $\{i_j^g\}_{j=0}^{n-1}$ respectively, it also follows inductively that $i_j^f = w(i_j^g)$ for each $j = 0, 1, \ldots, n-1$; in other words $Q_n(g, X\mathcal{X}) = W(Q_n(f, \mathcal{X}))$. □

Corollary 5.2. *If $I \in \mathcal{I}_n$, then: $X(\tilde{\mathcal{B}}_I) = \tilde{\mathcal{B}}_{W(I)}$ (recall Definition 5.13).*

Proof. If $\mathcal{X} \in \tilde{\mathcal{B}}_I$, then there exists a function f such that $I = Q_n(f, \mathcal{X})$ and $\mathcal{X} \in \mathcal{B}_f$. By defining $g(x) := f(-x)$, it follows from Definition 5.6 that: $\mathcal{X} \in \mathcal{B}_f \Leftrightarrow X\mathcal{X} \in \mathcal{B}_g$. Moreover by Lemma 29, $Q_n(g, X\mathcal{X}) = W(Q_n(f, \mathcal{X})) = W(I)$. Since g satisfies $Q_n(g, X\mathcal{X}) = W(I)$, it follows from Definition 5.13 that $X\mathcal{X} \in \tilde{\mathcal{B}}_{W(I)}$. □

Corollary 5.3. *For any $\mathcal{X}$, it holds that $U_I \mathcal{X} = X U_{W(I)} X \mathcal{X}$, $\forall I \in \mathcal{I}_n$.*

Proof. Let f be a function such that $I = Q(f, \mathcal{X})$. By Lemma 29, the function $g(x) = f(-x)$ has the property that $Q_n(g, X\mathcal{X}) = W(Q_n(f, \mathcal{X}))$. Furthermore $U_I(\mathcal{X}) = U_{Q_n(f,\mathcal{X})}(\mathcal{X}) = U_{Q_n(f,\mathcal{X})}(XX\mathcal{X}) = X U_{Q_n(g,X\mathcal{X})}(X\mathcal{X}) = X U_{W(I)}(X\mathcal{X})$. □

The map W serves as a natural bijection between sequences of EUPM iterations which start with 1 or 4 and those which start with 2 or 3. Dividing $\mathcal{I}_5$ is useful due to the following result.

Corollary 5.4. *Given $I \in \mathcal{I}_n$ a sequence of EUPM iterations, it holds that:*

$$\max_{\mathcal{X} \in \tilde{\mathcal{B}}_I} \frac{b(U_I \mathcal{X})}{b(\mathcal{X})} = \max_{\mathcal{X} \in \tilde{\mathcal{B}}_{W(I)}} \frac{b(U_{W(I)} \mathcal{X})}{b(\mathcal{X})}$$

Proof. Using Lemma 28 and Corollaries 5.2 and 5.3 it follows that:

$$\max_{\mathcal{X} \in \tilde{\mathcal{B}}_I} \frac{b(U_I \mathcal{X})}{b(\mathcal{X})} = \max_{\mathcal{X} \in \tilde{\mathcal{B}}_I} \frac{b(X U_{W(I)} X \mathcal{X})}{b(\mathcal{X})} = \max_{\mathcal{X} \in \tilde{\mathcal{B}}_I} \frac{b(U_{W(I)} X \mathcal{X})}{b(X\mathcal{X})} = \max_{\boldsymbol{s} \in \tilde{\mathcal{B}}_{W(I)}} \frac{b(U_{W(I)} \boldsymbol{s})}{b(\boldsymbol{s})}.$$

□

We are now equipped to prove Lemma 24.

Proof of Lemma 24. To prove this lemma, we list out elements of $\mathcal{I}_3, \mathcal{I}_4$ and $\mathcal{I}_5$ which begin with either 1 or 4 and mark them in the following way. If a sequence is contained in $\mathcal{A}$, we overline it. If a sequence contains a sub-string which is contained in $\mathcal{A}$, we underline the relevant sub-string. Finally, if a sequence contains a sub-string J such that $W(J) \in \mathcal{A}$, then we place square brackets around J. We will show that all elements of $\mathcal{I}_5$ can be marked in one of these three ways. There is no need to list the sequences starting with 2 or 3 because each of those sequences is merely $W(J)$ for some J listed below.

$\overline{111}$	141	$\overline{143}$	411	$\overline{422}$	431	4[33]	$\underline{44}1$	$\underline{44}3$
114	142	$1\underline{44}$	$\overline{414}$	423	432	$\overline{434}$	$\underline{44}2$	$\underline{444}$

We see that only 7 (or 14 if you count the equivalent sequences starting with 2 or 3) of the sequences above are unmarked. We take these seven sequences, and list all elements of $\mathcal{I}_4$ which begin with any of these 7 sub-strings. We neglect to list out elements of $\mathcal{I}_4$ which begin with a marked sub-string because these elements are themselves guaranteed to be marked .

$\overline{1141}$	$1\underline{143}$	$\overline{1411}$	$1\underline{422}$	$4\underline{111}$	4231	42[33]	4[311]	$\overline{4322}$
1142	$1\underline{144}$	$1\underline{414}$	$\overline{1423}$	$\overline{4114}$	4232	4[234]	$\overline{4314}$	4[323]

Using the same procedure as earlier, we see only 3 sequences are left unmarked. We list out all elements of $\mathcal{I}_5$ which begin with one of these three sequences and note that all of them are marked.

$$11\underline{422} \qquad 42\underline{311} \qquad 4[2322] \qquad 1\underline{1423} \qquad 4[2314] \qquad 42[323].$$

If a sequence J is marked in any of these ways, then it follows from Lemma 26 and

Corollary 5.4 that $b(U_I\mathcal{X})/b(\mathcal{X}) \leq \frac{1}{2}\ \forall I \in \mathcal{A}$ implies $b(U_J\mathcal{X})/b(\mathcal{X}) \leq \frac{1}{2}$. □

5.3.3.3 Numerical performance of the EUPM

In Section 5.3.3.1 we bounded the convergence rate over 5 iterations of the EUPM. In practice we observe far superior convergence rates. From our observation, this seems to be because the extremal values of $\mathcal{X}_0$ which yield the worst convergence rate over 5 iterations tend to yield best case performance over 6 iterations and so on. In this section, we show what performance might realistically be expected from the EUPM.

In order to construct a suitable experiment, we first must ask what factors affect the convergence of the EUPM? The answer to this question is the function to minimise f, along with the initial value of $\mathcal{X}_0$. However, the only effect that the function f actually has is to determine whether $f(\tilde{x}) < f(x^M)$. This along with the current value of $\mathcal{X}_0$ uniquely determines which update function will be used. Therefore, when testing the convergence rate of the EUPM, we do not test it on a range of functions, but rather for different binary sequences, where 1's mean $f(\tilde{x}) < f(x^M)$ and 0's the opposite.

The main advantage of this is that while the set $C([a,b])$ has infinite cardinality, the set of binary sequences of length n is finite. Therefore, given an initial bracket $\mathcal{X}_0$, it is possible to determine how the EUPM will perform on literally any function. Given $\mathcal{X}_0$ and $I \in \{0,1\}^n$, we plot the average convergence rate of the EUPM.

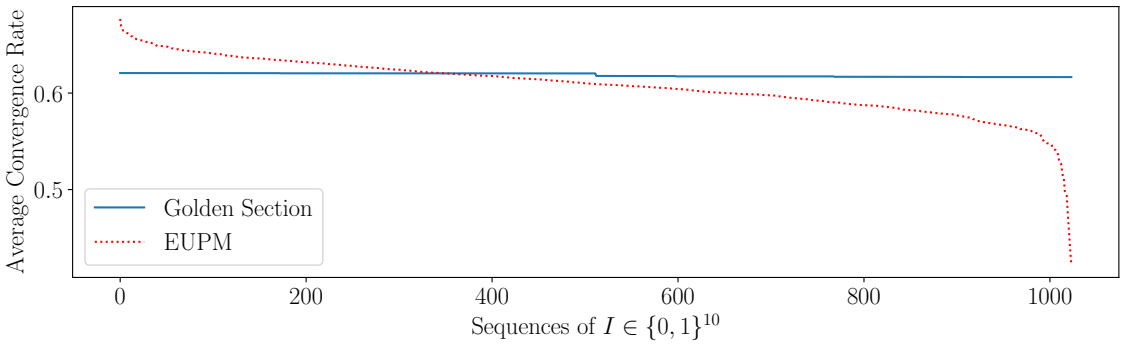

Figure 5.3.6 – We compare the convergence rate of the EUPM with Golden Section over all possible sequences $I \in \{0,1\}^{10}$. These sequences are ordered such that the slowest ones are plotted first. For each sequences, we sample a large number of values for $\mathcal{X}_0$ and average over these.

For each sequence $I \in \{0,1\}$, we sample a large number of $\mathcal{X}_0$ values, constrained such that $b(\mathcal{X}_0) = 1$, and average over these. This leaves us with an average convergence rate for every possible sequence in $\{0,1\}$, and by extension for every possible function. We choose $n = 10$ and plot the resulting data corresponding to both the EUPM and Golden Section in Figure 5.3.6.

What we see is that Golden Section is the more conservative of the two. It performs notably better in the worst case scenario even if notably worse in the best case scenario. Even though we see some convergence rates from the EUPM which are significantly worse than those observed in Tables 5.3.1 to 5.3.3, they still are far better than the upper bound from Theorem 22.

5.3.4 Dynamic underestimating polynomial method

In Section 5.3, we identified three main weaknesses of the SUPM: that of choosing a suitable α, and that of ensuring that both $x^R_{1,i}$ and $x^L_{1,i}$ converge to x^* fast enough. In this section, we introduce a heuristic which makes use of the features of the SUPM, while being equipped with a safeguarding option designed to avoid the situations where the SUPM fails. We call this heuristic the dynamic underestimating polynomial method (DUPM), and denote the step and update functions for the DUPM by σ_D and U_D respectively.

The effect that α has on σ_S lies in how the model functions q^L and q^R (see Equation (5.16)) are constructed. Since α is multiplied to $h(\mathcal{X})$, and $h(\mathcal{X}) \to 0$ should occur as the algorithm converges, a finite but excessively large value of α should not prevent $\alpha h(\mathcal{X}) \to 0$ eventually. However, the SUPM may stall temporarily when α is too small. Therefore, we construct the DUPM such that it will increase α when necessary, but never decrease it.

In order for Theorem 20 to apply, we need α to satisfy Lemma 18. While there is no way to check this, a necessary condition for Lemma 18 to apply is that $f(x^M) \geq q^L(x^M)$ and $f(x^M) \geq q^R(x^M)$. Using a method similar to the proof of Lemma 18, we find that

this is equivalent to:

$$\alpha \geq \frac{1}{h(\mathcal{X})} \max_{k\in\{L,R\}} \left(f[x_1^k, x_2^k, x_3^k] - f[x^M, x_1^k, x_2^k] \right). \tag{5.50}$$

Therefore, at each iteration of the DUPM we set $\alpha_{i+1} \geq \max(\alpha_i, \alpha^*)$ where α^* is the smallest value α which satisfies Equation (5.50).

Next we wish to force the DUPM to update both $x_{1,i}^L$ and $x_{1,i}^R$ regularly. In order to do this, we must first equip the DUPM to recognize when this occurs. We append the three variables u_1, u_2, u_3, whose purpose is record whether the update function U_D updated x^L or x^R during each of the last three iterations, to the extended bracket $\mathcal{X}$. The update function U_D is defined to be the natural extension of U_S (see Equation (5.18)) which includes u_1, u_2 and u_3:

$$U_D(\tilde{x}; \mathcal{X}, \alpha) = \begin{cases} (U_1(\tilde{x}; \mathcal{X}), R, u_1, u_2) & \text{if} \quad \tilde{x} < x^M \text{ and } f(\tilde{x}) < f(x^M) \\ (U_2(\tilde{x}; \mathcal{X}), L, u_1, u_2) & \text{if} \quad \tilde{x} > x^M \text{ and } f(\tilde{x}) < f(x^M) \\ (U_3(\tilde{x}; \mathcal{X}), R, u_1, u_2) & \text{if} \quad \tilde{x} > x^M \text{ and } f(\tilde{x}) > f(x^M) \\ (U_4(\tilde{x}; \mathcal{X}), L, u_1, u_2) & \text{if} \quad \tilde{x} < x^M \text{ and } f(\tilde{x}) > f(x^M) \end{cases}, \tag{5.51}$$

where the functions $U_1, \ldots, U_4$ are as defined in Equation (5.19).

If $u_1 = u_2 = u_3$, then this means that the DUPM has updated either $x_{1,i}^L$, or $x_{1,i}^R$ for each of the last three iterations. When this occurs, we force the DUPM to take an EUPM step instead, $\sigma_D(\mathcal{X}; \alpha) = \sigma_E(\mathcal{X})$. This interference is similar to the fall back option used by Brent's method [79, 80]. The choice to interfere after three iterations is based on practical experience, and not on any theoretical insight.

Finally, we define one more condition under which we interfere with the SUPM. From practical experience, we observe situations where either q^L or q^R is a convex function whose local minimum is returned by σ_S, which result in slow convergence. As a response to this, we ensure that α is sufficiently big that σ_D returns a point of

intersection between q^L and q^R. More rigorously, we require α such that

$$q^L\left(\sigma_S(\mathcal{X};\alpha);\mathcal{X},\alpha\right) = q^R\left(\sigma_S(\mathcal{X};\alpha);\mathcal{X},\alpha\right) \tag{5.52}$$

holds, where σ_S is the step function for the SUPM (see Equation (5.17)).

Define $\chi(\mathcal{X})$ to be the the smallest value of α such that Equation (5.52) holds for all $\alpha \geq \chi(\mathcal{X})$. Then at each iteration of the DUPM, we require $\alpha_{i+1} \geq \max(\alpha_i, \chi(\mathcal{X}))$. In order to compute $\chi(\mathcal{X})$, note Equation (5.52) is satisfied whenever q^L and q^R are concave functions, provided that Equation (5.50) holds. Define α^+ to be $\max_k f[x_1^k, x_2^k, x_3^k]/h(\mathcal{X})$, $k \in \{L, R\}$. Then it follows that for all $\alpha > \alpha^+$, Equation (5.52) is satisfied, implying that $\chi(\mathcal{X}) \leq \alpha^+$.

If at iteration i, α_i satisfies Equation (5.52), then there is no need to compute $\chi(\mathcal{X})$. Otherwise, we know that $\chi(\mathcal{X})$ lies in the interval $(\alpha_i, \alpha^+]$. Therefore, we can apply bisection to compute $\chi(\mathcal{X})$ to any desired accuracy.

To define the DUPM rigorously, we must insert an additional line into Algorithm 12. Before computing σ_D, we set

$$\alpha_{i+1} = \max\left(\alpha_i, \chi(\mathcal{X}_i), \frac{1}{h(\mathcal{X})}\max_{k\in\{L,R\}}\left(f[x_1^k, x_2^k, x_3^k] - f[x^M, x_1^k, x_2^k]\right)\right).$$

Finally we define

$$\sigma_D(\mathcal{X};\alpha) := \begin{cases} \text{if } u_1 = u_2 = u_3 & \sigma_E(\mathcal{X}) \\ \text{else} & \sigma_S(\mathcal{X};\alpha) \end{cases}. \tag{5.53}$$

Ultimately the purpose of these interferences is to ensure that Theorem 20 applies as soon as possible while forcing both $x_{1,i}^L$ and $x_{1,i}^R$ to be updated.

Remark. *When running the DUPM in practice, one sometimes encounters values of $\chi(\mathcal{X})$ which are very large $\sim 10^{10}$. If defined as above, the DUPM will update α to be this enormous number, thus resulting in the DUPM behaving like the EUPM. We have found that better performance is encountered if we select a maximum value of α, such that we only update α to be a number less than this bound. Whenever α would otherwise*

	SUPM with parameter value				Other methods			
Functions	$\alpha = 0$	$\alpha = 0.1$	$\alpha = 1$	$\alpha = 10$	EUPM	DUPM	Brent	MSA
f_1^{SU}	0.3268	0.4273	0.4912	0.5472	0.6192	0.48	0.2159	0.1477
f_2^{SU}	0.68	0.6164	0.6183	0.6187	0.6188	0.6182	0.3282	0.6614
f_3^{SU}	∞	0.4848	0.5438	0.5882	0.6349	0.5293	0.3329	∞
f_4^{SU}	∞	0.6105	0.6104	0.6104	0.6107	0.6103	0.5626	∞
f_5^{SU}	∞	0.5469	0.5925	0.6233	0.6323	0.4854	0.2756	0.2581
f_6^{SU}	∞	0.5871	0.615	0.6246	0.6249	0.5569	0.2812	∞
f_7^{SU}	0.35	0.4966	0.5521	0.5944	0.6353	0.4885	0.2575	∞
f_1^{NU}	∞	0.1868	0.2684	0.3272	0.6188	0.264	0.5115	∞
f_2^{NU}	∞	0.3999	0.4445	0.4806	0.6265	0.427	0.6337	∞
f_3^{NU}	∞	0.4538	0.4931	0.5255	0.6413	0.4421	0.6457	0.3259
f_4^{NU}	∞	0.4153	0.4665	0.5006	0.6204	0.4051	0.5934	0.3378
f_5^{NU}	∞	0.4204	0.4645	0.4974	0.6195	0.4142	0.4903	0.3592
f_1^{SM}	∞	0.6146	0.6138	0.6144	0.6149	0.6224	0.4828	∞
f_2^{SM}	∞	0.5076	0.5579	0.5961	0.6183	0.486	0.3588	∞
f_3^{SM}	∞	0.5481	0.5918	0.6177	0.6234	0.5085	0.3372	∞
f_4^{SM}	∞	0.4536	0.5308	0.5789	0.6338	0.5282	0.2703	∞
f_5^{SM}	∞	0.4704	0.5188	0.5582	0.6165	0.5143	0.311	∞
f_6^{SM}	∞	0.5599	0.5962	0.6191	0.624	0.5353	0.2982	∞
f_7^{SM}	∞	0.4733	0.5353	0.5776	0.6224	0.4815	0.273	∞

Table 5.3.4 – Average convergence rate of different algorithms on the set of smooth multimodal test functions.

become greater than this bound, we leave α unchanged, and apply the EUPM update instead.

5.3.5 Numerical performance of the DUPM

Having defined the DUPM, and justified parts of it based on observed results, all that remains is to show these results. We present a comparison of the DUPM vs Brent's method, Golden Section, the EUPM, SUPM and MSA in Table 5.3.4, using the same format as in Tables 5.3.1 to 5.3.3.

There are a few points to note from this table. First and foremost, the DUPM performs at worst at the rate of Golden Section, but usually better. Two of the three particular cases where the DUPM only matches Golden Section corresponds to the test-functions where f'' and $f^{(3)}$ also vanish at the minimiser. On non-smooth functions, the DUPM outperforms Brent's method, except for f_5^{NU}, where Brent's method converges

surprisingly fast. Finally, when comparing the DUPM with the SUPM, the DUPM converges at a rate between the SUPM for the two best values of α, while it never fails to converge.

5.3.6 Using the DUPM within Algorithm 8

In Section 4.1.2, we defined a LS algorithm which relies on calling an external univariate optimiser. Given that this motivated the derivation of the DUPM, we must ensure that all the conditions required by Algorithm 8 are satisfied. In particular, the univariate optimiser called by Algorithm 8 must

1. require no inputs other than a bracket,

2. be able to distinguish a smooth minimum from a NS minimum.

To resolve the first of these issues, we implement the algorithm such that the DUPM will run Golden Section, until it has constructed a length 5 extended bracket. From this point it will apply the EUPM, until finding a length 7 extended bracket. Only at this point will the DUPM operate as described.

Regarding identifying a NS minimum, we use the points x_1^L, x_2^L, x_1^R and x_2^R to construct two slopes m_L and m_R. Then we check if f is continuously differentiable by investigating if: $|m_L - m_R| \leq L|x_2^R - x_2^L|$. Given a parameter value for L, this process can identify, subject to error, whether φ' is continuously differentiable near to the univariate optimiser.

5.4 Conclusions and future work

In this chapter , we have constructed a univariate optimization algorithm for black box, piece-wise smooth functions. As seen in Section 5.3.5, this new method, the DUPM, converges both more robustly and often faster than existing methods for such problems. Furthermore, while it is not the fastest algorithms for standard smooth test functions, it is not prohibitively slow either.

It must be acknowledged that the comparison between the UPM, the Mifflin-Strodiot method and Brent's method is not entirely fair given that they require a different number of points to start. Therefore, in the context of a line-search, we expect to start with a method like Golden Section, and switch to the DUPM once we have accumulated enough points to form and extended bracket. Regardless, the DUPM offers a univariate solver which performs well in most contexts and is therefore suitable for non-smooth functions.

However, we do not view the UPM algorithms to be a completed body of research. Indeed, we consider the following points to be valid avenues of further research which we have not managed to explore thoroughly:

1. Can the convergence rate bound for the EUPM be made sharper and thus better reflect its numerical performance observed in Figure 5.3.6.

2. Do we actually need to consider all possible EUPM sequences (recall Definition 5.10) to construct a general bound, or might there exist an alternative property of f which might refine this list?

3. What alternative approaches exists for modifying α for the DUPM?

4. Can we develop a convergence theory for the DUPM specifically?

Appendix A

Additional content for NSO

A.1 What does 3dv1d look like?

In Chapter 4, we chose to assume that the objective function corresponding the efficiency of a steam turbine was PS. At the time, we justified this by showing a single plot which illustrated the shape of 3dv1d when two particular variables were modified. In this section, we explain in greater detail how this plot was produced, in addition to proving a host of additional plots of this type. By doing so, we hope to strengthen the argument for treating 3dv1d as a PS function.

Recall from Section 1.3 that we have access to five different test functions. Only two of these are simultaneously large enough to be realistic while small enough not to be prohibitively expensive for numerical testing. Two others are unrealistically small. While the performance of optimisers on these test functions are of little interest to Siemens, we make use of them to assess the properties of 3dv1d. The fact that these functions are far cheaper computationally is a desirable feature for this.

Our approach in this section is as follows. We start by defining $\tilde{\boldsymbol{x}}$ to be the point at which e04ucf stalls. Since the smooth optimiser gets stuck here, we know there must be kinks present, making this an ideal location to investigate. Next, we loop over all possible pairs of variables, and plot the outcome using the format of Figure 4.0.1 from earlier. While we do not provide literally every one of these images here, we do show a

selection which captures all the phenomena observed.

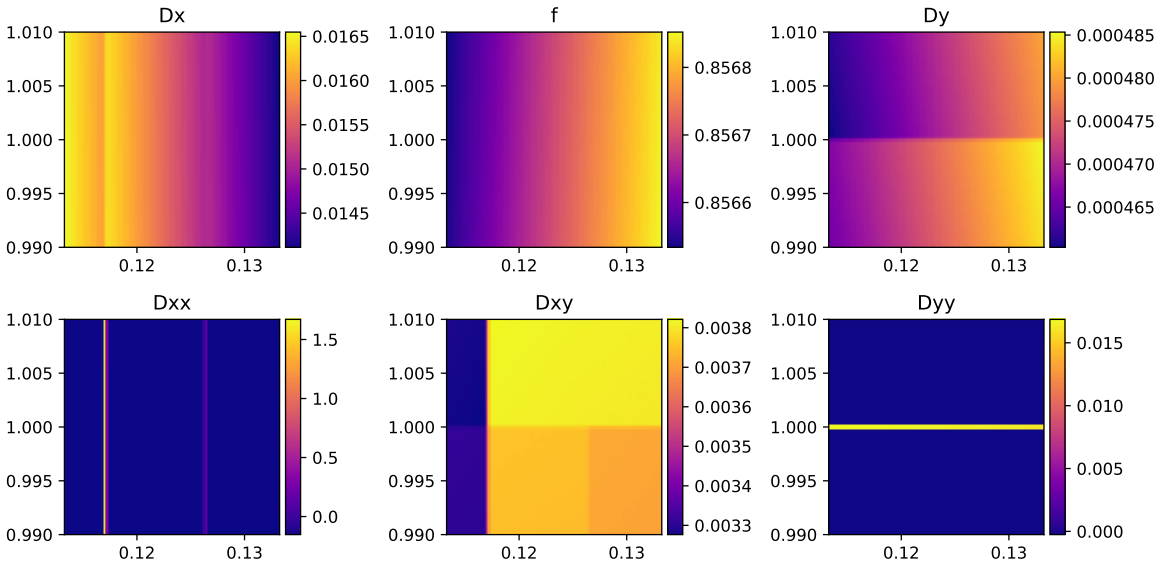

Figure A.1.1 – Starting from the value of $\boldsymbol{x}$ where e04ucf stalls, we modify $x^{(1)}$ (x axis) and $x^{(12)}$ (y axis) and plot the result to f, as well as the numerical approximations to the first and second derivatives.

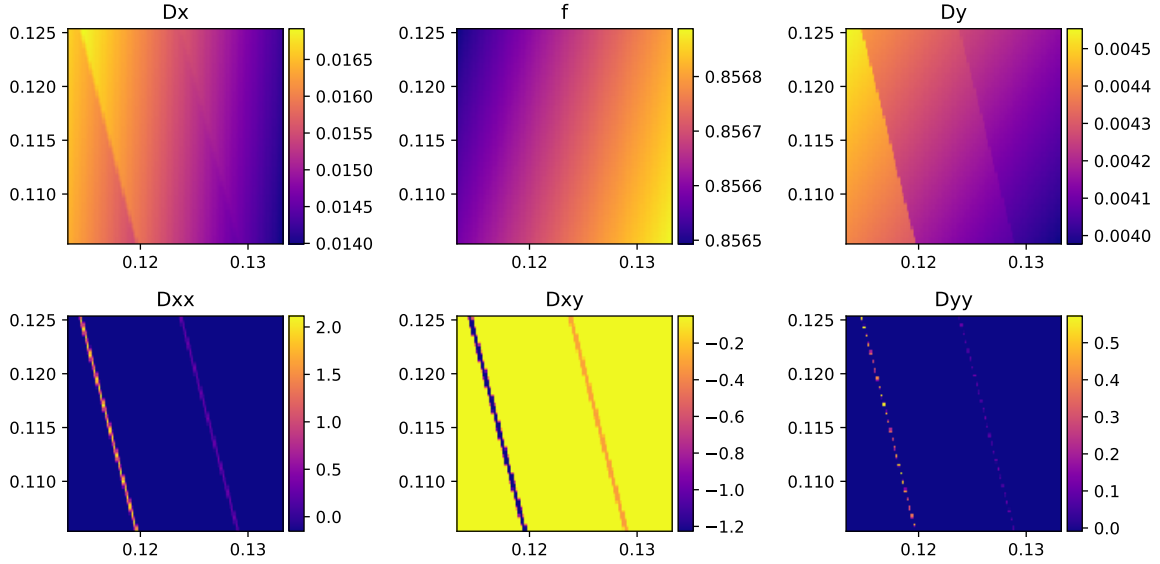

Figure A.1.2 – Starting from the value of $\boldsymbol{x}$ where e04ucf stalls, we modify $x^{(1)}$ (x axis) and $x^{(3)}$ (y axis) and plot the result to f, as well as the numerical approximations to the first and second derivatives.

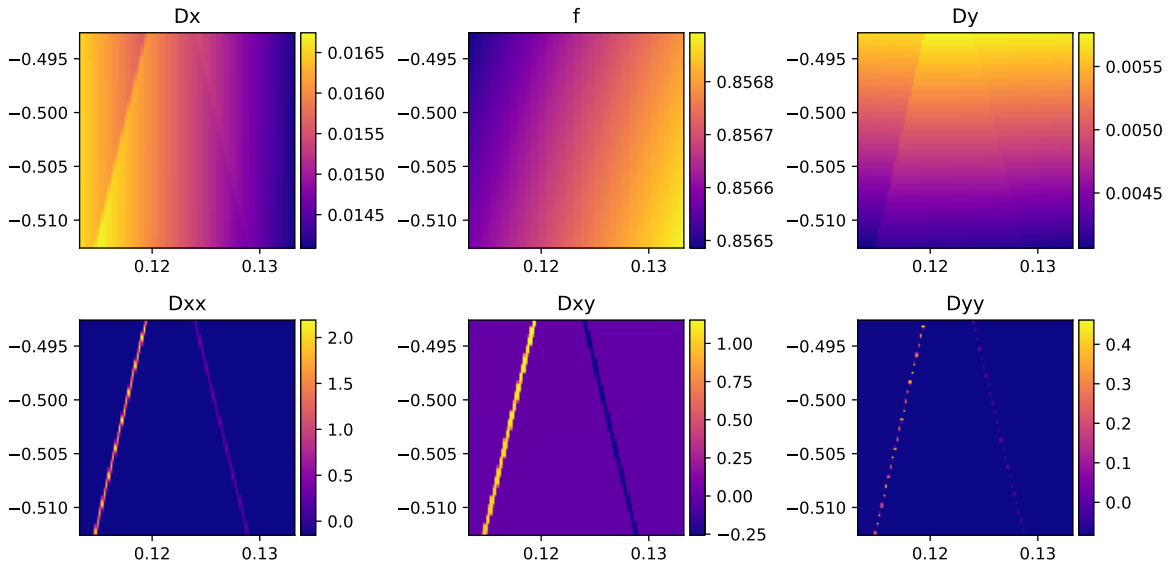

Figure A.1.3 – Starting from the value of $\boldsymbol{x}$ where e04ucf stalls, we modify $x^{(1)}$ (x axis) and $x^{(7)}$ (y axis) and plot the result to f, as well as the numerical approximations to the first and second derivatives.

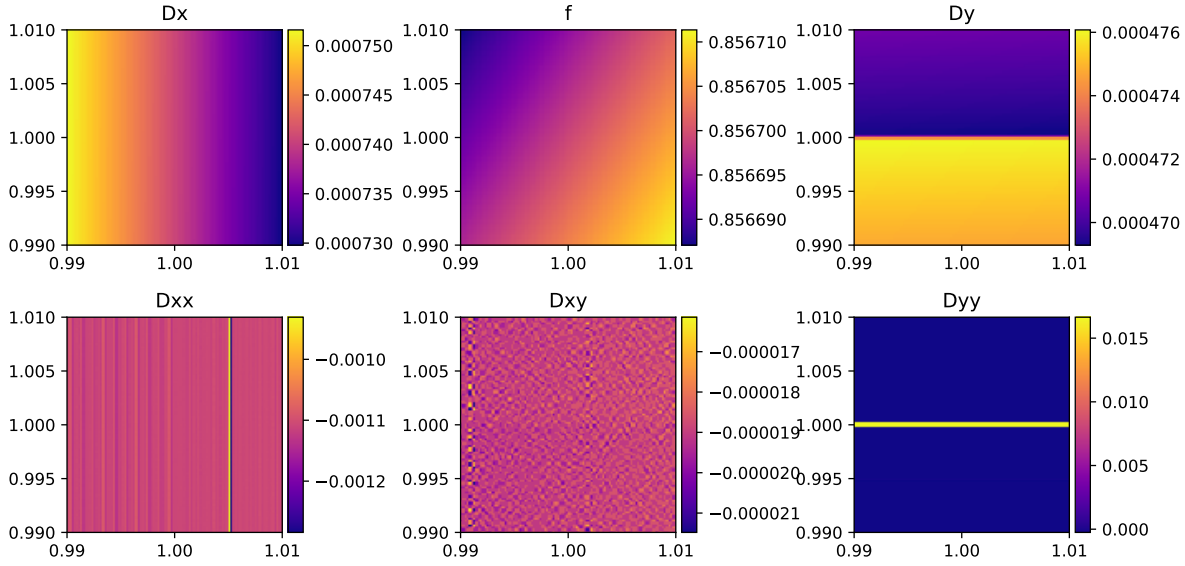

Figure A.1.4 – Starting from the value of $\boldsymbol{x}$ where e04ucf stalls, we modify $x^{(2)}$ (x axis) and $x^{(12)}$ (y axis) and plot the result to f, as well as the numerical approximations to the first and second derivatives.

A.2 A list of NS test functions

1. Generalization of MAXQ [65]

$$\underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad \max_{1 \le i \le n} x_i^2 \tag{A.1}$$

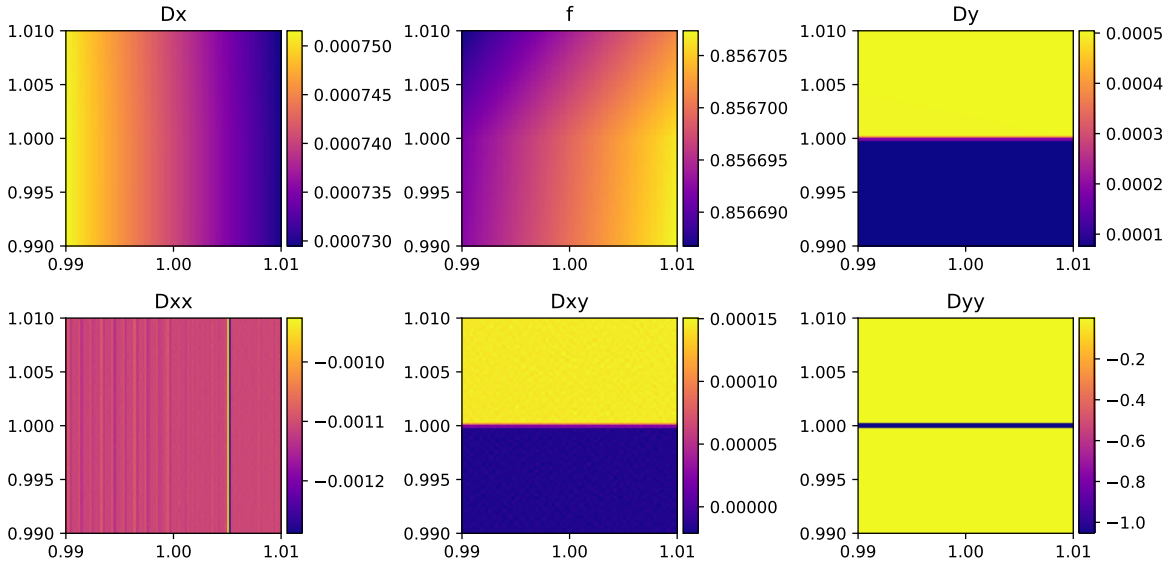

Figure A.1.5 – Starting from the value of $\boldsymbol{x}$ where e04ucf stalls, we modify $x^{(2)}$ (x axis) and $x^{(17)}$ (y axis) and plot the result to f, as well as the numerical approximations to the first and second derivatives.

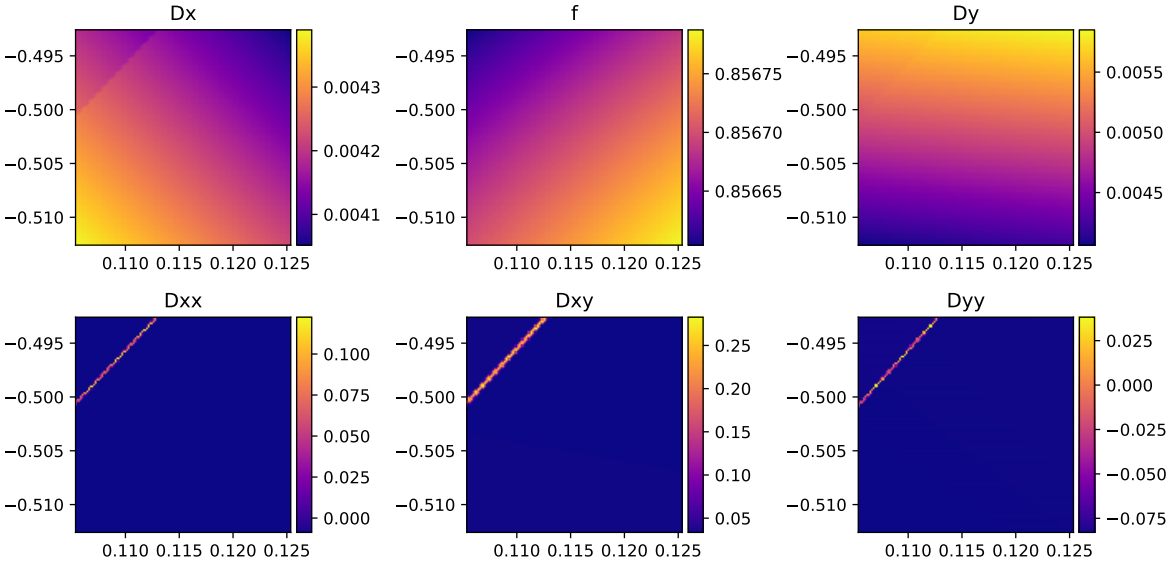

Figure A.1.6 – Starting from the value of $\boldsymbol{x}$ where e04ucf stalls, we modify $x^{(3)}$ (x axis) and $x^{(7)}$ (y axis) and plot the result to f, as well as the numerical approximations to the first and second derivatives.

starting from $\boldsymbol{y}$ where $y_i = i$ for $i = 1, 2, \ldots, n/2$ and $y_i = -i$ for $i = n/2 + 1 \ldots, n$.

For this function, it is known that $f(\boldsymbol{x}^*) = 0$.

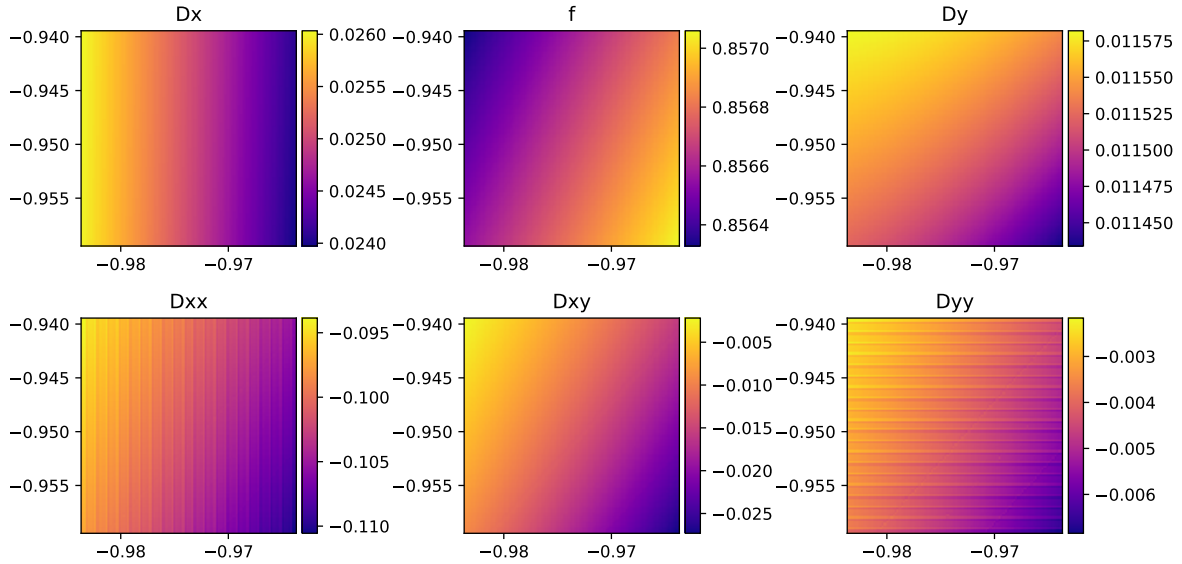

Figure A.1.7 – Starting from the value of $\boldsymbol{x}$ where e04ucf stalls, we modify $x^{(5)}$ (x axis) and $x^{(13)}$ (y axis) and plot the result to f, as well as the numerical approximations to the first and second derivatives.

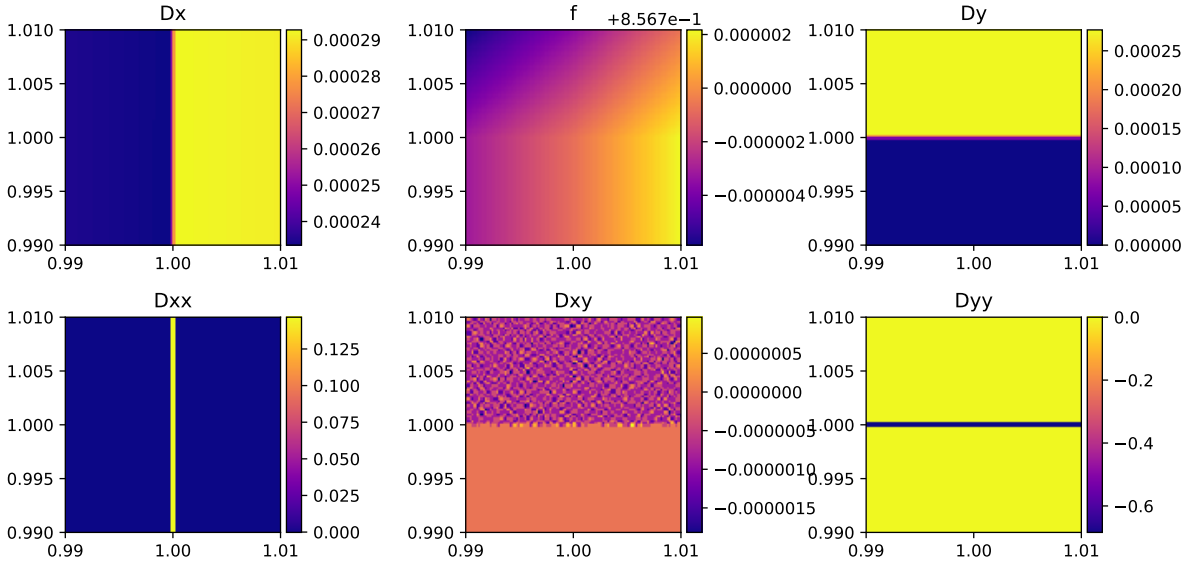

Figure A.1.8 – Starting from the value of $\boldsymbol{x}$ where e04ucf stalls, we modify $x^{(11)}$ (x axis) and $x^{(16)}$ (y axis) and plot the result to f, as well as the numerical approximations to the first and second derivatives.

2. Generalization of MXHILB [65]

$$\underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad \max_{1 \le i \le n} \left| \sum_{j=1}^{n} \frac{x_j}{i+j-1} \right| \tag{A.2}$$

starting from $\boldsymbol{y}$ where $y_i = i$ for $i = 1, 2, \ldots, n$. For this function, it is known that

$f(\boldsymbol{x}^*) = 0$.

3. Chained LQ [65]

$$\underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad \sum_{i=1}^{n-1} \max\left(-x_i - x_{i+1}, -x_i - x_{i+1} + (x_i^2 + x_{i+1}^2 - 1)\right) \tag{A.3}$$

starting from $\boldsymbol{y}$ where $y_i = -0.5$ for $i = 1, 2, \ldots, n$. For this function, it is known that $f(\boldsymbol{x}^*) = -(n-2)\sqrt{2}$.

4. Chained CB3 1 [65]

$$\underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad \sum_{i=1}^{n-1} \max\left(x_i^4 + x_{i+1}^2, (2 - x_i)^2 + (2 - x_{i+1})^2, 2\exp(-x_i + x_{i+1})\right) \tag{A.4}$$

starting from $\boldsymbol{y}$ where $y_i = 2$ for $i = 1, 2, \ldots, n$. For this function, it is known that $f(\boldsymbol{x}^*) = 2(n-1)$.

5. Chained CB3 2 [65]

$$\underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad \max\left(\sum_{i=1}^{n-1} x_i^4 + x_{i+1}^2, \sum_{i=1}^{n-1} (2 - x_i)^2 + (2 - x_{i+1})^2, \sum_{i=1}^{n-1} 2\exp(-x_i + x_{i+1})\right) \tag{A.5}$$

starting from $\boldsymbol{y}$ where $y_i = 2$ for $i = 1, 2, \ldots, n$. For this function, it is known that $f(\boldsymbol{x}^*) = 2(n-1)$.

6. Number of Active Faces [65]

$$\begin{aligned} \underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad & \max_{1 \le i \le n}\left(g\left(-\sum_{i=1}^{n} x_i\right), g(x_i)\right) \\ \text{subject to} \quad & g(y) = \ln(|y| + 1), \end{aligned} \tag{A.6}$$

starting from $\boldsymbol{y}$ where $y_i = 1$ for $i = 1, 2, \ldots, n$. For this function, it is known that $f(\boldsymbol{x}^*) = 0$.

7. NS generalization of Brown function 2 [65]

$$\underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad \sum_{i=1}^{n-1} \left(|x_i|^{x_{i+1}^2+1} + |x_{i+1}|^{x_i^2+1} \right) \tag{A.7}$$

starting from $\boldsymbol{y}$ where $y_i = 1$ when $\mod(i,2) = 0$ and $y_i = -1$ when $\mod(i,2) = 1$. For this function, it is known that $f(\boldsymbol{x}^*) = 0$.

8. Chained Mifflin 2 [65]

$$\underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad \sum_{i=1}^{n-1} \left(-x_i + 2(x_i^2 + x_{i+1}^2 - 1) + 1.75 \left| x_i^2 + x_{i+1}^2 - 1 \right| \right) \tag{A.8}$$

starting from $\boldsymbol{y}$ where $y_i = -1$ for $i = 1, 2, \ldots, n$.

9. Chained Crescent 1 [65]

$$\underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad \max \left(\sum_{i=1}^{n-1} (x_i^2 + (x_{i+1} - 1)^2 + x_{i+1} - 1), \sum_{i=1}^{n-1} (-x_i^2 - (x_{i+1} - 1)^2 + x_{i+1} - 1) \right) \tag{A.9}$$

starting from $\boldsymbol{y}$ where $y_i = 2$ when $\mod(i,2) = 0$ and $y_i = -1.5$ when $\mod(i,2) = 1$. For this function, it is known that $f(\boldsymbol{x}^*) = 0$.

10. Chained Crescent 2 [65]

$$\underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad \sum_{i=1}^{n-1} \max \left((x_i^2 + (x_{i+1} - 1)^2 + x_{i+1} - 1), (-x_i^2 - (x_{i+1} - 1)^2 + x_{i+1} - 1) \right) \tag{A.10}$$

starting from $\boldsymbol{y}$ where $y_i = 2$ when $\mod(i,2) = 0$ and $y_i = -1.5$ when $\mod(i,2) = 1$. For this function, it is known that $f(\boldsymbol{x}^*) = 0$.

11. Referred to as "Problem 2 in TEST29" [65]

$$\underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad \max_{1 \le i \le n} |x_i| \tag{A.11}$$

starting from $\boldsymbol{y}$ where $y_i = i/n$ for $i = 1, 2, \ldots, n/2$ and $y_i = -i/n$ for $i = n/2+1, \ldots, n$.

12. Referred to as "Problem 5 in TEST29" [65]

$$\underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad \sum_{i=1}^{n} \left| \sum_{j=1}^{n} \frac{x_j}{i+j-1} \right| \tag{A.12}$$

starting from $\boldsymbol{y}$ where $y_i = 1$ for $i = 1, 2, \ldots, n$.

13. Referred to as "Problem 6 in TEST29" [65]

$$\begin{aligned} \underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad & \max_{1 \le i \le n} |(3 - 2x_i)x_i + 1 - x_{i-1} - x_{i+1}| \\ \text{subject to} \quad & x_0 = x_{n+1} = 0, \end{aligned} \tag{A.13}$$

starting from $\boldsymbol{y}$ where $y_i = -1$ for $i = 1, 2, \ldots, n$.

14. Referred to as "Problem 19 in TEST29" [65]

$$\begin{aligned} \underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad & \max_{1 \le i \le n} \left((3 - 2x_i)x_i - x_{i-1} - 2x_{i+1} + 1\right)^2 \\ \text{subject to} \quad & x_0 = x_{n+1} = 0, \end{aligned} \tag{A.14}$$

starting from $\boldsymbol{y}$ where $y_i = -1$ for $i = 1, 2, \ldots, n$.

15. Referred to as "Problem 20 in TEST29" [65]

$$\begin{aligned} \underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad & \max_{1 \le i \le n} |(0.5x_i - 3)x_i - 1 + x_{i-1} - 2x_{i+1}| \\ \text{subject to} \quad & x_0 = x_{n+1} = 0, \end{aligned} \tag{A.15}$$

starting from $\boldsymbol{y}$ where $y_i = -1$ for $i = 1, 2, \ldots, n$.

16. Referred to as "Problem 22 in TEST29" [65]

$$\begin{aligned} \underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad & \max_{1 \le i \le n} \left| 2x_i + \frac{1}{2(n+1)^2} \left(x_i + \frac{i}{n+1} + 1 \right)^3 - x_{i-1} - x_{i+1} \right| \\ \text{subject to} \quad & x_0 = x_{n+1} = 0, \end{aligned} \tag{A.16}$$

starting from $\boldsymbol{y}$ where $y_i = \frac{i}{n+1}\left(\frac{i}{n+1} - 1\right)$ for $i = 1, 2, \ldots, n$.

17. Nesterov 1 [88]

$$\underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad \frac{1}{4}(x_1 - 1)^2 + \sum_{i=1}^{n-1}(x_{i+1} - 2x_i^2 + 1)^2 \tag{A.17}$$

starting from $\boldsymbol{y}$ where $y_i = 3$ for $i = 1, 2, \ldots, n$. For this function, it is known that $f(\boldsymbol{x}^*) = 0$.

18. Nesterov 2 [88]

$$\underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad \frac{1}{4}(x_1 - 1)^2 + \sum_{i=1}^{n-1}\left|x_{i+1} - 2x_i^2 + 1\right| \tag{A.18}$$

starting from $\boldsymbol{y}$ where $y_i = 3$ for $i = 1, 2, \ldots, n$. For this function, it is known that $f(\boldsymbol{x}^*) = 0$.

19. Nesterov 3 [88]

$$\underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \quad \frac{1}{4}|x_1 - 1| + \sum_{i=1}^{n-1}|x_{i+1} - 2|x_i| + 1| \tag{A.19}$$

starting from $\boldsymbol{y}$ where $y_i = 3$ for $i = 1, 2, \ldots, n$. For this function, it is known that $f(\boldsymbol{x}^*) = 0$.

A.3 Further results from experiment

In Section 4.1.3, we ran a numerical experiment comparing DGM operating with BLS to HLS. At the time, we stated that we ran each algorithm both for the designated starting point of each test function, but also from 20 randomly generated starting points. Moreover, in Section 4.1.3 we only showed the results corresponding to test functions for which $f(\boldsymbol{x}^*)$ was known.

In this section, we present the remainder of this experiment. In Tables A.3.1, A.3.4, A.3.7 and A.3.10 we show the outcomes corresponding to the test functions for which the minimum is not known, while starting from the default starting position. When comparing f_{min} here, all we can do is see which algorithm performed better than the other. Thus we write 0 in this column for the algorithm which won, and the performance difference for the algorithm which was slower.

f	Alg	f_{min}	$\#f$	$\#\varphi$	#g	Alg	f_{min}	$\#f$	$\#\varphi$	#g
A.8	BLS	2.05e-07	3075	21.23	1.33	HLS	0	1672	14.47	1.65
A.11	BLS	3.47e-06	787	17.82	1.12	HLS	0	37	4.33	2
A.12	BLS	1.26e-06	4909	23.9	1.22	HLS	0	194	7.92	2.08
A.13	BLS	4.28e-07	858	20.18	1.27	HLS	0	43	7.5	4
A.14	BLS	4.06e-11	710	10.26	2.13	HLS	0	266	20.83	7.17
A.15	BLS	4.2e-06	866	21.5	1.19	HLS	0	119	14.6	2.4
A.16	BLS	4.14e-06	818	20.71	1.23	HLS	0	116	15.2	2

Table A.3.1 – The outcome of running DGM with both HLS and BLS on the set of functions with unknown local minima found in Appendix A.2 taking $n = 2$

f	Alg	f_{min}	$\#f$	$\#\varphi$	#g	Alg	f_{min}	$\#f$	$\#\varphi$	#g
A.1	BLS	1.37e-11	254.8	1.65	1.1	HLS	9.28e-12	114.05	10.47	1.92
A.2	BLS	9.36e-07	5397.95	23.19	1.27	HLS	3.09e-08	266.65	7.02	1.93
A.3	BLS	5.38e-10	850.25	15.68	1.98	HLS	3.24e-10	72.5	4.67	5.83
A.4	BLS	3.82e-06	2132.9	25.35	1.18	HLS	6.11e-07	116.8	14.79	2.19
A.5	BLS	4.63e-06	2150.6	25.51	1.19	HLS	6.37e-07	116.9	14.57	2.19
A.6	BLS	3.29e-06	819.05	17.96	1.25	HLS	1.26e-06	67.05	8.01	2.25
A.7	BLS	4.9e-06	771.1	17.54	1.24	HLS	1.43e-14	83.2	12.95	2.04
A.9	BLS	3e-07	1070.45	16.51	1.4	HLS	1.08e-08	646.1	13.27	2.23
A.10	BLS	2.32e-07	1093.85	16.75	1.43	HLS	3.03e-08	626.95	13.36	2.2
A.17	BLS	0.00478	32502.5	25.45	1.69	HLS	0.00312	7181.5	9.96	1.9
A.18	BLS	1.23e-06	15115.5	26.04	1.32	HLS	3.17e-07	678.5	9.39	1.86

Table A.3.2 – The average outcome of running DGM with both HLS and BLS on the set of functions with known local minima found in Appendix A.2 taking $n = 2$

Furthermore, in Tables A.3.2, A.3.3, A.3.5, A.3.6, A.3.8, A.3.9, A.3.11 and A.3.12, we display the results for the randomly generated starting points. In each cell of the every table, the number displayed is the average value over all starting points.

f	Alg	f_{min}	$\#f$	$\#\varphi$	#g	Alg	f_{min}	$\#f$	$\#\varphi$	#g
A.8	BLS	0	3612.9	21.5	1.3	HLS	4.3e-07	1716.1	14.57	1.68
A.11	BLS	4.27e-06	712.1	17.23	1.15	HLS	0	37.6	4.33	2.07
A.12	BLS	2.42e-06	5591.35	23.26	1.21	HLS	0	225.4	8.61	1.99
A.13	BLS	3.58e-06	880.9	19.9	1.3	HLS	0	47.3	7.9	3.99
A.14	BLS	9.27e-11	735	10.12	2.02	HLS	0	222.65	19.38	6.09
A.15	BLS	2.64e-06	900.85	21.63	1.24	HLS	0	119.85	14.58	2.4
A.16	BLS	5.31e-06	873.2	21.13	1.26	HLS	0	117.6	15.43	2.03

Table A.3.3 – The average outcome of running DGM with both HLS and BLS on the set of functions with unknown local minima found in Appendix A.2 taking $n = 2$

f	Alg	f_{min}	$\#f$	$\#\varphi$	#g	Alg	f_{min}	$\#f$	$\#\varphi$	#g
A.8	BLS	0	17568	21.64	2.69	HLS	7.51e-09	3174	19.04	3.36
A.11	BLS	3.14e-06	2227	16.16	1.15	HLS	0	664	19.33	1.71
A.12	BLS	0	35933	24.33	1.94	HLS	0.0045	570	14.29	4.07
A.13	BLS	3.34e-06	2225	19.65	1.21	HLS	0	5128	22.77	2.45
A.14	BLS	2.61e-12	2478	11.86	1.82	HLS	0	3604	20.8	3.1
A.15	BLS	4.29e-06	22720	31	1.31	HLS	0	1413	21.91	3.15
A.16	BLS	4.05e-07	42286	31.22	1.75	HLS	0	2015	21.74	3.19

Table A.3.4 – The outcome of running DGM with both HLS and BLS on the set of functions with unknown local minima found in Appendix A.2 taking $n = 5$. Whenever one algorithm has notably outperformed the other, whether due to being cheaper or finding a better solution, we have highlighted that algorithm.

f	Alg	f_{min}	$\#f$	$\#\varphi$	#g	Alg	f_{min}	$\#f$	$\#\varphi$	#g
A.1	BLS	5.53e-11	857.05	1.89	1.07	HLS	2.07e-11	586.8	15.11	1.32
A.2	BLS	1.23e-05	17751.8	21.94	2	HLS	1.01e-05	2760.85	9.6	2.74
A.3	BLS	0.0019	5937	24.17	2.54	HLS	0.00122	6605.95	24.65	1.88
A.4	BLS	1.69e-05	1957.05	21.66	1.45	HLS	1.38e-05	9445.85	28.67	1.32
A.5	BLS	3.61e-07	23511.8	27.88	2.44	HLS	2e-08	1764.05	16.02	3.86
A.6	BLS	5.67e-06	949.25	18.45	1.49	HLS	2.96e-06	142.75	7.62	2.3
A.7	BLS	8.71e-06	1401.6	20.52	1.72	HLS	0.106	14311.3	28.07	1.19
A.9	BLS	7.65e-10	5968.7	19.05	1.78	HLS	7.28e-10	1269.1	14.21	2.95
A.10	BLS	4.25e-06	2650.35	19.58	1.66	HLS	0.000109	17839.8	24.74	1.29
A.17	BLS	0.481	18136.4	28.02	1.52	HLS	9.17e-05	3659.4	25.21	1.6
A.18	BLS	0.0452	2252.95	22.15	2.15	HLS	0.0453	2771.1	24.86	1.66

Table A.3.5 – The average outcome of running DGM with both HLS and BLS on the set of functions with known local minima found in Appendix A.2 taking $n = 5$. Whenever one algorithm has notably outperformed the other, whether due to being cheaper or finding a better solution, we have highlighted that algorithm.

f	Alg	f_{min}	$\#f$	$\#\varphi$	#g	Alg	f_{min}	$\#f$	$\#\varphi$	#g
A.8	BLS	3.86e-08	18915.8	22.23	2.64	HLS	0	3554.8	18	3.54
A.11	BLS	2.95e-06	2291	16.64	1.15	HLS	0	802.95	19.88	1.43
A.12	BLS	0	28482.9	23.65	1.89	HLS	0.00281	542.55	12.88	4.05
A.13	BLS	0	2417.65	20.42	1.26	HLS	0.0276	3429	22.13	2.37
A.14	BLS	1.67e-11	2681.25	11.66	2.07	HLS	0	2627.55	19.84	3.07
A.15	BLS	8.69e-05	25622.3	31.59	1.48	HLS	0	1405.5	22.62	2.8
A.16	BLS	0.000155	36134.9	31.24	1.69	HLS	0	1880	21.18	3.2

Table A.3.6 – The average outcome of running DGM with both HLS and BLS on the set of functions with unknown local minima found in Appendix A.2 taking $n = 5$. Whenever one algorithm has notably outperformed the other, whether due to being cheaper or finding a better solution, we have highlighted that algorithm.

f	Alg	f_{min}	$\#f$	$\#\varphi$	#g	Alg	f_{min}	$\#f$	$\#\varphi$	#g
A.8	BLS	0.131	51943	32.69	1.57	HLS	0	79913	21.43	6.89
A.11	BLS	2.35e-06	5589	16.34	1.16	HLS	0	3047	24.36	1.25
A.12	BLS	0.000267	23942	23.38	2.55	HLS	0	7050	19.15	3.6
A.13	BLS	3.74e-06	6722	20.99	1.55	HLS	0	47639	23.33	2.63
A.14	BLS	2.37e-10	4575	12.6	1.78	HLS	0	31621	23.48	3.02
A.15	BLS	0.28	48956	32.52	1.31	HLS	0	29434	24.19	3.29
A.16	BLS	0.00625	74849	30.25	3.88	HLS	0	63389	21.62	5.86

Table A.3.7 – The outcome of running DGM with both HLS and BLS on the set of functions with unknown local minima found in Appendix A.2 taking $n = 10$. Whenever one algorithm has notably outperformed the other, whether due to being cheaper or finding a better solution, we have highlighted that algorithm.

f	Alg	f_{min}	$\#f$	$\#\varphi$	#g	Alg	f_{min}	$\#f$	$\#\varphi$	#g
A.1	BLS	2.64e-11	2350.25	1.91	1.08	HLS	1.99e-11	2609.9	18.39	1.12
A.2	BLS	0.00636	20504.5	22.49	1.78	HLS	0.00817	1552.75	8.79	3.3
A.3	BLS	0.00696	58263.5	32.48	2.17	HLS	0.00168	48490.4	22.65	3.12
A.4	BLS	0.0424	49953.8	35.76	2	HLS	0.129	39986.1	29.46	1.58
A.5	BLS	0.0843	64956.4	35.4	2.7	HLS	0.0937	7496.6	20.8	5.24
A.6	BLS	7.22e-06	1970.8	18.42	2.42	HLS	3.97e-06	667.9	6.35	4.45
A.7	BLS	1.31e-05	2646.7	21.47	2.01	HLS	1.14e-05	19847.7	29.36	1.49
A.9	BLS	5.71e-09	14931	21.87	1.86	HLS	6.32e-10	2500.9	15.17	3.91
A.10	BLS	8.23e-06	5053.55	20.7	1.84	HLS	0.0012	38358.2	26.37	2.74
A.17	BLS	0.109	84591.8	29.93	7.46	HLS	0.32	42164	25.89	2.57
A.18	BLS	0.00147	4547.3	22.91	3.3	HLS	0.00149	25968.2	26.16	1.51

Table A.3.8 – The average outcome of running DGM with both HLS and BLS on the set of functions with known local minima found in Appendix A.2 taking $n = 10$. Whenever one algorithm has notably outperformed the other, whether due to being cheaper or finding a better solution, we have highlighted that algorithm.

f	Alg	f_{min}	$\#f$	$\#\varphi$	#g	Alg	f_{min}	$\#f$	$\#\varphi$	#g
A.8	BLS	0.0203	54558.4	31.03	2.22	HLS	0	85611.4	21.05	7.79
A.11	BLS	0.00115	4593.85	15.64	1.17	HLS	0	4127.55	24.78	1.32
A.12	BLS	0.000675	20026	23.67	1.81	HLS	0	6516.75	19.49	3.21
A.13	BLS	0	5915.05	20.71	1.38	HLS	0.0283	37442.1	24.38	2.54
A.14	BLS	2.13e-11	5484.4	12.21	2.28	HLS	0	30370.1	22.53	2.95
A.15	BLS	0.36	52550.3	33.34	1.57	HLS	0	31602.6	24.22	3.29
A.16	BLS	0.00456	71240.9	30.76	3.51	HLS	0	35136.5	21.87	5.49

Table A.3.9 – The average outcome of running DGM with both HLS and BLS on the set of functions with unknown local minima found in Appendix A.2 taking $n = 10$. Whenever one algorithm has notably outperformed the other, whether due to being cheaper or finding a better solution, we have highlighted that algorithm.

f	Alg	f_{min}	$\#f$	$\#\varphi$	#g	Alg	f_{min}	$\#f$	$\#\varphi$	#g
A.8	BLS	0	191546	30.94	6.32	HLS	0.505	84954	23.03	2.31
A.11	BLS	5.95e-07	24805	15.93	1.8	HLS	0	43408	26.87	1.79
A.12	BLS	0.0344	51366	22.86	1.02	HLS	0	48004	21.03	6.22
A.13	BLS	0	35623	19.64	2.37	HLS	1.99	102704	23.1	2.98
A.14	BLS	0	60789	28.71	1.16	HLS	0.797	129703	23.6	4
A.15	BLS	0.0731	185949	32.71	5.82	HLS	0	211499	22.59	7.19
A.16	BLS	0	124623	24.47	10.48	HLS	0.000136	21612	20.44	8.08

Table A.3.10 – The outcome of running DGM with both HLS and BLS on the set of functions with unknown local minima found in Appendix A.2 taking $n = 25$. Whenever one algorithm has notably outperformed the other, whether due to being cheaper or finding a better solution, we have highlighted that algorithm.

f	Alg	f_{min}	$\#f$	$\#\varphi$	#g	Alg	f_{min}	$\#f$	$\#\varphi$	#g
A.1	BLS	3.16e-11	12396.5	2.01	1.06	HLS	3e-11	11830.4	19.28	1.1
A.2	BLS	0.0208	29773.8	20.06	1.75	HLS	0.0342	4487.6	8.65	2.97
A.3	BLS	0.036	92593.2	33.56	2.2	HLS	0.071	73773.1	27.77	1.69
A.4	BLS	3.88	76127.3	37.18	1.43	HLS	2.32	76689.9	27.74	1.81
A.5	BLS	0.0706	134184	35.85	3.97	HLS	0.124	40028.7	23.93	10.1
A.6	BLS	1.29e-05	11269.4	17.29	6.45	HLS	9.14e-06	5077.6	5.51	9.85
A.7	BLS	2.11e-05	5560.25	24.46	3.24	HLS	0.0644	57392.6	27.61	1.9
A.9	BLS	4.98e-09	31819	23.31	2.84	HLS	7.47e-10	5736.3	16.25	5.27
A.10	BLS	0.0192	91448	34.84	2.15	HLS	0.0754	73852.6	26.38	3.06
A.17	BLS	0.232	79390.2	27.2	13.66	HLS	2.85	67019.3	29.57	1.36
A.18	BLS	0.349	55700.8	30.02	4.39	HLS	3.02	57472.6	25.66	1.15

Table A.3.11 – The average outcome of running DGM with both HLS and BLS on the set of functions with known local minima found in Appendix A.2 taking $n = 25$. Whenever one algorithm has notably outperformed the other, whether due to being cheaper or finding a better solution, we have highlighted that algorithm.

f	Alg	f_{min}	$\#f$	$\#\varphi$	#g	Alg	f_{min}	$\#f$	$\#\varphi$	#g
A.8	BLS	0	147395	31.26	4.57	HLS	0.444	79987.4	23.13	2.11
A.11	BLS	0	19829.8	15.85	1.26	HLS	1.54e-05	40809.5	27.05	1.3
A.12	BLS	0.0164	51796.1	22.57	1.09	HLS	0	22097.4	21.46	4.17
A.13	BLS	0	29088.5	19.98	1.59	HLS	1.92	97185	24.33	2.73
A.14	BLS	0	58198.9	27.91	1.09	HLS	0.83	126431	24.24	3.85
A.15	BLS	0.0743	187497	32.69	5.88	HLS	0	188550	22.43	6.31
A.16	BLS	0	156360	24.08	10.77	HLS	4.52e-06	51009.6	19.26	10.4

Table A.3.12 – The average outcome of running DGM with both HLS and BLS on the set of functions with unknown local minima found in Appendix A.2 taking $n = 25$. Whenever one algorithm has notably outperformed the other, whether due to being cheaper or finding a better solution, we have highlighted that algorithm.

Appendix B

Additional content for univariate optimisation

B.1 More on Brent's Method

B.1.1 A practical implementation of Brent's Method

When implementing Brent's method in practice, the main question faced is how to determine when to use a Golden Section step and when to use an interpolation step. In Algorithm 13, we present an implementation of Brent's method in pseudo-code as taken from Numerical Recipes [85].

There are a few points to note within this implementation. First observe that two tolerances ϵ_1 and ϵ_2 are required. The first of these, ϵ_1 can be thought of as the minimum step size which is considered meaningful. That is, if $|x - y| < \epsilon_1$, then we consider x and y to be numerically equivalent. Separately we have a second tolerance ϵ_2. This value refers to our targeted accuracy; that is the length of the final bracket must have a length of at most ϵ_2.

We determine whether to carry out an interpolation step in Line 8. Having already computed the interpolating quadratic in Line 5, we accept the minimum of the quadratic x_i as the next test point if it satisfies the following conditions:

Algorithm 13: Brent's Method [85]

```
/* Initialisation:                                                    */
```
input : $br([\ell_0, r_0], m_0), 0 < \epsilon_1 < \frac{1}{10}\epsilon_2$.

$i = 0$;
$d_{old} = |r_i - \ell_i|$;
$f_\ell, f_m, f_r = f(\ell_i), f(m_i), f(r_i)$;
while $|r_i - \ell_i| > 2\epsilon_2$ **do**
    Interpolate the points ℓ_i, r_i, m_i to obtain the quadratic function $q(x)$.
    $x_i = \arg\min_x q(x)$.
    $d_{new} = x_i - m_i$;
    if $(x_i \notin (\ell_i, r_i)$ *or* $|d_{old}| \leq \epsilon_1$ *or* $|d_{new}| \geq \frac{1}{2}|d_{old}|$ **then**
        if $x_i > m_i$ **then**
            $d_{new} = (r_i - m_i)\frac{3-\sqrt{5}}{2}$.
        else
            $d_{new} = (\ell_i - m_i)\frac{3-\sqrt{5}}{2}$.
        $x_i = m_i + d_{new}$;
    else if $|d_{new}| < \epsilon_1$ **then**
        $x_i = m_i + \epsilon_1 sign(d_{new})$;
    $f_x = f(x_i)$.
    if $f_x \leq f_m$ **then**
        if $x_i < m_i$ **then**
            $\ell_{i+1}, m_{i+1}, r_{i+1} = \ell_i, x_i, m_i$
            $f_\ell, f_m, f_r = f_\ell, f_x, f_m$
        else
            $\ell_{i+1}, m_{i+1}, r_{i+1} = m_i, x_i, r_i$
            $f_\ell, f_m, f_r = f_m, f_x, f_r$
    else
        if $x_i < m_i$ **then**
            $\ell_{i+1}, m_{i+1}, r_{i+1} = \ell_i, m_i, x_i$
            $f_\ell, f_m, f_r = f_\ell, f_m, f_x$
        else
            $\ell_{i+1}, m_{i+1}, r_{i+1} = x_i, m_i, r_i$
            $f_\ell, f_m, f_r = f_x, f_m, f_r$
    $i = i + 1$;
    $d_{old} = d_{new}$;

output: ℓ_i, m_i, r_i

1. $x_i \in (\ell_i, r_i)$. If ℓ_i, m_i, r_i is indeed a bracket, then this should always be the case. Regardless, this condition ensures that we never take a step outside of the current

bracket.

2. $|d_{new}| < \frac{1}{2}|d_{old}|$. When the interpolation method is working properly, convergence should be rapid. Therefore we require that the step from the current minimum decrease by at least a factor of two with each iteration. Failure to do this in practice implies that the interpolation steps are not converging. When this happens, we resort to a Golden Section step for ensured robustness.

3. $|d_{old}| > \epsilon_1$. When this condition is not satisfied, it means that the previous test point was within a distance ϵ_1 of a previously known point, and thus must have been generated with an interpolation step. When this condition applies, it could either mean that the interpolation step has converged successfully, or else that it is going very slowly. At this point we force a Golden Section step in order to ensure robustness. If the interpolation step process had indeed converged, than this will only add one additional function evaluation before the algorithm terminates.

When running the algorithm, it may happen that the interpolation step suggests a new point which is close to a previously known point. While this may imply convergence, it is also dangerous given that the closer two points are together, the more sensitive the interpolation process is to computational error. Hence in Line 14 we check whether the step length is less than ϵ_1, our minimum allowable step length. If so, we increase the length of the step to ϵ_1 while maintaining the original direction.

Finally, depending on the relative positions of x_i and m_i along with the relative values of f_x and f_m, there exists a unique trio of points which form a smaller bracket than previously. Lines 17 to 30 lay out which points we pick for our refined bracket given x_i, m_i, f_x, f_m.

B.1.2 Super-linear convergence of Brent's method

As seen in previous sections, Brent's method does not necessarily behave in the same manner for every iteration with both an interpolation step and a Golden Section step

being possible. Therefore, when determining the convergence rate of Brent's method we view these possibilities separately.

Clearly when a Golden Section step is taken, convergence is the same as in Section 5.2.1. Thus all that we need is the convergence rate when using interpolation steps. For the purpose of rigour, we formally define a sequence of points generated in this way:

Definition B.1. *Let $f \in C^2[a,b]$ and $\{x_i\} \in [a,b]$ be a sequence of points. We call $\{x_i\}$ a sequence of* ***successive interpolated minima*** *if it holds that:*

$$P'(x_{i+3}) = 0,$$

where P is the unique 2nd order polynomial which interpolates the function f at the points x_i, x_{i+1} and x_{i+2}.

We now present the following definition and result taken from [80].

Definition B.2 ([80])**.** *Let $f : [a,b] \ni \mathbb{R} \to \mathbb{R}$ be a function. The* ***modulus of continuity*** *$w(f;\delta)$ is defined by:*

$$w(f;d) := \sup_{\substack{x,y\in[a,b] \\ |x-y|\leq\delta}} |f(x) - f(y)|,$$

given any $\delta > 0$.

Remark. *We note that if f is Lipschitz Continuous with Lipschitz constant M, then it holds that $w(f,\delta) \leq M\delta$.*

Theorem 30 ([80])**.** *Suppose that $f \in C^2[a,b]$, $\{x_i\} \in [a,b]$ is a sequence of successive interpolated minima , $\zeta \in [a,b]$ such that $f'(\zeta) = 0$, $[\zeta - \delta_0, \zeta + \delta_0] \subset [a,b]$ where $\delta_0 = \max_i |x_i - \zeta|$ and $3w(f^{(2)};\delta_0) < |f^{(2)}(\zeta)|$. Then the sequence $\{x_i\}$ converges superlinearly to ζ. In particular, if we define:*

$$\delta_n = \max_{i=0,1,2} |x_{n+i} - \zeta|$$

and

$$\lambda_n = \frac{3w(f^{(2)}; \delta_n)}{|f^{(2)}(\zeta)|},$$

then the sequence δ_n is monotonically decreasing and $\delta_{n+3} \leq \lambda_n \delta_{n+1}$.

Remark. *What Theorem 30 proves is that the sequence of points generated by the interpolation step converges superlinearly under certain conditions. This is not enough to guarantee superlinear convergence for Brent's method itself as it does not guarantee that the algorithm consistently uses the interpolation step instead of Golden section step. While we do not prove that this happens, it is what we observe in practice.*

B.2 Calculations for Theorem 21

For the calculations in this section, we use the following change of variables: $\boldsymbol{t} = \begin{pmatrix} p & q & r & s \end{pmatrix}^T = \begin{pmatrix} x_1^L - x_2^L, & x^M - x_1^L, & x_1^R - x^M, & x_2^R - x_1^R \end{pmatrix}^T$. Converting our calculations to being in terms of $\boldsymbol{t}$ both reduces the number of variables, and simplifies the constraint $\mathcal{X} \in \tilde{\mathcal{B}}$ (recall Definition 5.13) into $\boldsymbol{t} \in \mathbb{R}^4_+$.

The algebra for the calculations which follow is tedious, and therefore we employed Mathematica to compute the values of U_I, $I \in \mathcal{A}$ as well as changing the variables used.

$$
\begin{aligned}
\frac{b(U_{44}\mathcal{X})}{b(\mathcal{X})} &= \frac{(q+r)(p+q+r)}{2(q+r)(p+q+r)+(q+r)^2+s^2+s(p+3q+3r)} < \frac{1}{2},\\
\frac{b(U_{111}\mathcal{X})}{b(\mathcal{X})} &= \frac{q}{p+2q+r} < \frac{1}{2},\\
\frac{b(U_{143}\mathcal{X})}{b(\mathcal{X})} &= \frac{q(p+q)(p+q+r)}{(p+2q+r)(3q^2+3qr+r^2+p(2q+r))},\\
&< \frac{q(p+q)}{3q^2+3qr+r^2+p(2q+r)} < \frac{1}{2},\\
\frac{b(U_{422}\mathcal{X})}{b(\mathcal{X})} &= \frac{(q+r)(p+q+r)}{2(q+r)(p+q+r)+(q+r)^2+s^2+s(p+3q+3r)} < \frac{1}{2},\\
\frac{b(U_{1411}\mathcal{X})}{b(\mathcal{X})} &= \frac{1}{2+\left(\dfrac{4q^3+p^2r+7q^2r+5qr^2+r^3+p(q+r)(3q+2r)}{q(p+q)(p+q+r)}\right)} \leq \frac{1}{2},\\
\frac{b(U_{1141}\mathcal{X})}{b(\mathcal{X})} &= \frac{q(p+q)(p+q+r)}{(p+2q+r)(p^2+3q^2+r(r+s)+2q(2r+s)+p(3q+2r+s))},\\
&< \frac{q(p+q)}{3q(p+q)+p^2+r(r+s)+2q(2r+s)+2rp+ps} < \frac{1}{3},\\
\frac{b(U_{1423}\mathcal{X})}{b(\mathcal{X})} &= \frac{(p+q+r)(p+2q+r)(q(p+q)-r(r+s))}{(p+2q+2r+s)} \times\\
&\quad \frac{1}{(4q(q+r)^2+r^3+p^2(2q+r)+q^2(s-r)+p(6q^2+2r^2+q(7r+s)))},\\
&< \frac{(p+q+r)(q(p+q)-r(r+s))}{4q(q+r)^2+r^3+p^2(2q+r)+q^2(s-r)+p(6q^2+2r^2+q(7r+s))},\\
&< \frac{(p+q+r)(q(p+q)-r^2)}{4q(q+r)^2+r^3+p^2(2q+r)-q^2r+p(6q^2+2r^2+7qr)},\\
&< \frac{q(p+q)-r^2}{2q(p+q)} < \frac{1}{2}.
\end{aligned}
$$

For the sequence 414, we need to employ another piece of information. From Equation (5.18), we see that U_1 is only applied to $\mathcal{X}$ if $\varphi(\mathcal{X}) < 0$ (see Equation (5.49)). This is equivalent to $q(p+q)-r(r+s) > 0$ when written in terms of our alternative variables.

Proceeding with the calculation we find:

$$\frac{b(U_{414}\mathcal{X})}{b(\mathcal{X})} = \frac{1}{2 + \left(\dfrac{q^3 + r^2(p+r) + q^2(7r+s) + qr(3p+7r+3s)}{q(q(p+q) - r(r+s))} \right)},$$

which is bounded above by $\frac{1}{2}$ precisely when $q(p+q) - r(r+s) > 0$. Therefore $b(U_{414}\mathcal{X})/b(\mathcal{X}) < \frac{1}{2}$ as required.

There remain 4 elements of $\mathcal{A}$ for which we must establish a bound: 434, 4322, 4314 and 4114. As the previous change of variables does little to simplify the calculations for these subsequences, we instead use the following alternative change of variables: $(a,b,c,d) = \left(\; x_1^R - x_1^L, \quad x_1^R - x_2^L, \quad x_2^R - x_1^L, \quad \varphi(\mathcal{X}) \; \right)$. For this set of variables, $\mathcal{X} \in \tilde{\mathcal{B}}$ is equivalent to $a, b, c > 0$, $b > a$ and $c > a$. Using this transformation, we find:

$$\begin{aligned}
\frac{b(U_{434}\mathcal{X})}{b(\mathcal{X})} &= \frac{bc^2(b+c)(b+c-a)}{(ab+bc+c^2)(ab^2+2b^2c+3bc^2+c^3)} < \frac{bc(b+c)}{ab^2+2b^2c+3bc^2+c^3} \\
&< \frac{bc(b+c)}{2b^2c+3bc^2+c^3} < \frac{b(b+c)}{2b^2+3bc+c^2} < \frac{b}{2b+c} < \frac{1}{2}, \\
\frac{b(U_{4322}\mathcal{X})}{b(\mathcal{X})} &= \frac{bc^2(b+c)(b+c-a)}{(ab+bc+c^2)(ab^2+2b^2c+3bc^2+c^3)} < \frac{bc(b+c)}{ab^2+2b^2c+3bc^2+c^3} \\
&< \frac{bc(b+c)}{2b^2c+3bc^2+c^3} < \frac{b(b+c)}{2b^2+3bc+c^2} < \frac{b}{2b+c} < \frac{1}{2}.
\end{aligned}$$

Remark. *From the calculations above, we see that* $b(U_{434}\mathcal{X})/b(\mathcal{X}) = b(U_{4322}\mathcal{X})/b(\mathcal{X})$ *and* $b(U_{44}\mathcal{X})/b(\mathcal{X}) = b(U_{422}\mathcal{X})/b(\mathcal{X})$. *While this may imply another relation which we have not taken advantage of, we have not examined this further.*

Finally we have the sequences 4314 and 4114. Similar to what we did with the sequence 414, we need to make use of additional information, specifically the fact that U_4 is only applied if $\varphi(\mathcal{X}) = d < 0$. In addition, U_1 may only follow after U_4 if $\varphi(U_4\mathcal{X}) < 0$.

This is equivalent to: $-d(ab+bc+c^2) > abc(b+c-a)$. From this it follows that:

$$\begin{aligned}
\frac{b(U_{4114}\mathcal{X})}{b(\mathcal{X})} &= \frac{abc^2(b+c)(b+c-a)}{(ab+bc+c^2)(ac(b+c)^2-d(ab+bc+c^2))}, \\
&< \frac{abc^2(b+c)(b+c-a)}{(ab+bc+c^2)(ac(b+c)^2+abc(b+c-a))}, \\
&= \frac{bc(b+c)(b+c-a)}{(ab+bc+c^2)((2b+c)(b+c)-ab)} < \frac{bc(b+c)^2}{(ab+bc+c^2)(2b+c)(b+c)}, \\
&= \frac{bc(b+c)}{(ab+bc+c^2)(2b+c)} < \frac{b}{2b+c} < \frac{1}{2}.
\end{aligned}$$

Similarly for the sequence 4314, U_3 may only follow after U_4 if $\varphi(U_4\mathcal{X}) > 0$. This is equivalent to: $-d(ab+bc+c^2) < abc(b+c-a)$. First we compute

$$\frac{b(U_{4314}\mathcal{X})}{b(\mathcal{X})} = \frac{-d(ac-d)(ab+bc+c^2)}{a(b+c)(ac(b+c)^2-d(ab+bc+c^2))}.$$

Next we observe that $\varphi(U_4\mathcal{X}) > 0$ is equivalent to:

$$-d > \frac{abc(b+c-a)}{ab+bc+c^2} \Leftrightarrow ac-d > \frac{ac(b+c)^2}{ab+bc+c^2}. \tag{B.1}$$

This implies that:

$$\frac{b(U_{4314}\mathcal{X})}{b(\mathcal{X})} < \frac{-dc(b+c)}{ac(b+c)^2-d(ab+bc+c^2)}.$$

Finally we that a sufficient condition for $b(U_{4314}\mathcal{X})/b(\mathcal{X}) < \frac{1}{2}$ is:

$$\begin{aligned}
&-2dc(b+c) < ac(b+c)^2 - d(ab+bc+c^2), \\
\Leftrightarrow\ &-d(-ab+bc+c^2) < ac(b+c)^2.
\end{aligned}$$

We see that this is true when we combine the requirement that $\varphi(U_4\mathcal{X}) > 0$ must hold for the sequence 4314 to occur with Equation (B.1). In particular:

$$-d(-ab+bc+c^2) < -d(ab+bc+c^2) < (ac-d)(ab+bc+c^2) < ac(b+c)^2.$$

Therefore, $b(U_{4314}\mathcal{X})/b(\mathcal{X}) < \frac{1}{2}$ holds and moreover $b(U_I\mathcal{X})/b(\mathcal{X}) < \frac{1}{2}\ \forall I \in \mathcal{I}_5$.

References

[1] Paul Broughton. Investment urged for clean coal technologies, 2017. URL https://www.engineerlive.com/content/investment-urged-clean-coal-technologies.

[2] M Deckers, A Wichtmann, and W Ulm. Modern Steam Turbines for Ultra Supercritical Steam Power Plants. In *World Engineering Convention (WEC)*, 2004.

[3] World Energy Council. World Energy Resources. *@WECouncil*, 2007:1–1028, 2016. ISSN 09619534. doi: http://www.worldenergy.org/wp-content/uploads/2013/09/Complete_WER_2013_Survey.pdf. URL https://www.worldenergy.org/wp-content/uploads/2016/10/World-Energy-Resources-Full-report-2016.10.03.pdf.

[4] The Numerical Algorithms Group (NAG). The NAG Fortran Library. URL www.nag.com.

[5] Frank E. Curtis and Michael L. Overton. A Sequential Quadratic Programming Algorithm for Nonconvex, Nonsmooth Constrained Optimization. *SIAM Journal on Optimization*, 22(2):474–500, 2012. ISSN 1052-6234. doi: 10.1137/090780201. URL http://epubs.siam.org/doi/10.1137/090780201.

[6] Sébastien Le Digabel and Stefan M. Wild. A Taxonomy of Constraints in Simulation-Based Optimization. pages 1–21, 2015. URL http://arxiv.org/abs/1505.07881.

[7] David G Luenberger, Yinyu Ye, and Others. *Linear and nonlinear programming*, volume 2. Springer, 1984.

[8] Richard H Byrd, Jorge Nocedal, and Richard A Waltz. *Knitro: An Integrated Package for Nonlinear Optimization*, pages 35–59. Springer US, Boston, MA, 2006. ISBN 978-0-387-30065-8. doi: 10.1007/0-387-30065-1_4. URL `https://doi.org/10.1007/0-387-30065-1{_}4`.

[9] Kenneth Holmström and Marcus M Edvall. *The TOMLAB Optimization Environment*, pages 369–376. Springer US, Boston, MA, 2004. ISBN 978-1-4613-0215-5. doi: 10.1007/978-1-4613-0215-5_19. URL `https://doi.org/10.1007/978-1-4613-0215-5{_}19`.

[10] Andrew R Conn, G I M Gould, and Philippe L Toint. *LANCELOT: a Fortran package for large-scale nonlinear optimization (Release A)*, volume 17. Springer Science & Business Media, 2013.

[11] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[12] Philip E Gill, Walter Murray, Michael A Saunders, and Margaret H Wright. User's guide for NPSOL (version 4.0): A Fortran package for nonlinear programming. Technical report, Stanford Univ CA Systems Optimization Lab, 1986.

[13] A Nekvinda and L Zajíček. A simple proof of the Rademacher theorem. *Časopis pro pěstování matematiky*, 113(4):337–341, 1988. URL `http://eudml.org/doc/21712`.

[14] P E Gill and W Murray. Safeguarded steplength algorithms for optimization using descent methods National Physical Laboratory Report NAC 37. 1974.

[15] Philip E Gill, Walter Murray, Michael A Saunders, and Margaret H Wright. User's Guide for NPSOL 5.0: A Fortran Package for Nonlinear Programming. *Technical report, Technical report SOL 866*, 2001.

[16] Philip E Gill, Walter Murray, Michael A Saunders, and Margaret H Wright. Some Theoretical Properties of an Augmented Lagrangian Merit Function. Technical report, STANFORD UNIV CA SYSTEMS OPTIMIZATION LAB, 1986.

[17] M J D Powell. On the convergence of a wide range of trust region methods for unconstrained optimization. *IMA journal of numerical analysis*, 30(1):289–301, 2010.

[18] C Lemarechal. Nonsmooth Optimization and Descent Methods. Iiasa research report, IIASA, Laxenburg, Austria, mar 1978. URL `http://pure.iiasa.ac.at/id/eprint/838/`.

[19] Boris T Poljak. Subgradient methods: a survey of Soviet research. In *Nonsmooth optimization: Proceedings of the IIASA Workshop March*, pages 5–30, 1977.

[20] E Ward Cheney and Allen A Goldstein. Newton's method for convex programming and Tchebycheff approximation. *Numerische Mathematik*, 1(1):253–268, 1959.

[21] James E Kelley Jr. The cutting-plane method for solving convex programs. *Journal of the Society for Industrial and Applied Mathematics*, 8(4):703–712, 1960.

[22] Frank H Clarke. Generalized gradients and applications. *Transactions of the American Mathematical Society*, 205(1970):247, 1975. ISSN 0002-9947. doi: 10.1090/S0002-9947-1975-0367131-6.

[23] Frank H Clarke. A new approach to Lagrange multipliers. *Mathematics of Operations Research*, 1(2):165–174, 1976.

[24] Philip Wolfe. Note on a method of conjugate subgradients for minimizing nondifferentiable functions. *Mathematical Programming*, 7(1):380–383, 1974. ISSN 00255610. doi: 10.1007/BF01585533.

[25] Helga Schramm and Jochem Zowe. A version of the bundle idea for minimizing a nonsmooth function: Conceptual idea, convergence analysis, numerical results. *SIAM journal on optimization*, 2(1):121–152, 1992.

[26] M Gaudioso and M F Monaco. Variants to the cutting plane approach for convex nondifferentiable optimization. *Optimization*, 25(1):65–75, 1992.

[27] L Lukšan and J Vlček. Globally convergent variable metric method for convex nonsmooth unconstrained minimization1. *Journal of Optimization Theory and Applications*, 102(3):593–613, 1999.

[28] Claudia Sagastizábal and Mikhail Solodov. An infeasible bundle method for nonsmooth convex constrained optimization without a penalty function or a filter. *SIAM Journal on Optimization*, 16(1):146–169, 2005.

[29] Warren L Hare, Claudia Sagastizábal, and M Solodov. A proximal bundle method for nonsmooth nonconvex functions with inexact information. *Computational Optimization and Applications*, 63(1):1–28, 2016.

[30] Robert Mifflin. A quasi-second-order proximal bundle algorithm. *Mathematical Programming*, 73(1):51–72, 1996.

[31] Warren L Hare and Claudia Sagastizábal. A redistributed proximal bundle method for nonconvex optimization. *SIAM Journal on Optimization*, 20(5):2442–2473, 2010.

[32] Liqun Qi and Xiaojun Chen. A preconditioning proximal Newton method for nondifferentiable convex optimization. *Mathematical Programming*, 76(3):411–429, 1997.

[33] Xiaojun Chen and Masao Fukushima. Proximal quasi-Newton methods for nondifferentiable convex optimization. *Mathematical Programming*, 85(2):313–334, 1999.

[34] Adil M. Bagirov, Bülent Karasözen, and Meral Sezer. Discrete gradient method: derivative-free method for nonsmooth optimization. *Journal of Optimization Theory and Applications*, 137(2):317–334, 2008.

[35] James V Burke, Adrian S Lewis, and Michael L Overton. Approximating subdifferentials by random sampling of gradients. *Mathematics of Operations Re-*

search, 27(3):567–584, 2002. ISSN 0364-765X. doi: 10.1287/moor.27.3.567.317. URL http://dx.doi.org/10.1287/moor.27.3.567.317{%}5Cnpapers2://publication/uuid/727743E2-EFAD-4D84-9E91-408C9CC2FFA5.

[36] James V. Burke, Adrian S. Lewis, and Michael L. Overton. A Robust Gradient Sampling Algorithm for Nonsmooth, Nonconvex Optimization. *SIAM Journal on Optimization*, 15(3):751–779, 2005. ISSN 1052-6234. doi: 10.1137/030601296. URL http://epubs.siam.org/doi/10.1137/030601296.

[37] Krzysztof C. Kiwiel. Convergence of the gradient sampling algorithm for nonsmooth nonconvex optimization. *SIAM Journal on Optimization*, 18(2):379–388, 2007.

[38] Frank E Curtis and Xiaocun Que. An adaptive gradient sampling algorithm for non- smooth optimization. 6788, 2013. doi: 10.1080/10556788.2012.714781.

[39] Frank E. Curtis and Xiaocun Que. A quasi-Newton algorithm for nonconvex , nonsmooth optimization with global convergence guarantees. *Mathematical Programming Computation*, 7(4):399–428, 2015. ISSN 1867-2957. doi: 10.1007/s12532-015-0086-2.

[40] Adrian S Lewis and Michael L Overton. Nonsmooth optimization via quasi-Newton methods. *Mathematical Programming*, 141(1):135–163, 2013.

[41] Frank E. Curtis, Tim Mitchell, and Michael L. Overton. A BFGS-SQP method for nonsmooth, nonconvex, constrained optimization and its evaluation using relative minimization profiles. *Optimization Methods and Software*, 32(1):148–181, 2017. ISSN 10294937. doi: 10.1080/10556788.2016.1208749. URL https://doi.org/10.1080/10556788.2016.1208749.

[42] James V Burke, Frank E Curtis, Adrian S Lewis, Michael L Overton, Lucas E A Simões, Adil M. Bagirov, M Gaudioso, Napsu Karmitsa, and Marko M. Mäkelä. Gradient Sampling Methods for Nonsmooth Optimization. pages 1–18, 2018. URL https://arxiv.org/pdf/1804.11003.pdf.

[43] Charles Audet and Warren Hare. *Derivative-Free and Blackbox Optimization*. Springer, 2017.

[44] Charles Audet and John E Dennis Jr. Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on optimization*, 17(1):188–217, 2006.

[45] Sébastien Le Digabel. Algorithm xxx: NOMAD: Nonlinear Optimization with the MADS algorithm. 2010.

[46] Michael J D Powell. The NEWUOA software for unconstrained optimization without derivatives. In *Large-scale nonlinear optimization*, pages 255–297. Springer, 2006.

[47] Michael J D Powell. Developments of NEWUOA for minimization without derivatives. *IMA journal of numerical analysis*, 28(4):649–664, 2008.

[48] Michael J D Powell. The BOBYQA algorithm for bound constrained optimization without derivatives. *Cambridge NA Report NA2009/06, University of Cambridge, Cambridge*, pages 26–46, 2009.

[49] Serge Gratton, Philippe L Toint, and Anke Tröltzsch. An active-set trust-region method for derivative-free nonlinear bound-constrained optimization. *Optimization Methods and Software*, 26(4-5):873–894, 2011.

[50] Coralia Cartis, Jan Fiala, Benjamin Marteau, and Lindon Roberts. Improving the flexibility and robustness of model-based derivative-free optimization solvers. *ACM Transactions on Mathematical Software (TOMS)*, 45(3):1–41, 2019.

[51] Giampaolo Liuzzi, Stefano Lucidi, Francesco Rinaldi, and Luis Nunes Vicente. Trust-region methods for the derivative-free optimization of nonsmooth black-box functions. *SIAM Journal on Optimization*, 29(4):3012–3035, 2019.

[52] Krzysztof C Kiwiel. A nonderivative version of the gradient sampling algorithm for nonsmooth nonconvex optimization. *SIAM Journal on Optimization*, 20(4): 1983–1994, 2010.

[53] Erlend S Riis, Matthias J Ehrhardt, G R W Quispel, and Carola-Bibiane Schönlieb. A geometric integration approach to nonsmooth, nonconvex optimisation. *arXiv preprint arXiv:1807.07554*, 2018.

[54] Adrian S Lewis and Michael L Overton. Nonsmooth optimization via quasi-Newton methods. *Mathematical Programming*, 141(1):135–163, 2013.

[55] Claude Lemaréchal. An introduction to the theory of nonsmooth optimization. *Optimization*, 17(6):827–858, 1986. ISSN 10294945. doi: 10.1080/02331938608843204.

[56] Alexei Kuntsevich and Franz Kappel. SolvOpt. *Available from World Wide Web: http://www. kfunigraz. ac. at/imawww/kuntsevich/solvopt/. WWW-Document.[106]*, 1997.

[57] Hermann Schichl and Hannes Fendl. A Second Order Bundle Algorithm for Nonsmooth, Nonconvex Optimization Problems. In *Numerical Nonsmooth Optimization*, pages 117–165. Springer, 2020.

[58] Yoav Benyamini and Joram Lindenstrauss. *Geometric nonlinear functional analysis*, volume 48. American Mathematical Soc., 1998.

[59] Jordan Bell. Fréchet derivatives and Gâteaux derivatives. pages 1–13, 2014.

[60] Frank H Clarke. *Optimization and nonsmooth analysis*, volume 5. Siam, 1990.

[61] Walter Rudin and Others. *Principles of mathematical analysis*, volume 3. McGraw-Hill New York, 1976.

[62] Marko M. Mäkelä. Survey of bundle methods for nonsmooth optimization. *Optimization Methods and Software*, 17(1):1–29, 2002. ISSN 10556788. doi: 10.1080/10556780290027828.

[63] A. A. Goldstein. Optimization of Lipschitz continuous functions. Technical Report 1, 1977.

[64] Jin Yu, S V N Vishwanathan, Simon Günter, and Nicol N Schraudolph. A quasi-Newton approach to nonsmooth convex optimization problems in machine learning. *Journal of Machine Learning Research*, 11(Mar):1145–1200, 2010.

[65] Marjo Haarala. *Large-scale nonsmooth optimization: variable metric bundle method with limited memory*. Number 40. University of Jyväskylä, 2004.

[66] Robert Mifflin and Claudia Sagastizábal. On VU-theory for Functions with Primal-Dual Gradient Structure. *SIAM Journal on Optimization*, 11(2):547–571, 2000.

[67] Robert Mifflin and Claudia Sagastizábal. A VU -algorithm for convex minimization. *Science*, 608(452966):583–608, 2005.

[68] Robert Mifflin and Claudia Sagastizábal. VU-decomposition derivatives for convex max-functions. In *Ill-Posed Variational Problems and Regularization Techniques*, pages 167–186. Springer, 1999.

[69] Yu G Evtushenko. Numerical methods for finding global extrema (case of a non-uniform mesh). *USSR Computational Mathematics and Mathematical Physics*, 11 (6):38–54, 1971.

[70] S A Piyavskii. An algorithm for finding the absolute extremum of a function. *USSR Computational Mathematics and Mathematical Physics*, 12(4):57–67, 1972.

[71] Bruno O Shubert. A sequential method seeking the global maximum of a function. *SIAM Journal on Numerical Analysis*, 9(3):379–388, 1972.

[72] Eldon R Hansen. Global optimization using interval analysis: the one-dimensional case. *Journal of Optimization Theory and Applications*, 29(3):331–344, 1979.

[73] Ramon E Moore. *Methods and applications of interval analysis*. SIAM, 1979.

[74] Arnold Neumaier. Complete search in continuous global optimization and constraint satisfaction. *Acta numerica*, 13(1):271–369, 2004.

[75] Marco Locatelli and Fabio Schoen. *Global optimization: theory, algorithms, and applications*. SIAM, 2013.

[76] Jack Kiefer. Sequential minimax search for a maximum. *Proceedings of the American mathematical society*, 4(3):502–506, 1953.

[77] Philip E Gill, Walter Murray, and Margaret H Wright. *Practical optimization*. Academic Press, 1981.

[78] P Jarratt. An iterative method for locating turning points. *The Computer Journal*, 10(1):82–84, 1967.

[79] R P Brent. A new algorithm for minimizing a function of several variables without calculating derivatives. *Algorithms for minimization without derivatives*, pages 200–248, 1976.

[80] R P Brent. Algorithms for minimization without derivatives. 1973.

[81] W W Hager. A derivative-based bracketing scheme for univariate minimization and the conjugate gradient method. *Computers & Mathematics with Applications*, 18(9):779–795, 1989.

[82] Walter Murray and Michael L Overton. Steplength algorithms for minimizing a class of nondifferentiable functions. *Computing*, 23(4):309–331, 1979.

[83] Robert Mifflin and J-J Strodiot. A rapidly convergent five-point algorithm for univariate minimization. *Mathematical programming*, 62(1-3):299–319, 1993.

[84] Stephen Abbott. *Understanding analysis*, volume 2. Springer, 2001.

[85] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[86] Robert Mifflin. On superlinear convergence in univariate nonsmooth minimization. *Mathematical Programming*, 49(1-3):273–279, 1990.

[87] Momin Jamil and Xin-She Yang. A literature survey of benchmark functions for global optimization problems. *arXiv preprint arXiv:1308.4008*, 2013.

[88] Mert Gürbüzbalaban and Michael L Overton. On Nesterov's nonsmooth Chebyshev–Rosenbrock functions. *Nonlinear Analysis: Theory, Methods & Applications*, 75(3):1282–1289, 2012.