A COMPARISON OF AUTOMATED METHODS FOR SOLVING VARIABLE DEMAND EQUILIBRIUM PROBLEMS

Andrew P. Woods

Supervisor: Prof. Mike Smith

Dissertation submitted for the
MSc in Data Analysis, Networks
& Non-linear Dynamics

Department of Mathematics
The University of York
York YO10 5DD

August 26, 2005

EPSRC Engineering and Physical Sciences Research Council

Contents

Introduction

In this dissertation we will compare several different automated methods for finding the solution to variable demand equilibrium problems. These will vary from the more classical search direction based algorithm to more modern nature-inspired algorithms such as simulated annealing. The work is motivated by research currently being undertaken, some of it at the University of York, on behalf of the Department for Transport. The DIADEM project, as it is known, desires rapid and reliable methods for solving these variable demand equilibrium problems so that the models can be put to use in modelling and hence designing Britain's road network.

In Chapter 1 first some brief details are given about the DfT's involvement in this project. Then it goes on to outline the theoretical concepts necessary for defining and understanding variable demand equilibrium. It also introduces a rough idea of how the algorithms we will examine are going to work.

In Chapter 2 we start by examining a basic algorithm for solving variable demand equilibrium problems that involves defining a search direction based upon the cost and demand functions. In this chapter we have linear cost and demand functions only, so line minimisation along the the search vector is achievable by an explicit formula. For each of our search directions it turns out that the minimum can be found as the solution to a quadratic equation.

In Chapter 3 we relax the condition of linearity upon our cost and demand functions to simply differentiability. Line minimisation along the search directions can now no-longer be performed explicitly with a formula. It requires a new concept, the Armijo-like step-length. This scheme gives the step lengths a sort of inertia, choosing the next step size by either doubling, halving or maintaining the previous step size. A promising new search direction created by pre-multiplying an existing search direction by a new component in order to fix a serious flaw is also examined.

In Chapter 4 we examine two of the most commonly used conventional multidimensional minimisation methods, the downhill simplex method and simulated annealing. Due to the nature of the memory requirements of the simplex method it is not applied directly to the problem, but works in terms of two perpendicular search directions. These techniques have the further advantage that they place no requirements upon the differentiability of the cost and demand functions.

The dissertation is concluded by discussing what we have learned throughout Chapters 1 to 4. Some ideas for further work to be done are also discussed.

CHAPTER 1

The central variable demand equilibrium model

1.1. Network flows

We typically call a directed graph a network. Now suppose we have a network with two nodes α and β called the origin and destination respectively. A flow on the network is an allocation of values to arcs such that the the sum of values into all nodes (except the origin and destination) is the same as the sum leaving them.

Now in this case, as we would expect, the total flow out of the origin, α, is identical to the total flow into the destination, β. With flow v out of α and flow v into β we say the *flow has value* v.

1.2. Context

Obviously these concepts are important in the modelling of transport networks, including walking, cycling, car and train networks. The Department for Transport (DfT) demands that major schemes for highways use variable demand modelling to simulate traffic flow. In order to aid the DfT create good schemes we aim to perform rigourous tests upon these models. This will allow them to build a better equilibrium transportation model and to find the distribution of traffic where most people are happy.

So different strategies can be compared to see which are likely to be the best in practise. Estimating the equilibrium flow and cost states in the model for each strategy will allow us to do this.

In addition to road traffic networks, equilibrium distributions are also extremely important in determining the economic and environmental impact of the real-life changes to the transportation system.

So what we want to know is how to:

- Make an applicable network model and
- Find its solutions.

1.3. The central variable demand equilibrium model

1.3.1. The main variables and functions.

DEFINITION 1.1. We suppose that in our model network we have K OD pairs and that each is connected by just $\mathbf{N}_{ij}$ routes. Thus the total number of routes is $N = \sum_{ij} \mathbf{N}_{ij}$. The main variables are as follows:

$$\begin{aligned} \mathbf{X}_{ijr} &= \text{“the flow along the } r^{th} \text{ route joining OD pair } ij\text{”};\\ \mathbf{X} &= \text{“the route flow vector comprising all the } \mathbf{X}_{ijr}\text{”};\\ \mathbf{Y}_{ij} &= \text{“the cost of travel between OD pair } ij\text{”};\\ \mathbf{Y} &= \text{“the cost vector comprising all the } \mathbf{Y}_{ij}\text{”}. \end{aligned}$$

Flows are measured in vehicles per minute (or similar). Costs are measured in time per unit of flow. Therefore a flow times a cost has no dimension.

NOTATION 1.2. *Let 0^N denote an N-vector of zeros and $+\infty^N$ denote an N-vector with each ordinate being $+\infty$. Then we define $[0^N, +\infty^N)$ to be $[0, +\infty)^N$ and $[0^K, +\infty^K)$ to be $[0, +\infty))^K$.*

DEFINITION 1.3. We now define two functions; the cost function $C(.)$ and the demand function $D(.)$.

Now let $\mathbf{C}_{ijr}(X)$ be the cost of traversing the r^{th} route joining OD pair ij when the flow vector is $X \geq 0$ and $\mathbf{D}_{ij}(Y)$ be the total flow between OD pair ij when the OD cost vector is Y where $Y \geq 0$.

Suppose that the cost function $\mathbf{C}(.)$ is defined throughout $[0^N, +\infty^N)$ and that the demand function $\mathbf{D}(.)$ is defined throughout $[0^K, +\infty^K)$. Thus including domains and co-domains, our two functions are:

$$\mathbf{C} : [0^N, +\infty^N) \to [0^N, +\infty^N) \text{ and}$$
$$\mathbf{D} : [0^K, +\infty^K) \to [0^K, +\infty^K).$$

DEFINITION 1.4. We define the two functions

$$\mathbf{T} : [0^N, +\infty^N) \to [0^K, +\infty^K) \text{ and}$$
$$\mathbf{S} : [0^K, +\infty^K) \to [0^N, +\infty^N)$$

as follows. For each ij and each ijr:

$$\mathbf{T}_{ij}(\mathbf{X}) = \sum_r \mathbf{X}_{ijr} \; \forall \, \mathbf{X} \in [0^N, +\infty^N) \text{ and}$$
$$\mathbf{S}_{ijr}(\mathbf{Y}) = Y_{ij} \; \forall \, \mathbf{Y} \in [0^K, +\infty^K).$$

$\mathbf{T}_{ij}(X)$ is the total flow from i to j and $\mathbf{S}_{ijr}(Y)$ spreads each cost $\mathbf{Y}_{ij}$ over all routes joining node i and j.

1.3.2. Assumptions about the variables and functions.

ASSUMPTION 1. ***Positivity of* $\mathbf{C}$ *and non-negativity of* $\mathbf{D}$.** *We suppose that, always, $\mathbf{C}(\mathbf{X}) > 0$ for all $\mathbf{X} \in [0^N, +\infty^N)$ and $\mathbf{D}(\mathbf{Y}) \geq 0$ for all $\mathbf{Y} \in [0^N, +\infty^N)$.*

ASSUMPTION 2. ***Boundedness**. We suppose always (i) that $\mathbf{D}$ is bounded and (ii) that $\mathbf{C}$ is bounded on bounded sets.*

DEFINITION 1.5. A function $\mathbf{F}(.)$ is *monotone* if and only if:

$$[\mathbf{F}(\mathbf{X}^1) - F(\mathbf{X}^2)]^T(\mathbf{X}^1 - \mathbf{X}^2) \geq 0$$

for all $\mathbf{X}^1 \in [0^N, +\infty^N)$ and $\mathbf{X}^2 \in [0^N, +\infty^N)$.

ASSUMPTION 3. ***Monotonicity**. We usually suppose that $\mathbf{C} : [0^N, +\infty^N) \to [0^N, +\infty^N)$ is monotone. We also suppose that $-\mathbf{D}$ is monotone on $[0^K, +\infty^K)$.*

REMARK 1.6. It is easy to show that if $\mathbf{C}$ and $-\mathbf{D}$ are both monotone then

$$\begin{pmatrix} \mathbf{C}(\mathbf{X}) - \mathbf{S}(\mathbf{Y}) \\ \mathbf{T}(\mathbf{X}) - \mathbf{D}(\mathbf{Y}) \end{pmatrix}$$

is also a monotone function of $\begin{pmatrix} \mathbf{X} \\ \mathbf{Y} \end{pmatrix} \in [0^N, +\infty^N) \times [0^K, +\infty^K)$.

ASSUMPTION 4. ***Continuous differentiability**. We also usually suppose that $\mathbf{C}(.)$ and $\mathbf{D}(.)$ are differentiable throughout their domains and that their derivatives $\mathbf{C}'(.)$ and $\mathbf{D}'(.)$ are continuous.*

1.4. Variable demand equilibrium

DEFINITION 1.7. Suppose we are given a (flow-vector, cost-vector) pair $(\mathbf{X}, \mathbf{Y})^T$ in which all $\mathbf{X}_{ijr} > 0$ and all $\mathbf{Y}_{ij} > 0$. Then $(\mathbf{X}, \mathbf{Y})^T$ will be called an *equilibrium* if and only if:

$$(\mathbf{X}, \mathbf{Y})^T \in [0^N, +\infty^N) \times [0^K, +\infty^K);$$
$$\forall\, ij,\ \mathbf{D}_{ij}(\mathbf{Y}) = \mathbf{T}_{ij}(\mathbf{X}) = \sum_r \mathbf{X}_{ijr}; \text{ and}$$
$$\forall\, ijr,\ \mathbf{C}_{ijr}(\mathbf{X}) = \mathbf{S}_{ijr}(\mathbf{Y}).$$

DEFINITION 1.8. A (flow-vector, cost-vector) pair $(\mathbf{X}, \mathbf{Y})^T$ is in *variable demand equilibrium* if and only if:

$$(\mathbf{X}, \mathbf{Y})^T \in [0^N, +\infty^N) \times [0^K, +\infty^K);$$
$$\mathbf{D}_{ij}(\mathbf{Y}) - \mathbf{T}_{ij}(\mathbf{X}) = 0\ \forall\, ij; \text{ and}$$
$$\mathbf{S}_{ijr}(\mathbf{Y}) - \mathbf{C}_{ijr}(\mathbf{X}) = 0\ \forall\, ijr. \tag{1.1}$$

The equilibrium equations (1.1) may be written in vector form:

$$\begin{pmatrix} \mathbf{D}(\mathbf{Y}) - \mathbf{T}(\mathbf{X}) \\ \mathbf{S}(\mathbf{Y}) - \mathbf{C}(\mathbf{X}) \end{pmatrix} = 0.$$

Here we are assuming initially that at equilibrium all $X_{ijr} > 0$ and all $Y_{ij} > 0$. To relax this initial assumption and so to allow the possibility that a listed route may have a high cost and zero flow at equilibrium we revise (1.1) to introduce a partial complementarity condition. The revised equilibrium condition is as follows. For each ijr:

$$\mathbf{Y}_{ij} - \mathbf{C}_{ijr}(\mathbf{X}) \leq 0\ \wedge\ \mathbf{Y}_{ij} - \mathbf{C}_{ijr}(\mathbf{X}) < 0 \Rightarrow \mathbf{X}_{ijr} = 0; \text{ and}$$
$$\mathbf{D}_{ij}(\mathbf{Y}) - \mathbf{T}_{ij}(\mathbf{X}) = 0. \tag{1.2}$$

REMARK 1.9. $\mathbf{D}_{ij}(\mathbf{Y}) - \mathbf{T}_{ij}(\mathbf{X}) = 0$ may be similarly replaced by $\mathbf{D}_{ij}(\mathbf{Y}) - \mathbf{T}_{ij}(\mathbf{X}) \leq 0$ and $\mathbf{D}_{ij}(\mathbf{Y}) - \mathbf{T}_{ij}(\mathbf{X}) < 0 \Rightarrow \mathbf{Y}_{ij} = 0$. This change would make (1.2) exactly into a complementary problem. We choose not to make this change here as it proves to be unnecessary under our assumptions above.

1.4.1. A simplification. Without consideration for its implication on the accuracy of the traffic model, we will make a simplification to this quite intimidating set of equations. From here on in this document we will assume that there is precisely one route per OD-pair so that there will be exactly one cost associated with each route-flow and as such each OD-pair.

This simplification has the obvious effect of setting $\mathbf{T}(\mathbf{X}) = \mathbf{X}$ and $\mathbf{S}(\mathbf{Y}) = \mathbf{Y}$ because clearly now the total flow along each OD-pair is just the same as its one associated route. This condensing of notation alters our equilibrium equations (1.1) to the much friendlier form below

$$\begin{pmatrix} \mathbf{D}(\mathbf{Y}) - \mathbf{X} \\ \mathbf{Y} - \mathbf{C}(\mathbf{X}) \end{pmatrix} = 0.$$

So what we need to know now is how to calculate the solutions to (1.2) with the much simpler equations here.

1.5. How to find equilibria

In this document we will examine iterative algorithms that maintain a current solution. At each step they will attempt to improve upon it by either perturbing it slightly or by following some sort of heuristic-based search direction. The ideal

outcome of this will be to, hopefully quickly, move closer and closer toward the actual solution vector, $\mathbf{X}^{\text{sol}}$.

CHAPTER 2

Linear cost and demand functions

In this section we shall examine linear cost and demand functions to facilitate theoretical analysis of the methods we apply in finding variable demand equilibrium.

2.1. Defining our linear equations

DEFINITION 2.1. We define our linear cost and demand functions $\mathbf{C}(.)$ and $\mathbf{D}(.)$ as follows

$$\mathbf{C}(\mathbf{X}) = \mathbf{A}_0 + A\mathbf{X}$$
$$\mathbf{D}(\mathbf{Y}) = \mathbf{B}_0 - B\mathbf{Y}$$

where $\mathbf{A}_0$, $\mathbf{B}_0 \in \mathbb{R}^N$ and A, $B \in M_N(\mathbb{R})$.

We now define three instances of this problem that we will use in our analysis of the minimisation techniques that will be discussed later. In each of the problems we assume a network with two routes of the form shown in Fig. 2.1.

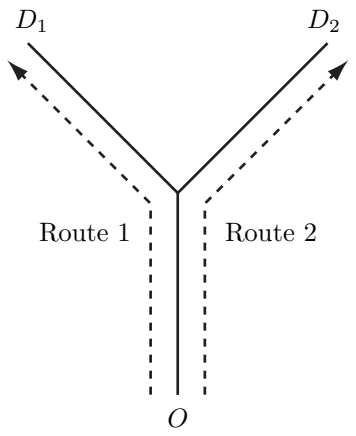


Figure 2.1: A network with two routes

Problem 1

$$A = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}, \quad \mathbf{A}_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mathbf{B}_0 = \begin{pmatrix} 2 \\ 3 \end{pmatrix} \tag{2.1}$$

With solution

$$\mathbf{X}^{\text{sol}} = \begin{pmatrix} \frac{2}{3} \\ 1 \end{pmatrix}, \quad \mathbf{Y}^{\text{sol}} = \begin{pmatrix} \frac{4}{3} \\ 2 \end{pmatrix}$$

Problem 2

$$A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}, \quad \mathbf{A}_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 0 & 31 \end{pmatrix}, \quad \mathbf{B}_0 = \begin{pmatrix} 4 \\ 94 \end{pmatrix} \tag{2.2}$$

With solution

$$\mathbf{X}^{\text{sol}} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \mathbf{Y}^{\text{sol}} = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$$

Problem 3

$$A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}, \quad \mathbf{A}_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}, \quad \mathbf{B}_0 = \begin{pmatrix} 4 \\ 7 \end{pmatrix} \tag{2.3}$$

With solution

$$\mathbf{X}^{\text{sol}} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \mathbf{Y}^{\text{sol}} = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$$

2.2. An outline of an algorithm

The algorithms we investigate will all be based, to an extent, upon the following idea. Define some objective function $V(\mathbf{X})$ constructed such that its global minimum (usually made to be zero) lies at the point of variable demand equilibrium for $\mathbf{C}(\mathbf{X})$ and $\mathbf{D}(\mathbf{Y})$. i.e.

$$V(\mathbf{X}^{\text{sol}}) = 0$$
$$\Updownarrow$$
$$\mathbf{Y}^{\text{sol}} - \mathbf{C}(\mathbf{X}^{\text{sol}}) = 0 \text{ and } \mathbf{D}(\mathbf{Y}^{\text{sol}}) - \mathbf{X}^{\text{sol}} = 0$$

We assume our algorithm starts at a given initial point $\mathbf{X}^0$ and moves along a path of consecutive points $\{\mathbf{X}^0, \mathbf{X}^1, \mathbf{X}^2, \ldots\}$. This happens by the following process, assuming we are currently at the point $\mathbf{X}^n$:

(1) The algorithm defines a search direction $\mathbf{\Delta}(\mathbf{X}^n) \in \mathbb{R}^N$;
(2) From the point $\mathbf{X}^n$ we perform a line minimisation of the function
$$\lambda \mapsto V(\mathbf{X}^n + \lambda \mathbf{\Delta}(\mathbf{X}^n))$$
in λ to find $\lambda^{\min}$;
(3) We then set $\mathbf{X}^{n+1} \leftarrow \mathbf{X}^n + \lambda^{\min}\mathbf{\Delta}(\mathbf{X}^n)$;
(4) We determine if we have moved significantly with a certain tolerance and if we consider the algorithm to have stalled then we stop. Otherwise we go back to (1).

A single step of this basic algorithm is illustrated in Fig. 2.2.

We can now see that the defining components of a basic algorithm are:

- The objective function $V(\mathbf{X})$;
- The search direction $\mathbf{\Delta}(\mathbf{X})$.

2.3. Algorithm 1

Algorithm 1 takes its search direction taken from the now considered defunct 'cobweb' algorithm defined thus

$$\mathbf{\Delta}^{\text{A1}}(\mathbf{X}) = \mathbf{D}(\mathbf{C}(\mathbf{X})) - \mathbf{X}. \tag{2.4}$$

Clearly when $\mathbf{\Delta}^{\text{A1}}(\mathbf{X}) = 0$ we are in variable demand equilibrium as per Definition 1.7.

The objective function of *Algorithm 1* is based upon the magnitude of it's search direction

$$V^{\text{A1}}(\mathbf{X}) = \frac{1}{2}\mathbf{\Delta}^{\text{A1}}(\mathbf{X}) \cdot \mathbf{\Delta}^{\text{A1}}(\mathbf{X}). \tag{2.5}$$

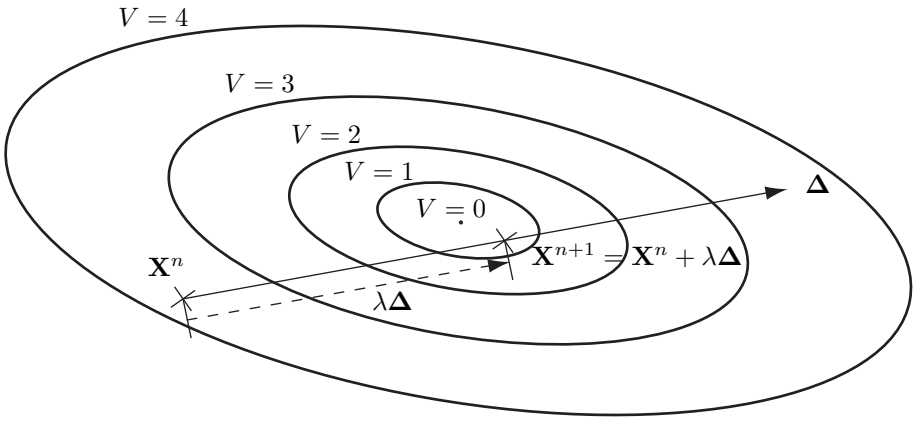


Figure 2.2: One step of a basic search algorithm in two dimensions

THEOREM 2.2. *(2.4) is always a descent direction for (2.5) if the matrix*

$$I - [\nabla_{\mathbf{X}}\mathbf{D}(\mathbf{C}(\mathbf{X}))]^{\top}, \tag{2.6}$$

is strictly positive definite.

PROOF. First calculating the gradient of $V^{\mathrm{A1}}(\mathbf{X})$

$$\begin{aligned}
\frac{\partial\left(V^{\mathrm{A1}}\right)}{\partial\mathbf{X}_r} &= \frac{1\!\!\!/}{2\!\!\!/} \times 2\!\!\!/ \frac{\partial\left(\boldsymbol{\Delta}^{\mathrm{A1}}\right)}{\partial\mathbf{X}_r} \cdot \boldsymbol{\Delta}^{\mathrm{A1}}, \\
\Rightarrow \nabla_{\mathbf{X}} V^{\mathrm{A1}} = \frac{\partial\left(V^{\mathrm{A1}}\right)}{\partial\mathbf{X}} &= \frac{1\!\!\!/}{2\!\!\!/} \times 2\!\!\!/ \times \left(\nabla_{\mathbf{X}}\boldsymbol{\Delta}^{\mathrm{A1}}\right)^{\top} \boldsymbol{\Delta}^{\mathrm{A1}}, \\
&= \left[(\nabla_{\mathbf{X}}\mathbf{D}(\mathbf{C}(\mathbf{X})))^{\top} - (\nabla_{\mathbf{X}}\mathbf{X})^{\top}\right] \boldsymbol{\Delta}^{\mathrm{A1}}, \\
&= \left[(\nabla_{\mathbf{X}}\mathbf{D}(\mathbf{C}(\mathbf{X})))^{\top} - I^{\top}\right] \boldsymbol{\Delta}^{\mathrm{A1}}, \\
&= \left[(\nabla_{\mathbf{X}}\mathbf{D}(\mathbf{C}(\mathbf{X})))^{\top} - I\right] \boldsymbol{\Delta}^{\mathrm{A1}}.
\end{aligned}$$

Now the directional derivative in the direction $\boldsymbol{\Delta}^{\mathrm{A1}}$ is

$$\begin{aligned}
\nabla_{\mathbf{X}} V^{\mathrm{A1}} \cdot \boldsymbol{\Delta}^{\mathrm{A1}} &= \left\{\left[(\nabla_{\mathbf{X}}\mathbf{D}(\mathbf{C}(\mathbf{X})))^{\top} - I\right] \boldsymbol{\Delta}^{\mathrm{A1}}\right\} \cdot \boldsymbol{\Delta}^{\mathrm{A1}}, \\
&= -\left(\boldsymbol{\Delta}^{\mathrm{A1}}\right)^{\top} \left(I - [\nabla_{\mathbf{X}}\mathbf{D}(\mathbf{C}(\mathbf{X}))]^{\top}\right) \boldsymbol{\Delta}^{\mathrm{A1}}, \\
&< 0,
\end{aligned}$$

if $I - [\nabla_{\mathbf{X}}\mathbf{D}(\mathbf{C}(\mathbf{X}))]^{\top}$ is strictly positive definite, which we have assumed. Thus $\boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X})$ is a descent direction for $V^{\mathrm{A1}}(\mathbf{X})$. □

REMARK 2.3. It should be noted that in several previous works [**6, 7**] that the gradient $\nabla_{\mathbf{X}} V^{\mathrm{A1}}$ has been calculated *incorrectly* as

$$\nabla_{\mathbf{X}} V^{\mathrm{A1}} = [\nabla_{\mathbf{X}}\mathbf{D}(\mathbf{C}(\mathbf{X})) - I] \boldsymbol{\Delta}^{\mathrm{A1}};$$

notice the missing transpose. This has potentially created spurious results, although if $\nabla_{\mathbf{X}} V^{\mathrm{A1}}$ happens to be symmetric, which it commonly is, then no difference will result in many calculations.

Even in our simple case with linear $\mathbf{C}(.)$ and $\mathbf{D}(.)$, unfortunately we cannot be certain that the conditions for Theorem 2.2 will hold. This is shown below

$$\begin{aligned} I - [\nabla_{\mathbf{X}}\mathbf{D}(\mathbf{C}(\mathbf{X}))]^{\top} &= I - \{\nabla_{\mathbf{X}}[(\mathbf{B}_0 - B\mathbf{A}_0) + (-BA)\mathbf{X}]\}^{\top}, \\ &= I + (BA)^{\top}, \\ &= I + A^{\top}B^{\top} \end{aligned}$$

now let $\mathbf{x} \in \mathbb{R}^N$, then

$$\begin{aligned} \mathbf{x}^{\top}(I + A^{\top}B^{\top})\mathbf{x} &= \mathbf{x}^{\top}\mathbf{x} + \mathbf{x}^{\top}A^{\top}B^{\top}\mathbf{x}, \\ &= \underbrace{\mathbf{x}^{\top}\mathbf{x}}_{\geq 0} + \underbrace{\mathbf{x}^{\top}BA\mathbf{x}}_{\text{who knows?}}. \end{aligned}$$

so this in turn depends upon the positive definiteness of BA. We may believe that logically the product of two strictly positive definite matrices is itself positive definite. Sadly this is not the case and in-fact problem 2 (2.2) was chosen because it is an instance of this phenomenon and we will see shortly the problems this causes.

2.3.1. Explicit calculation of λ. In this simple case of linear cost and demand functions it is possible to explicitly calculate the exact value of λ that minimises

$$\begin{aligned} V^{\mathrm{A1}}(\mathbf{X}^n + \lambda\boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}^n)) &= \frac{1}{2}\boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}^n + \lambda\boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}^n)) \cdot \boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}^n + \lambda\boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}^n)) \\ &= \frac{1}{2}\sum_{i=1}^{N}\left[\left(\boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}^n + \lambda\boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}^n))\right)_i\right]^2 \\ &= \frac{1}{2}\sum_{i=1}^{N}\left[\left(\boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}^n)\right)_i - \lambda\left((BA + I)\boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}^n)\right)_i\right]^2 \end{aligned}$$

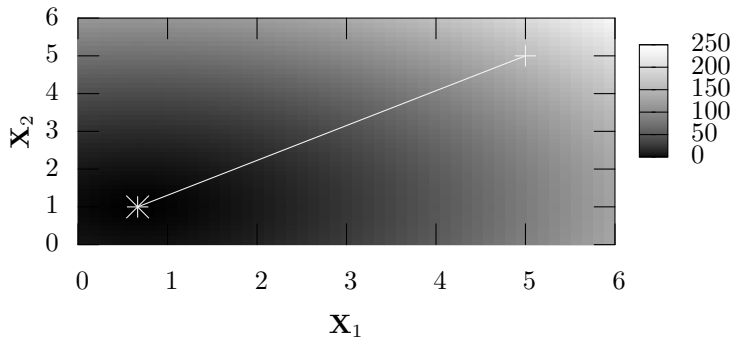
now label $\mathbf{P} = \boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}^n)$ and $\mathbf{Q} = (BA + I)\mathbf{P}$, differentiate and set equal zero

$$\begin{aligned} \frac{1}{2}\frac{d}{d\lambda}\sum_{i=1}^{N}(\mathbf{P}_i - \lambda\mathbf{Q}_i) &= \frac{\cancel{1}}{\cancel{2}} \times -\cancel{2} \times \sum_{i=1}^{N}\mathbf{Q}_i(\mathbf{P}_i - \lambda\mathbf{Q}_i) \\ &= \sum_{i=1}^{N}\mathbf{Q}_i(\lambda\mathbf{Q}_i - \mathbf{P}_i) \\ &= \lambda\sum_{i=1}^{N}{\mathbf{Q}_i}^2 - \sum_{i=1}^{N}\mathbf{P}_i\mathbf{Q}_i = 0 \\ \Rightarrow \lambda &= \frac{\sum_{i=1}^{N}\mathbf{P}_i\mathbf{Q}_i}{\sum_{i=1}^{N}{\mathbf{Q}_i}^2} \end{aligned}$$

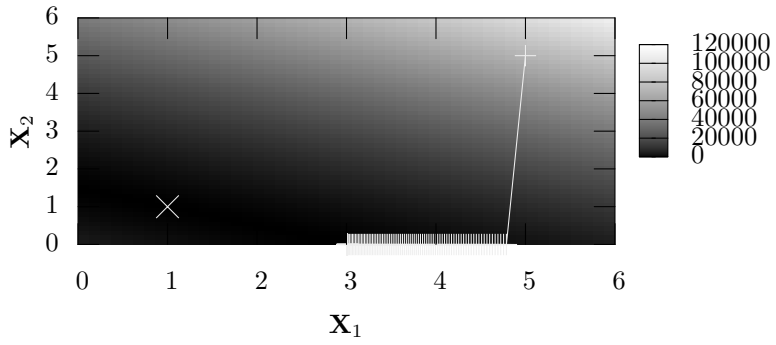
so we have found the optimal λ to be

$$\lambda^{\min} = \frac{\mathbf{P}\cdot\mathbf{Q}}{\mathbf{Q}\cdot\mathbf{Q}} \tag{2.7}$$

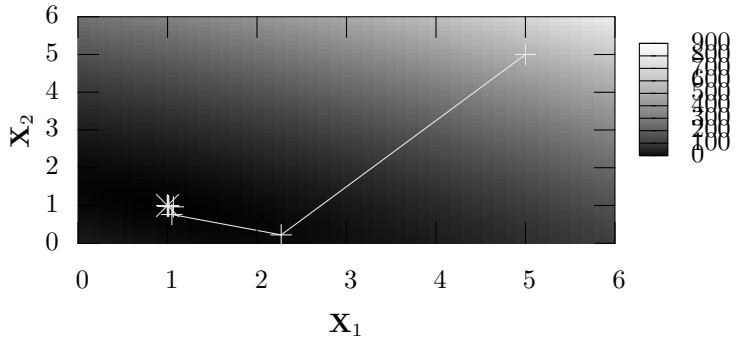
So at each step of the algorithm we simply choose the optimal λ using (2.7).



(a) Problem 1



(b) Problem 2



(c) Problem 3

Figure 2.3: Algorithm 1 using optimal steps length on linear flow/cost functions

Test with problem 1. The algorithm was started at the point $\mathbf{X}^0 = (5,\, 5)^\top$ with $\mathbf{C}(.)$ and $\mathbf{D}(.)$ defined as in (2.1). The results of this can be seen in Fig. 2.3a. It actually converges in a single iteration of the algorithm. This happens because we are choosing the optimal step-length A having a zero alternate diagonal means the search direction points directly at the solution.

Test with problem 2. $\mathbf{C}(.)$ and $\mathbf{D}(.)$ defined as in (2.2). If we remember the values of A, $\mathbf{A}_0$, B and $\mathbf{B}_0$ were chosen because *algorithm 1* requires

$$I - \nabla_{\mathbf{X}}\mathbf{D}(\mathbf{C}(\mathbf{X})) = I + BA$$

in turn requiring BA to be positive definite in order for $\boldsymbol{\Delta}^{\mathrm{A1}}$ to be a descent direction for V^{A1}. Specifically here

$$A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 0 & 31 \end{pmatrix} \quad \Longrightarrow \quad BA = \begin{pmatrix} 2 & 1 \\ 31 & 62 \end{pmatrix}$$

Now if we take $\mathbf{X} = (3 - 2t, t)^\top$, $t \in [0,\, 1.5]$ then

$$\begin{aligned} \mathbf{D}(\mathbf{C}(\mathbf{X})) - \mathbf{X} &= (\mathbf{B}_0 - \cancel{B\mathbf{A}_0}) - (BA)\mathbf{X} \\ &= \begin{pmatrix} 4 \\ 94 \end{pmatrix} - \begin{pmatrix} 2 & 1 \\ 31 & 62 \end{pmatrix} \begin{pmatrix} 3 - 2t \\ t \end{pmatrix} - \begin{pmatrix} 3 - 2t \\ t \end{pmatrix} \\ &= \begin{pmatrix} 4 - 6 + 4t - t - 3 + 2t \\ 94 - 93 + \cancel{62t - 62t} - t \end{pmatrix} \\ &= \begin{pmatrix} 5t - 5 \\ 1 - t \end{pmatrix} \end{aligned}$$

now the directional derivative

$$\begin{aligned} \nabla_{\mathbf{X}} V^{\mathrm{A1}} \begin{pmatrix} 5t - 5 \\ 1 - t \end{pmatrix} \cdot \begin{pmatrix} 5t - 5 \\ 1 - t \end{pmatrix} &= - \begin{pmatrix} 5t - 5 \\ 1 - t \end{pmatrix}^\top [I - BA] \begin{pmatrix} 5t - 5 \\ 1 - t \end{pmatrix} \\ &= 22(1 - 2t + t^2) \\ &= 22(t - 1)^2 \\ &> 0 \; \forall \; t \; \in \; [0,\, 1.5] \end{aligned}$$

So we have found a whole line of points along which the search direction $\boldsymbol{\Delta}^{\mathrm{A1}}$ is not a descent direction. It was predicted that close to this line that *algorithm 1* would fail because the search direction it produced would point uphill.

In order to see what happens *algorithm 1* was started at the point $\mathbf{X}^0 = (5,\, 5)^\top$. The path it followed is illustrated in Fig. 2.3b. In the first step it chooses a reasonable direction, however selecting the optimal step-length leads it all the way to the bottom of the space. From this point the search direction $\boldsymbol{\Delta}^{\mathrm{A1}}$ points off the bottom of the search space. However in the implementation the downward component of the search direction is clipped, so that it points directly left. It then proceeds along the lower boundary of the search space until it reaches the point $(3,\, 0)^\top$, where the search direction becomes uphill so the algorithm stalls.

Now in order to rule out the effects of the step-length selection technique a second test was performed where the step-length was fixed at $\lambda = 0.001$ and the algorithm started at several different locations. As can be seen in Fig. 2.4 the line $y = \frac{1}{2}(3 - x)$ appears to attract all of the trajectories toward it. On this line, as we predicted the trajectories are going uphill, however they are still pointing in the direction of the solution. What we do see, though, is that all the trajectories reach the solution eventually if we use small enough step-lengths.

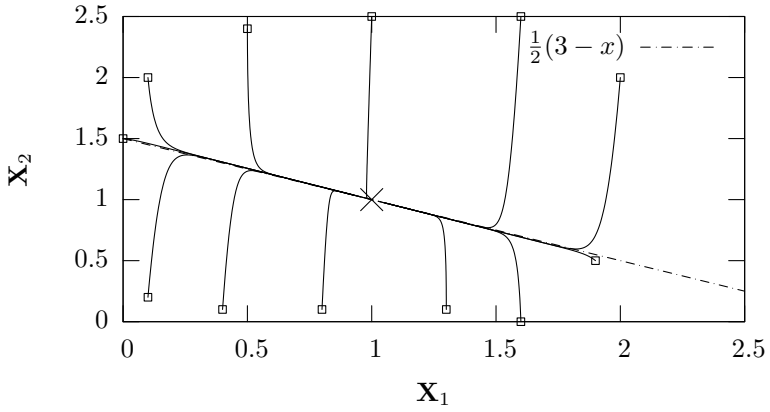


Figure 2.4: Several trajectories of algorithm 1 with a very small fixed step-length ($\lambda = 0.001$)

Test with problem 3. The algorithm was started at the point $\mathbf{X}^0 = (5,\, 5)^\top$ with $\mathbf{C}(.)$ and $\mathbf{D}(.)$ defined as in (2.3). The results of this can be seen in Fig. 2.3c.

Originally this problem caused *algorithm 1* to fail in one or two steps, as it passed off the bottom of the search space. More precisely minimised value of λ was corresponding to a point outside the 'safe' region $\mathbf{X} \in \mathbb{R}^N_+$. This situation had to be rectified, and so a method of containing the search to the 'safe' positive region was developed. It was decided that the path between the point outside $\mathbb{R}^N_+$ and the previous point would be traced along until a point was found just inside $\mathbb{R}^N_+$. Then a corresponding value of λ would be calculated corresponding to a movement of that length.

As can be seen in Fig. 2.3c, this method allows the search to 'stay safe' and subsequently find the solution. The method is described in more detail in Section 2.7.

2.4. Most obvious

The so-called *most obvious* algorithm requires us to extend our basic algorithm structure described earlier. In addition to considering the flow vector $\mathbf{X}$, it also takes into account the cost vector $\mathbf{Y}$. This allows it to "leave the tramlines" as it were; the search need not necessarily take place upon the surface of the objective function $V(\mathbf{X})$. This requires us to update our notation slightly. We have a higher dimensional function to minimise, specifically

$$V^{\mathrm{MO}}(\mathbf{X},\, \mathbf{Y}) = \boldsymbol{\Delta}^{\mathrm{MO}}_{\mathbf{X}} \cdot \boldsymbol{\Delta}^{\mathrm{MO}}_{\mathbf{X}} + \boldsymbol{\Delta}^{\mathrm{MO}}_{\mathbf{Y}} \cdot \boldsymbol{\Delta}^{\mathrm{MO}}_{\mathbf{Y}}, \tag{2.8}$$

where $\boldsymbol{\Delta}^{\mathrm{MO}}_{\mathbf{X}}(\mathbf{X},\, \mathbf{Y})$ and $\boldsymbol{\Delta}^{\mathrm{MO}}_{\mathbf{Y}}(\mathbf{X},\, \mathbf{Y})$ arrive from the search direction of

$$\boldsymbol{\Delta}^{\mathrm{MO}}(\mathbf{X},\, \mathbf{Y}) = \begin{pmatrix} \boldsymbol{\Delta}^{\mathrm{MO}}_{\mathbf{X}}(\mathbf{X},\, \mathbf{Y}) \\ \boldsymbol{\Delta}^{\mathrm{MO}}_{\mathbf{Y}}(\mathbf{X},\, \mathbf{Y}) \end{pmatrix} = \begin{pmatrix} \mathbf{Y} - \mathbf{C}(\mathbf{X}) \\ \mathbf{D}(\mathbf{Y}) - \mathbf{X} \end{pmatrix}. \tag{2.9}$$

Clearly when $\boldsymbol{\Delta}^{\mathrm{MO}}(\mathbf{X},\, \mathbf{Y}) = 0$ we are in variable demand equilibrium as per Definition 1.7 and elsewhere the sum of two squares must be positive.

THEOREM 2.4. *(2.9) is always a descent direction for (2.8) if the cost and demand functions* $\mathbf{C}(.)$ *and* $\mathbf{D}(.)$ *are both monotone.*

PROOF. Assume we are away from equilibrium, so that either $\boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}}$ or $\boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}$ is be non-zero and additionally that $\mathbf{C}(.)$ and $\mathbf{D}(.)$ are both monotone. Now differentiating $V^{\mathrm{MO}}(\mathbf{X}, \mathbf{Y})$

$$\begin{aligned}
\frac{\partial \left(V^{\mathrm{MO}}\right)}{\partial \mathbf{X}_r} &= 2\frac{\partial \left(\boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}}\right)}{\partial \mathbf{X}_r} \cdot \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}} + 2\frac{\partial \left(\boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}\right)}{\partial \mathbf{X}_r} \cdot \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} \\
\Rightarrow \frac{\partial \left(V^{\mathrm{MO}}\right)}{\partial \mathbf{X}} &= 2\left(\nabla_{\mathbf{X}} \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}}\right) \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}} + 2\left(\nabla_{\mathbf{X}} \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}\right) \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} \\
&= 2\left[\cancel{\nabla_{\mathbf{X}} \overset{0}{\mathbf{Y}}} - \cancel{\nabla_{\mathbf{X}} \overset{A}{\mathbf{C}}(\mathbf{X})}\right] \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}} + 2\left[\cancel{\nabla_{\mathbf{X}} \overset{0}{\mathbf{D}}(\mathbf{Y})} - \cancel{\nabla_{\mathbf{X}} \overset{I}{\mathbf{X}}}\right] \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} \\
&= -2\left(A\boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}} + \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}\right)
\end{aligned}$$

similarly

$$\begin{aligned}
\frac{\partial \left(V^{\mathrm{MO}}\right)}{\partial \mathbf{Y}} &= 2\left(\nabla_{\mathbf{Y}} \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}}\right) \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}} + 2\left(\nabla_{\mathbf{Y}} \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}\right) \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} \\
&= 2\left[\cancel{\nabla_{\mathbf{Y}} \overset{I}{\mathbf{Y}}} - \cancel{\nabla_{\mathbf{Y}} \overset{0}{\mathbf{C}}(\mathbf{X})}\right] \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}} + 2\left[\cancel{\nabla_{\mathbf{Y}} \overset{-B}{\mathbf{D}}(\mathbf{Y})} - \cancel{\nabla_{\mathbf{Y}} \overset{0}{\mathbf{X}}}\right] \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} \\
&= 2\left(\boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}} - B\boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}\right)
\end{aligned}$$

now

$$\begin{aligned}
\text{grad } V^{\mathrm{MO}} \cdot \boldsymbol{\Delta}^{\mathrm{MO}} &= \frac{\partial \left(V^{\mathrm{MO}}\right)}{\partial \mathbf{X}} \cdot \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}} + \frac{\partial \left(V^{\mathrm{MO}}\right)}{\partial \mathbf{Y}} \cdot \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} \\
&= -2\left(A\boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}} + \cancel{\boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}}\right) \cdot \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}} + 2\left(\cancel{\boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}}} - B\boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}\right) \cdot \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} \\
&= -2\left[\underbrace{\left(\boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}}\right)^{\top} A\boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}}}_{\geq 0} + \underbrace{\left(\boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}\right)^{\top} B\boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}}_{\geq 0}\right].
\end{aligned}$$

Each of these components are ≥ 0 by our assumptions of A and B being positive definite. Additionally at least one of them must be > 0 by our assumption of being away from equilibrium. Therefore our directional derivative is negative, thus $\boldsymbol{\Delta}^{\mathrm{MO}}$ is a descent direction for V^{MO}. □

2.4.1. An updated algorithm. The algorithm starts at a given initial tuple $(\mathbf{X}^0, \mathbf{Y}^0)$ and moves along a path of consecutive tuples $\{(\mathbf{X}^0, \mathbf{Y}^0), (\mathbf{X}^1, \mathbf{Y}^1), (\mathbf{X}^2, \mathbf{Y}^2), \ldots\}$. This happens by the following process, assuming we are currently at the point $(\mathbf{X}^n, \mathbf{Y}^n)$:

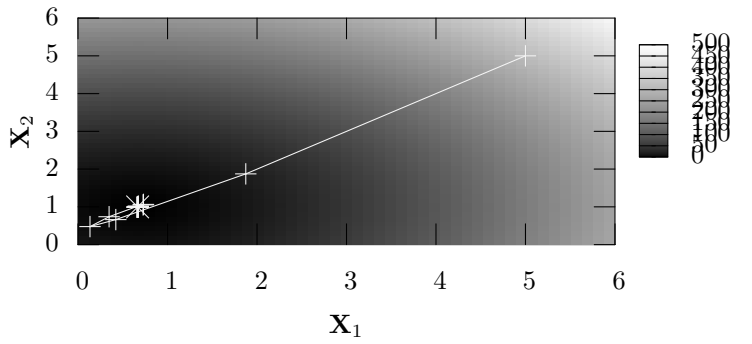
(1) We calculate the search direction $\delta \leftarrow \boldsymbol{\Delta}^{\mathrm{MO}}(\mathbf{X}^n, \mathbf{Y}^n) \in (\mathbb{R}^n, \mathbb{R}^n)$;
(2) From the tuple $(\mathbf{X}^n, \mathbf{Y}^n)$ we perform a line minimisation of the function

$$\lambda \mapsto V^{\mathrm{MO}}((\mathbf{X}^n, \mathbf{Y}^n) + \lambda\delta) \tag{2.10}$$

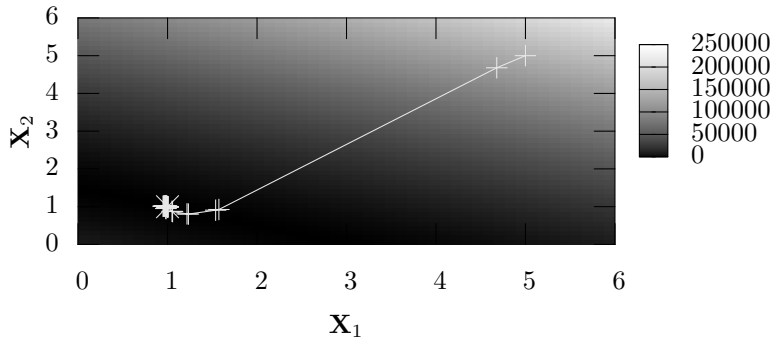
in λ to find $\lambda^{\min}$;
(3) We then set

$$(\mathbf{X}^{n+1}, \mathbf{Y}^{n+1}) \leftarrow (\mathbf{X}^n, \mathbf{Y}^n) + \lambda^{\min}\delta;$$

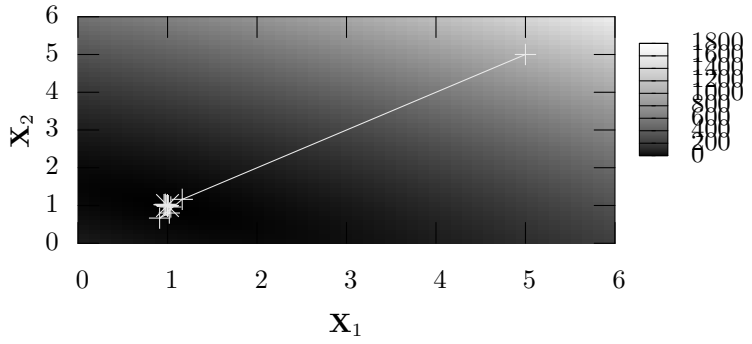
(4) We determine if we have moved significantly with a certain tolerance and if we consider the algorithm to have stalled then we stop. Otherwise we go back to (1).



(a) Problem 1



(b) Problem 2



(c) Problem 3

Figure 2.5: 'Most obvious' using optimal steps length on linear flow/cost functions

2.4.2. Explicit calculation of λ. As with *algorithm 1*, when the cost and demand functions $\mathbf{C}(.)$ and $\mathbf{D}(.)$ being used are linear, it is possible to explicitly calculate the ideal value of λ to minimise (2.10). Expanding this we get

$$\sum_{i=1}^{N} \left[\mathbf{\Delta}_{\mathbf{X}}^{\mathrm{MO}}((\mathbf{X}^n,\, \mathbf{Y}^n) + \lambda \mathbf{\Delta}^{\mathrm{MO}})_i^{\,2} + \mathbf{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}((\mathbf{X}^n,\, \mathbf{Y}^n) + \lambda \mathbf{\Delta}^{\mathrm{MO}})_i^{\,2} \right]$$

which, after a significant amount of slogging can be separated out into a quadratic in terms of λ, i.e.

$$V^{\mathrm{MO}}((\mathbf{X}^n,\, \mathbf{Y}^n) + \lambda \mathbf{\Delta}^{\mathrm{MO}}) = a\lambda^2 + b\lambda + c.$$

The coefficients are as follows

$$\begin{aligned} a &= ||A\mathbf{\Delta}_{\mathbf{X}}^{\mathrm{MO}} - \mathbf{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}||^2 + ||B\mathbf{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} + \mathbf{\Delta}_{\mathbf{X}}^{\mathrm{MO}}||^2; \\ b &= -2\left[\mathbf{\Delta}_{\mathbf{X}}^{\mathrm{MO}} \cdot \left(A\mathbf{\Delta}_{\mathbf{X}}^{\mathrm{MO}} - \mathbf{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}\right) + \mathbf{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} \cdot \left(B\mathbf{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} + \mathbf{\Delta}_{\mathbf{X}}^{\mathrm{MO}}\right)\right]; \\ c &= (\mathbf{\Delta}_{\mathbf{X}}^{\mathrm{MO}}) \cdot (\mathbf{\Delta}_{\mathbf{X}}^{\mathrm{MO}}) + (\mathbf{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}) \cdot (\mathbf{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}). \end{aligned}$$

The resultant quadratic clearly has a positive coefficient for λ^2 so we can find the minimum by simply differentiating with respect to λ and solving for it equal zero. Now

$$\begin{aligned} \frac{\partial(a\lambda^2 + b\lambda + c)}{\partial\lambda} &= 0 \\ 2a\lambda + b &= 0 \\ \lambda &= \frac{b}{-2a} \end{aligned} \tag{2.11}$$

So we can now calculate $\lambda^{\min}$ using (2.11).

Test with problem 1. The algorithm was started at the point $\mathbf{X}^0 = (5,\, 5)^\top$ with $\mathbf{C}(.)$ and $\mathbf{D}(.)$ defined as in (2.1). The results of this can be seen in Fig. 2.5a. This problem was designed to be easily solvable and we can see the search proceeds directly toward the solution. Strangely it does appear to go some distance past the solution and then back again. We can only conclude that the 'ridged' appearance of the objective function causes the minimum along the search path to go a little past the solution.

Test with problem 2. The algorithm was started at the point $\mathbf{X}^0 = (5,\, 5)^\top$ with $\mathbf{C}(.)$ and $\mathbf{D}(.)$ defined as in (2.2). The results of this can be seen in Fig. 2.5b.

If we remember this problem was chosen because the product BA is not positive definite and would therefore cause *algorithm 1* to trip. As we expected, for *algorithm 1* we found that it would not converge unless the step-length λ was fixed extremely small (~ 0.001). However with the *most obvious* method we find that it converges easily, proceeding directly towards the solution. From several other starting vectors the method also converged without problem. We already have reason to believe that this search direction is much better than that of *algorithm 1.*

Test with problem 3. The algorithm was started at the point $\mathbf{X}^0 = (5,\, 5)^\top$ with $\mathbf{C}(.)$ and $\mathbf{D}(.)$ defined as in (2.3). In Fig. 2.5c we can see its progress. It nearly hits the solution in a single step. This is lot better than *algorithm 1* which attempted to leave the 'safe' region of $\mathbb{R}_+^N$.

2.4.3. A flaw in the search direction. If we have constant cost and demand functions and only a single OD pair, i.e.

$$C(X) = C$$

and

$$D(Y) = D$$

i.e. $A = B = 0$, $A_0 = C$ and $B_0 = D$ then the directional derivative for the *most obvious* algorithm becomes (from Theorem 2.4)

$$\nabla_{\mathbf{X}} V^{\mathrm{MO}} = -2\left[\left(\boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}}\right)^{\top} \overset{0}{\cancel{A}} \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}} + \left(\boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}\right)^{\top} \overset{0}{\cancel{B}} \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}\right] = 0$$

So $\boldsymbol{\Delta}^{\mathrm{MO}}(\mathbf{X}, \mathbf{Y})$ is not a descent direction in the situation of un-congested rigid demand networks.

Clearly this is a bad thing and in 2004 M. Smith[**6**] suggested an alternative as the use of an alternative search direction to consider.

2.5. Two-direction search

So now we remedy the problem discovered above with the introduction of this second search direction we will call *alternate direction*. Notably this direction is chosen to be perpendicular to the *most obvious* direction

$$\boldsymbol{\Delta}^{\mathrm{AD}}(\mathbf{X}, \mathbf{Y}) = \begin{pmatrix} \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{AD}}(\mathbf{X}, \mathbf{Y}) \\ \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{AD}}(\mathbf{X}, \mathbf{Y}) \end{pmatrix} = \begin{pmatrix} \mathbf{D}(\mathbf{Y}) - \mathbf{X} \\ \mathbf{C}(\mathbf{X}) - \mathbf{Y} \end{pmatrix}. \tag{2.12}$$

We use the same objective as for the *most obvious* algorithm here (2.8), so again when $V^{\mathrm{MO}}(\mathbf{X}, \mathbf{Y}) = 0$ we are in equilibrium.

2.5.1. Revising our algorithm outline again. The algorithm starts at a given initial tuple $(\mathbf{X}^0, \mathbf{Y}^0)$ and moves along a path of consecutive tuples $\{(\mathbf{X}^0, \mathbf{Y}^0), (\mathbf{X}^1, \mathbf{Y}^1), (\mathbf{X}^2, \mathbf{Y}^2), \ldots\}$. This happens by the following process, assuming we are currently at the point $(\mathbf{X}^n, \mathbf{Y}^n)$:

(1) We calculate the search direction

$$\delta \leftarrow \begin{cases} \boldsymbol{\Delta}^{\mathrm{MO}}(\mathbf{X}^n, \mathbf{Y}^n), & \text{if } n \text{ is even;} \\ \boldsymbol{\Delta}^{\mathrm{AD}}(\mathbf{X}^n, \mathbf{Y}^n), & \text{if } n \text{ is odd.} \end{cases} \in (\mathbb{R}^n, \mathbb{R}^n);$$

(2) From the tuple $(\mathbf{X}^n, \mathbf{Y}^n)$ we perform a line minimisation of the function

$$\lambda \mapsto V^{\mathrm{MO}}((\mathbf{X}^n, \mathbf{Y}^n) + \lambda\delta)$$

in λ to find $\lambda^{\min}$;

(3) We then set

$$(\mathbf{X}^{n+1}, \mathbf{Y}^{n+1}) \leftarrow (\mathbf{X}^n, \mathbf{Y}^n) + \lambda^{\min}\delta;$$

(4) We determine if we have moved significantly with a certain tolerance and if we consider the algorithm to have stalled then we stop. Otherwise we go back to (1).

2.5.2. Explicit calculation of λ. Very similarly to *most obvious*, when the cost and demand functions $\mathbf{C}(.)$ and $\mathbf{D}(.)$ being used are linear, it is possible to explicitly calculate the ideal value of λ to minimise (2.10). Expanding this we get

$$\sum_{i=1}^{N} \left[\boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}}((\mathbf{X}^n, \mathbf{Y}^n) + \lambda \boldsymbol{\Delta}^{\mathrm{AD}})_i{}^2 + \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}((\mathbf{X}^n, \mathbf{Y}^n) + \lambda \boldsymbol{\Delta}^{\mathrm{AD}})_i{}^2 \right]$$

which, after a significant amount of slogging can be separated out into a quadratic in terms of λ, i.e.

$$V^{\mathrm{MO}}((\mathbf{X}^n, \mathbf{Y}^n) + \lambda \boldsymbol{\Delta}^{\mathrm{AD}}) = a\lambda^2 + b\lambda + c.$$

The coefficients are as follows

$$\begin{aligned}
a &= \left(A\boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} + \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}}\right)^2 + \left(\boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} - B\boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}}\right)^2; \text{ (squared means dot product)} \\
b &= -2\left[\boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}} \cdot \left(\boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} - B\boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}}\right) + \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} \cdot \left(A\boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} + \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}}\right)\right]; \\
c &= \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}} \cdot \boldsymbol{\Delta}_{\mathbf{X}}^{\mathrm{MO}} + \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}} \cdot \boldsymbol{\Delta}_{\mathbf{Y}}^{\mathrm{MO}}.
\end{aligned}$$

The resultant quadratic clearly has a positive coefficient for λ^2 so we can find the minimum by exactly the same method as we did for *most obvious* resulting in the equation (2.11).

Test with problem 1. The algorithm was started at the point $\mathbf{X}^0 = (5, 5)^\top$ with $\mathbf{C}(.)$ and $\mathbf{D}(.)$ defined as in (2.1). The results of this can be seen in Fig. 2.6a. This problem was designed to be easily solvable and we can see the search proceeds directly toward the solution. Where the *most obvious* algorithm went a little past the solution, the new *two-direction* method gets close to the solution and stays there.

Test with problem 2. The algorithm was started at the point $\mathbf{X}^0 = (5, 5)^\top$ with $\mathbf{C}(.)$ and $\mathbf{D}(.)$ defined as in (2.2). The results of this can be seen in Fig. 2.6b.

This problem was chosen in order to make *algorithm 1* fail and it succeeded in doing this. It was then demonstrated that the *most obvious* search direction alone could solve the problem successfully. Applying the *two-direction* method in this instance badly affects the performance of the algorithm. Instead of heading directly toward the solution, the trajectory almost hits the left side of the search space. It then takes a great number of further iterations to make its way to the solution. Clearly the alternate search direction $\boldsymbol{\Delta}^{\mathrm{AD}}$ is not ideally suited to problems of this nature.

Test with problem 3. The algorithm was started at the point $\mathbf{X}^0 = (5, 5)^\top$ with $\mathbf{C}(.)$ and $\mathbf{D}(.)$ defined as in (2.3). In Fig. 2.6c we can see its progress. Its performance is extremely comparable to that of the *most obvious* direction by itself.

2.6. Proper comparison

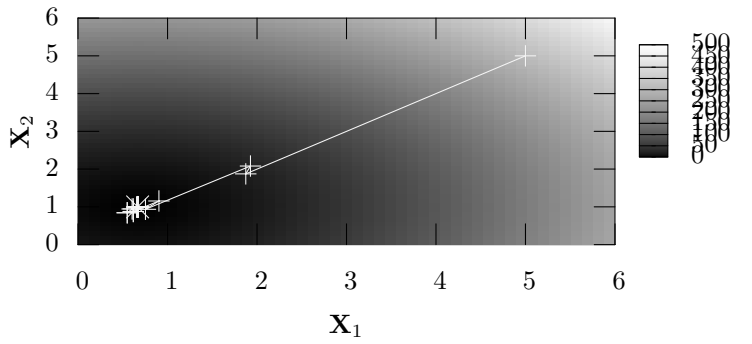
Here we take two networks in the form shown in Fig. 2.1 but with different cost and demand functions.

First comparison. We define

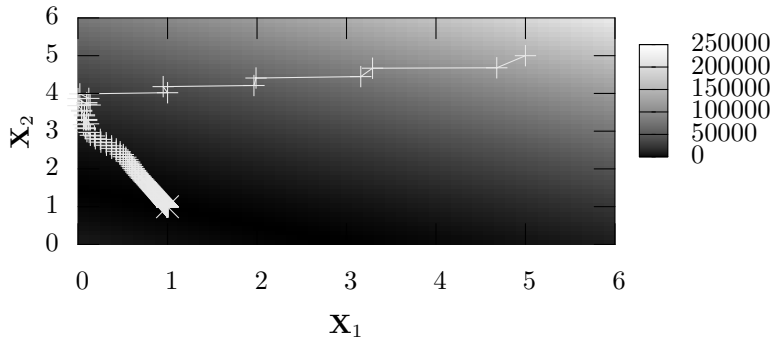
$$A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}, \quad \mathbf{A}_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 0 & k \end{pmatrix}, \quad \mathbf{B}_0 = \begin{pmatrix} 4 \\ 3k+1 \end{pmatrix} \tag{2.13}$$

so $-\mathbf{D}(.)$ still has a positive definite definite Jacobian B. Also the solution is still

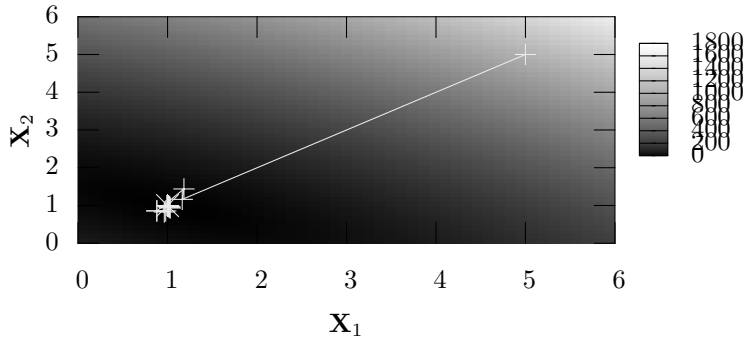
$$\mathbf{X}^{\mathrm{sol}} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \mathbf{Y}^{\mathrm{sol}} = \begin{pmatrix} 3 \\ 3 \end{pmatrix}.$$



(a) Problem 1



(b) Problem 2



(c) Problem 3

Figure 2.6: 'Two-direction' using optimal steps length on linear flow/cost functions

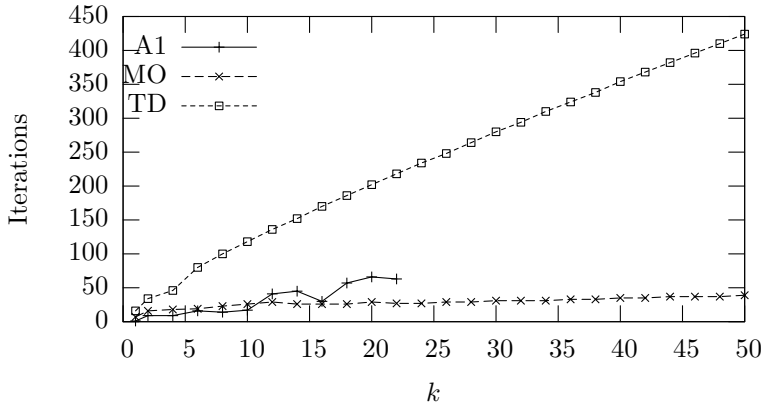


Figure 2.7: Tests with first problem using optimal step-lengths

Additionally for $k > 0$ the cost and demand functions are both separable, so assuming $k > 0$ the problem is still very standard mathematically. Using optimal step-lengths and an initial vector $\mathbf{X}^0 = (3, 3)$, $\mathbf{Y}^0 = (5, 5)$ each search direction was tested for a variety of values of k. The results of this can bee seen in Fig. 2.7.

The first thing we notice is that the number of iterations needed to converge increases as k is increased. So as the network becomes increasingly unbalanced. Secondly that *algorithm 1* and *most obvious* are very similar in their performance until $k > 25$ where *algorithm 1* fails. Thirdly the performance of the *two-direction* search is significantly worse than both of the others in all cases.

Second comparison. As we saw in Section 2.4.3 the *most obvious* search direction does not work well for flat cost and demand functions. So we create a problem with a parameter that can be varied having the effect of changing the function from being steep to flat. We define the problem as below

$$A = \begin{pmatrix} 2 & 1-k \\ 1-k & 2(1-k) \end{pmatrix}, \quad \mathbf{A}_0 = \begin{pmatrix} k \\ 3k \end{pmatrix} \tag{2.14}$$

$$B = \begin{pmatrix} 1 & 0 \\ 0 & 32(1-k) \end{pmatrix}, \quad \mathbf{B}_0 = \begin{pmatrix} 4 \\ 1+96(1-k) \end{pmatrix}$$

so $\mathbf{C}(.)$ has a positive definite definite Jacobian A as long as $0 \leq k < 1$. Also $\mathbf{D}(.)$ has a positive definite Jacobian B. The solution, as before is still

$$\mathbf{X}^{\text{sol}} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \mathbf{Y}^{\text{sol}} = \begin{pmatrix} 3 \\ 3 \end{pmatrix}.$$

Additionally for $0 \leq k < 1$ the cost and demand functions are both separable, so in the range $0 \leq k < 1$ the problem is still very standard mathematically. Using optimal step-lengths and an initial vector $\mathbf{X}^0 = (3, 3)$, $\mathbf{Y}^0 = (5, 5)$ each search direction was tested for a variety of values of k. The results of this can bee seen in Fig. 2.8.

Like some previous tests [**6**] the *two-direction* method is slower than *most obvious* for $k \in [0, 1)$. Additionally *most obvious* fails to converge when $k = 1$ while the *two-direction* method does. Further tests from different starting locations and slightly transformed different problems provide similar results indicating that the *two-direction* method is a more reliable, if often slower, method of finding variable demand equilibrium than the *most obvious* one alone.

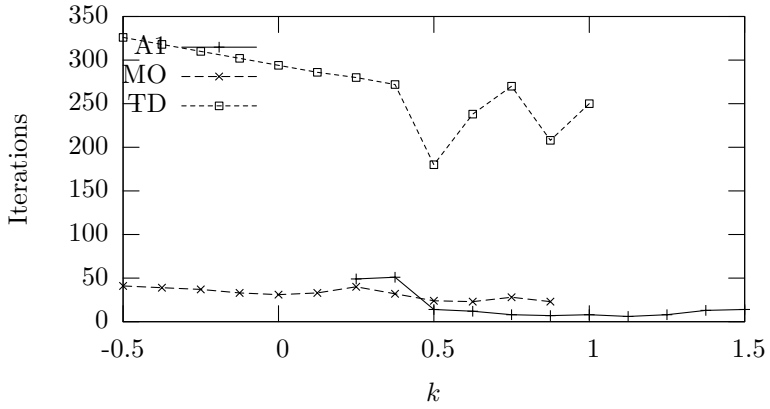


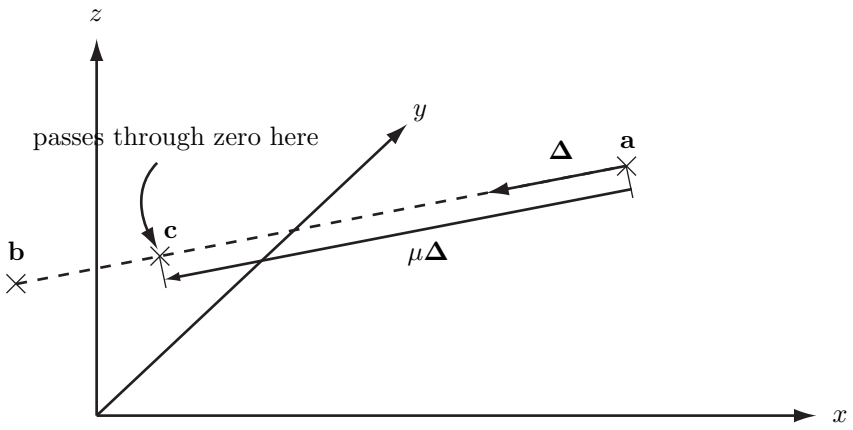
Figure 2.8: Tests with second problem using optimal step-lengths

2.7. How to keep the tuples within the "safe" positive area

During the running of the algorithms described so far, it is sometimes possible for the result of a line search to return a λ that produces a destination vector that lies outside the range of feasible flows. More specifically it may produce negative flow. A great many of our assumptions will break down if this is allowed to happen, so measures must be taken to prevent it.

Because of the nature of our algorithms it is necessary that this procedure be efficient both in terms of memory and computation time. The method we arrived at is as follows.

If we are at a point $\mathbf{a}$ and are asked to move in a search direction $\boldsymbol{\Delta}$ with λ as our step size, then we call the point we end up at $\mathbf{b} = \mathbf{a} + \lambda\boldsymbol{\Delta}$.



(1) Set μ to be very large and $i \leftarrow 1$,
(2) If $\mathbf{b}_i < 0$ then set

$$\mu \leftarrow \min\left(-\frac{\mathbf{a}_i}{\boldsymbol{\Delta}_i}, \mu\right),$$

(3) Increment $i \leftarrow i + 1$,
(4) If $i < N$ then go to (2),
(5) Set $\mathbf{c} \leftarrow \mathbf{a} + \mu\boldsymbol{\Delta}$.

We have now calculated $\mathbf{c}$ as it must be the point which corresponds to the solution of the linear equation corresponding to the smallest value of λ.

CHAPTER 3

Differentiable cost and demand functions

A limitation of the algorithms discussed so far is that we require that they are linear. It is safe to say that most real-world networks will not be so simple. Specifically the part of the algorithm that requires linearity is the step-length selection, so we now examine another method of performing this.

3.1. Armijo-like step-lengths

In the same manner as our original algorithm in Section 2.2, at each point $\mathbf{X}_n$ along its path it chooses a search direction $\boldsymbol{\Delta}^n$. However it bases its choice of λ_n upon the previous one, λ_{n-1}. Some notation quickly

- $\mathbf{X}^n$ — current point,
- λ_n — current step-length,
- $\boldsymbol{\Delta}^n$ — current search direction,
- $\mathbf{X}^{n+1} = \mathbf{X}^n + \lambda_n \boldsymbol{\Delta}^n$ — the next point.

So we are in the same situation as Fig. 2.2. In our Armijo algorithm, though, we already have a λ_n that is set by the last iteration of the algorithm. A rough explanation of the intuitive intention of the algorithm is as follows. Now examine Fig. 3.1, which is a two dimensional slice through Fig. 2.2 along the line of Δ. There are four projected next points in the diagram based upon four different values of λ_n we may be given. The algorithm bases its action upon the result of comparing the directional derivative at our current point, $\mathbf{X}^n$, to the gradient along our projected step which we will call c.

To calculate c we simply take the difference in the values of the objective function at each of the two points $\mathbf{X}^n$ and $\mathbf{X}^{n+1}$ and divide through by λ_n

$$c = \frac{V(\mathbf{X}^{n+1}) - V(\mathbf{X}^n)}{\lambda_n}.$$

Additionally our directional derivative is simply

$$g = \nabla_{\mathbf{X}} V(\mathbf{X}^n) \cdot \boldsymbol{\Delta}^n.$$

Now we know that there are four possible circumstances we could be in with c and g

- $c \geq 0$. If this is the case then we are looking at going uphill, this means we have gone too far, so we don't move anywhere ($\mathbf{X}^{n+1} \leftarrow \mathbf{X}^n$) and instead we set $\lambda_{n+1} \leftarrow \frac{1}{2}\lambda_n$;
- $0 > c \geq \frac{1}{3}g$. If this is the case then we have right past the minimum but have still gone downhill slightly. We should accept the move because it is downhill ($c < 0$) so $\mathbf{X}^{n+1} \leftarrow \mathbf{X}^n + \lambda_n \boldsymbol{\Delta}^n$, however we still went too far so we also set $\lambda_{n+1} \leftarrow \frac{1}{2}\lambda_n$;
- $\frac{1}{3}g > c \geq \frac{2}{3}g$. If this is the case then we have moved quite a way downhill and have likely come close to the minimum. So we accept the move setting $\mathbf{X}^{n+1} \leftarrow \mathbf{X}^n + \lambda_n \boldsymbol{\Delta}^n$ and preserve our value of λ, ($\lambda_{n+1} \leftarrow \lambda_n$);

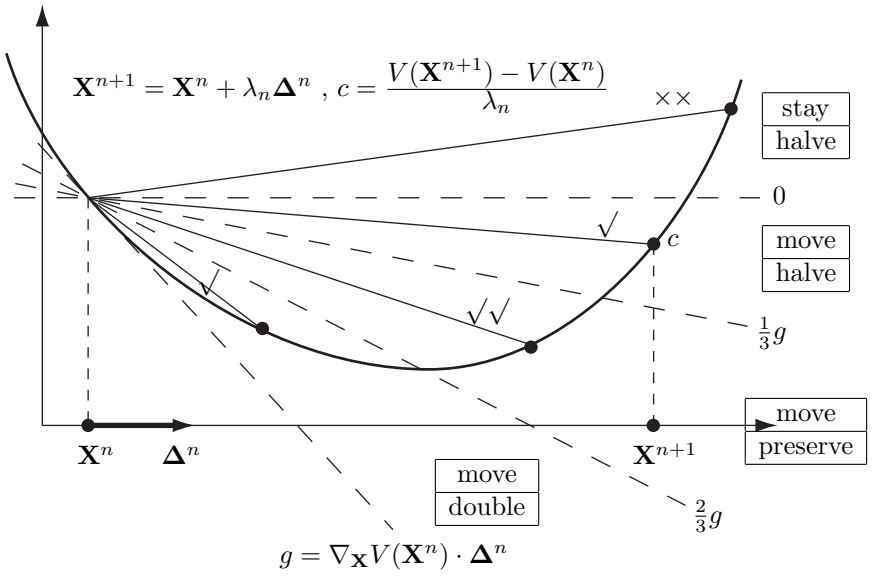


Figure 3.1: Armijo step-length selection. In this case we would move and halve.

- $\frac{2}{3}g > c \geq g$. In this situation we have moved downhill slightly, but because we've gone in a similar direction to the gradient at our start point, it's likely we could have gone a bit further to the more shallow part at the bottom near the minimum. So we set $\lambda \leftarrow 2\lambda$ and accept the move, $\mathbf{X}^{n+1} \leftarrow \mathbf{X}^n + \lambda_n \boldsymbol{\Delta}^n$.

3.1.1. Algorithm outline. The algorithm starts at a given initial point $\mathbf{X}^0$ and moves along a path of consecutive points $\{\mathbf{X}^0, \mathbf{X}^1, \mathbf{X}^2, \ldots\}$. This happens by the following process, assuming we are currently at the point $\mathbf{X}^n$ with λ-value λ_n:

(1) We calculate the search direction
$$\delta \leftarrow \boldsymbol{\Delta}(\mathbf{X}^n) \in \mathbb{R}^N;$$
(2) From $\mathbf{X}^n$ we calculate
$$\mathbf{X}^{\text{temp}} \leftarrow \mathbf{X}^n + \lambda_n \delta$$
(3) We determine if we have moved significantly with a certain tolerance and if we consider the algorithm to have stalled then we stop.
(4) We calculate
$$g \leftarrow \nabla_{\mathbf{X}^n} V(\mathbf{X}^n) \cdot \delta_{\mathbf{X}};$$
and
$$c \leftarrow \frac{V(\mathbf{X}^{\text{temp}}) - V(\mathbf{X}^n)}{\lambda_n};$$
(5) Armijo calculation:
   (a) If $c \geq 0$ then set $\lambda_{n+1} \leftarrow \frac{1}{2}\lambda_n$ and $\mathbf{X}^{n+1} \leftarrow \mathbf{X}^n$, (Stay where we are and halve λ);
   (b) else if $c \geq \frac{1}{3}g$ then set $\lambda_{n+1} \leftarrow \frac{1}{2}\lambda_n$ and $\mathbf{X}^{n+1} \leftarrow \mathbf{X}^{\text{temp}}$, (Move to $\mathbf{X}^{\text{temp}}$ and halve λ);

(c) else if $c \geq \frac{2}{3}g$ then set $\lambda_{n+1} \leftarrow \lambda_n$ and $\mathbf{X}^{n+1} \leftarrow \mathbf{X}^{\text{temp}}$, (Move to $\mathbf{X}^{\text{temp}}$ and preserve λ);
(d) else if $c \geq g$ then set $\lambda_{n+1} \leftarrow 2\lambda_n$ and $\mathbf{X}^{n+1} \leftarrow \mathbf{X}^{\text{temp}}$, (Move to $\mathbf{X}^{\text{temp}}$ and double λ).

(6) Increment n and go back to (1).

3.1.2. Application to algorithm 1. In order to use the Armijo-like step-length selection algorithm described in Section 3.1.1 we need to be able to calculate the values of

$$c(\mathbf{X},\, \lambda) = \frac{V^{\text{A1}}(\mathbf{X} + \lambda \boldsymbol{\Delta}^{\text{A1}}) - V^{\text{A1}}(\mathbf{X})}{\lambda},$$

and

$$g(\mathbf{X}) = \nabla_{\mathbf{X}} V^{\text{A1}}(\mathbf{X}) \cdot \boldsymbol{\Delta}^{\text{A1}}(\mathbf{X}).$$

Clearly the calculation of $c(\mathbf{X},\, \lambda)$ is straightforward by the simple substitution of values into the original definitions of V^{A1} (2.5) and $\boldsymbol{\Delta}^{\text{A1}}$ (2.4). We also explicitly found $\nabla_{\mathbf{X}} V^{\text{A1}}(\mathbf{X})$ in Theorem 2.2, so we can simply take the vector dot-product with $\boldsymbol{\Delta}^{\text{A1}}$ to find $g(\mathbf{X})$.

3.1.3. Application to the 'most obvious' direction. For our algorithm from Section 3.1.1 to work with the *most obvious* direction it must be extended again in the same manner as the previous chapter.

The algorithm starts at a given initial flow-cost tuple $(\mathbf{X}^0,\, \mathbf{Y}^0)$ and moves along a path of consecutive points $\{(\mathbf{X}^0,\, \mathbf{Y}^0),\, (\mathbf{X}^1,\, \mathbf{Y}^1),\, (\mathbf{X}^2,\, \mathbf{Y}^2),\, \ldots\}$. This happens by the following process, assuming we are currently at the tuple $(\mathbf{X}^n,\, \mathbf{Y}^n)$, with λ_n as our λ value:

(1) We calculate the search direction

$$\delta \leftarrow \boldsymbol{\Delta}^{\text{MO}}(\mathbf{X}^n, \mathbf{Y}^n) \in \mathbb{R}^N \times \mathbb{R}^N;$$

(2) From $(\mathbf{X}^n,\, \mathbf{Y}^n)$ we calculate

$$(\mathbf{X}^{\text{temp}},\, \mathbf{Y}^{\text{temp}}) \leftarrow (\mathbf{X}^n,\, \mathbf{Y}^n) + \lambda_n \delta$$

(3) We determine if we have moved significantly with a certain tolerance and if we consider the algorithm to have stalled then we stop.
(4) We calculate

$$g \leftarrow \nabla_{\mathbf{X}^n} V^{\text{MO}}(\mathbf{X}^n,\, \mathbf{Y}^n) \cdot \delta_{\mathbf{X}} + \nabla_{\mathbf{Y}^n} V^{\text{MO}}(\mathbf{X}^n,\, \mathbf{Y}^n) \cdot \delta_{\mathbf{Y}};$$

and

$$c \leftarrow \frac{V^{\text{MO}}(\mathbf{X}^{\text{temp}},\, \mathbf{Y}^{\text{temp}}) - V^{\text{MO}}(\mathbf{X}^n,\, \mathbf{Y}^n)}{\lambda_n};$$

(5) Armijo calculation:
(a) If $c \geq 0$ then set $\lambda_{n+1} \leftarrow \frac{1}{2}\lambda_n$ and $(\mathbf{X}^{n+1},\, \mathbf{Y}^{n+1}) \leftarrow (\mathbf{X}^n,\, \mathbf{Y}^n)$;
(b) else if $c \geq \frac{1}{3}g$ then set $\lambda \leftarrow \frac{1}{2}\lambda_n$ and $(\mathbf{X}^{n+1},\, \mathbf{Y}^{n+1}) \leftarrow (\mathbf{X}^{\text{temp}},\, \mathbf{Y}^{\text{temp}})$;
(c) else if $c \geq \frac{2}{3}g$ then set $\lambda_{n+1} \leftarrow \lambda_n$ and $(\mathbf{X}^{n+1},\, \mathbf{Y}^{n+1}) \leftarrow (\mathbf{X}^{\text{temp}},\, \mathbf{Y}^{\text{temp}})$;
(d) else if $c \geq g$ then set $\lambda_{n+1} \leftarrow 2\lambda_n$ and $(\mathbf{X}^{n+1},\, \mathbf{Y}^{n+1}) \leftarrow (\mathbf{X}^{\text{temp}},\, \mathbf{Y}^{\text{temp}})$.

(6) Increment n and go back to (1).

We know the values of $V^{\text{MO}}(\mathbf{X},\, \mathbf{Y})$, $\boldsymbol{\Delta}^{\text{MO}}(\mathbf{X}, \mathbf{Y})$ and $\nabla_{\mathbf{X},\mathbf{Y}} V^{\text{MO}}(\mathbf{X},\, \mathbf{Y})$ from (2.8), (2.9) and Theorem 2.4 respectively.

3.1.4. Two-direction. For the *two direction* algorithm we once again extend the previous algorithm to incorporate two search directions. We maintain two separate λ values; one λ^{MO} for the first search direction $\mathbf{\Delta}^{\mathrm{MO}}$ and the other λ^{AD} for the second search direction $\mathbf{\Delta}^{\mathrm{AD}}$. The algorithm otherwise is identical except we modify step (1) as follows.

(1) We calculate the search direction

$$\delta \leftarrow \begin{cases} \mathbf{\Delta}^{\mathrm{MO}}(\mathbf{X}^n, \mathbf{Y}^n), & \text{if } n \text{ is even;} \\ \mathbf{\Delta}^{\mathrm{AD}}(\mathbf{X}^n, \mathbf{Y}^n), & \text{if } n \text{ is odd.} \end{cases} \in \mathbb{R}^N \times \mathbb{R}^N.$$

3.2. Initial testing to verify implementation

We now test the 3 different techniques with the armijo-like step-lengths in order to verify the implementation was performed correctly. In each test we compare all three step-length selection techniques so far: a fixed step-length; an optimally calculated step-length, which is only available for linear functions; and our new Armijo-like method.

We use linear functions in each of the tests to allow comparison against the optimal step-length selection technique. The Armijo-like technique, however, does not take advantage of this fact, still only using gradient information about it.

Given that the algorithms used take so little time to run in these instances, for these tests we run each one 1000 times and take the mean in order to calculate the run time without the randomness of background tasks and other factors on the computer.

3.2.1. Tests with problem 1. Here we examine the nine different combinations of step-length technique and search direction available to us by applying them to problem 1 (2.1). The results of this can be seen in Table 3.1.

		Step-length selection method		
		Fixed (0.1)	Optimal	Armijo
Algorithm 1	Time (ms)	1.295	0.168	0.927
	Iterations	72	1	18
	δ	4.147E-11	3.331E-16	1.082E-11
Most obvious	Time (ms)	2.811	1.309	3.673
	Iterations	147	42	60
	δ	2.469E-10	1.787E-11	4.347E-11
Two-direction	Time (ms)	5.068	1.975	6.594
	Iterations	269	70	111
	δ	1.271E-11	8.584E-12	1.892E-11

Table 3.1: Comparison of all algorithms so far with problem 1

For *algorithm 1* we can see the Armijo-like step-length selection technique requires significantly less steps for a fixed λ; this is reassuring at least. It cannot, however, compete with the optimal step-length selection method because in this particular instance the variable demand equilibrium is solved by direct calculation. This causes the algorithm to converge in a single step.

In the case of *most obvious* the Armijo-like method competes favourably with the optimal step-length method. Clearly this is a very positive result as it indicates that we are not losing too much from not being able to do the line-minimisation

perfectly. Again both techniques significantly outperform the fixed step-length algorithm.

The results for each different step-length selection technique with the *two-direction* method are extremely similar to that of the *most obvious* case. The difference being that it takes almost two times as long to converge in each case.

In making a conclusion about the relative performance of the different search directions we cannot really include the results of the optimal step-length technique as this requires linear functions. For problems of this form, though, we can still conclude that using Armijo-like step-lengths *algorithm 1* performs best of all and is actually reliable. Although it still seems likely that the *most obvious* direction (and also *two-direction*) are more reliable in the general case as we will see shortly.

3.2.2. Tests with problem 2. Here we examine the nine different combinations of step-length technique and search direction available to us by applying them to problem 2 (2.2). The results of this can be seen in Table 3.2.

		Step-length selection method		
		Fixed (0.1)	Optimal	Armijo
Algorithm 1	Time (ms)	Didn't converge	Didn't converge	Didn't converge
	Iterations	Didn't converge	Didn't converge	Didn't converge
	δ	5.352E0	2.245E0	2.245E0
Most obvious	Time (ms)	Didn't converge	3.726	19.198
	Iterations	Didn't converge	140	311
	δ	1.038E-1	9.482E-11	1.728E-10
Two-direction	Time (ms)	Didn't converge	23.629	30.18
	Iterations	Didn't converge	912	526
	δ	5.442E1	2.583E-10	6.604E-11

Table 3.2: Comparison of all algorithms so far with problem 2

The first thing we notice is that *algorithm 1* does not converge using any of the step-length selection algorithms. Fortunately this is the result that we expect from this problem; the trajectories are attracted to the line $y = \frac{1}{2}(3 - x)$ on which the search direction $\boldsymbol{\Delta}^{\mathrm{A1}}$ points uphill causing the algorithm to stall.

In addition to *algorithm 1*, both the *most obvious* and *two-direction* algorithms fail when using the fixed step-length of 0.1. In the case of *most obvious* the lack of convergence is simply because the algorithm stops slightly early, before it reaches the solution, even though it has moved very close to it. The *two-direction* algorithm fails because although $\lambda = 0.1$ is a good value for *most obvious* it does not appear to be a good value for the alternate direction.

For the *most obvious* search direction both optimal and Armijo-like step-lengths cause convergence. The Armijo-like step-length selection method takes roughly twice as long to converge as the optimal step-length method. The actual time in ms, however, is even more than this. It takes over four times as long in real-world time. Clearly this means that complexity of a single step of the the Armijo-like method is approximately double that of the optimal method.

The results for the *two-direction* algorithm tell a different story. The optimal step-length method takes almost double that of the Armijo-like one. Still, though, we see that the Armijo-lime method takes longer in terms of real-world time again.

Overall the Armijo-like method has proved to be a good performer here like it was for the last problem.

3.2.3. Tests with problem 3. Here we examine the nine different combinations of step-length technique and search direction available to us by applying them to problem 3 (2.3). The results of this can be seen in Table 3.3.

		Step-length selection method		
		Fixed (0.1)	Optimal	Armijo
Algorithm 1	Time (ms)	1.645	0.589	1.128
	Iterations	96	22	24
	δ	4.683E-11	7.509E-14	1.266E-11
Most obvious	Time (ms)	3.377	1.342	3.409
	Iterations	183	51	58
	δ	1.75E-10	1.523E-11	5.894E-12
Two-direction	Time (ms)	5.21	2.424	5.577
	Iterations	265	94	96
	δ	4.84E-11	1.515E-11	1.566E-11

Table 3.3: Comparison of all algorithms so far with problem 3

All combinations of search direction and step-length selection technique converge successfully in this problem instance. This was expected as the problem was chosen to be easily solvable, with no difficulties such as BA not being positive definite for example.

Once again *algorithm 1* is the fastest to converge with *most obvious* second and *two-direction* last. This reinforces the idea that *algorithm 1*, although it does not converge in all circumstances, will be very rapid to converge.

Also noteworthy is that in this instance the Armijo-like step-length method almost draws equal with the optimal method, taking only a few more steps to converge. This is a very positive sign, as it shows once again that we are not losing much performance from removing the restriction of linear functions.

As before the fixed step-length of $\lambda = 0.1$ causes the slowest convergence of all reassuring us that our efforts are not producing algorithms that are completely pointless.

3.3. Flat function tests

Here we take the same two tests we performed in Section 2.6 to compare the success rate and performance of the three different algorithms so far but now we use Armijo-like step-lengths.

3.3.1. First test. We use the problem with equations as in (2.13). The results of this can bee seen in Fig. 3.3.1.

The first thing we notice is that the number of iterations needed to converge increases as k is increased. So as the network becomes increasingly unbalanced. Secondly that *algorithm 1* and *most obvious* are very similar in their performance until $k > 25$ where *algorithm 1* fails. Thirdly the performance of the *two-direction* search is significantly worse than both of the others in all cases.

3.3.2. Second test. As we noted earlier in Section 2.6 the *most obvious* search direction is not well suited to flat cost and demand functions. To examine this problem we created a set of equations (2.14) with a variable that could be changed to alter their 'flatness'. The results of this can bee seen in Fig. 3.3.2.

The results of this test mimic what we found when using the optimal step-lengths the last time round.

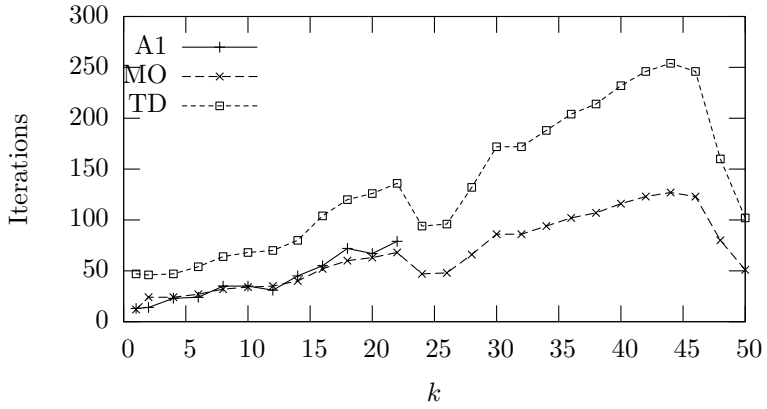


Figure 3.2: Tests with first problem using Armijo-like step-lengths

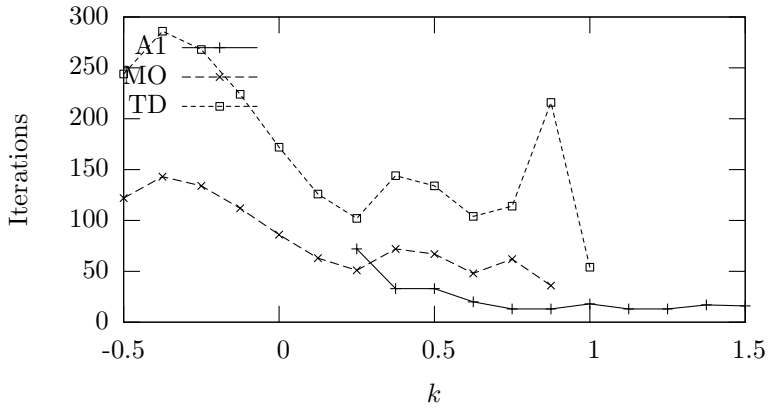


Figure 3.3: Tests with second problem using Armijo-like step-lengths

3.4. A new search direction

Algorithm 1 has a flaw in its search direction that means it does not guarantee being a descent direction for all $V^{\mathrm{A1}}(\mathbf{X})$. The flaw is that it requires more than simply $\mathbf{C}'$ and $-\mathbf{D}'$ to be positive definite. It additionally requires the product, $-\mathbf{D}'\mathbf{C}'$ is strictly positive definite. This is not necessarily the case given only that $\mathbf{C}'$ and $\mathbf{D}'$ are positive and negative definite respectively.

A possible 'cure' inspired by Professor M. Smith is to modify it by pre-multiplying by a matrix P as follows

$$\boldsymbol{\Delta}^{\mathrm{ND}}(\mathbf{X}) = P(\mathbf{X})\boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}). \tag{3.1}$$

When we are in variable demand equilibrium, i.e. $\mathbf{D}(\mathbf{C}(\mathbf{X})) = \mathbf{X}$, and $P(\mathbf{X})$ is non-singular this is equivalent to $P(\mathbf{X})[\mathbf{D}(\mathbf{C}(\mathbf{X})) - \mathbf{X}] = 0$. i.e. given that $P(\mathbf{X})$ is non-singular

$$\mathbf{D}(\mathbf{C}(\mathbf{X})) = \mathbf{X} \Longleftrightarrow \boldsymbol{\Delta}^{\mathrm{ND}}(\mathbf{X}) = 0,$$

so the search direction is zero if and only if we are in variable demand equilibrium. We will use the same objective function, $V^{\mathrm{A1}}(.)$, as *algorithm 1* (2.5).

Our idea is to find a definition of $P(\mathbf{X})$ that guarantees $\mathbf{\Delta}^{\mathrm{ND}}(\mathbf{X})$ is a descent direction for $V^{\mathrm{A1}}(\mathbf{X})$ with only the two conditions $\mathbf{C}'$ and $\mathbf{D}'$ to be positive and negative definite respectively.

We start by calculating $\nabla_{\mathbf{X}} V^{\mathrm{A1}}(\mathbf{X})$:

$$\begin{aligned}
\frac{\partial\left(V^{\mathrm{A1}}\right)}{\partial \mathbf{X}_r} &= \frac{\cancel{1}}{\cancel{2}} \times \cancel{2} \frac{\partial\left(\mathbf{\Delta}^{\mathrm{A1}}\right)}{\partial \mathbf{X}_r} \cdot \mathbf{\Delta}^{\mathrm{A1}}, \\
\Rightarrow \nabla_{\mathbf{X}} V^{\mathrm{A1}} = \frac{\partial\left(V^{\mathrm{A1}}\right)}{\partial \mathbf{X}} &= \frac{\cancel{1}}{\cancel{2}} \times \cancel{2} \times \left(\nabla_{\mathbf{X}} \mathbf{\Delta}^{\mathrm{A1}}\right)^{\top} \mathbf{\Delta}^{\mathrm{A1}}, \\
&= \left[\nabla_{\mathbf{X}} \mathbf{D}(\mathbf{C}(\mathbf{X})) - \nabla_{\mathbf{X}} \mathbf{X}\right]^{\top} \mathbf{\Delta}^{\mathrm{A1}}, \\
&= \left[\{\mathbf{D}'(\mathbf{C}(\mathbf{X}))\mathbf{C}'(\mathbf{X})\}^{\top} - I^{\top}\right] \mathbf{\Delta}^{\mathrm{A1}}, \\
&= \left[\mathbf{C}'(\mathbf{X})^{\top}\mathbf{D}'(\mathbf{C}(\mathbf{X}))^{\top} - I\right] \mathbf{\Delta}^{\mathrm{A1}}.
\end{aligned}$$

Now we find the directional derivative of $V^{\mathrm{A1}}(\mathbf{X})$ in the direction $\mathbf{\Delta}^{\mathrm{ND}}$:

$$\begin{aligned}
\nabla_{\mathbf{X}} V^{\mathrm{A1}} \cdot \mathbf{\Delta}^{\mathrm{ND}} &= \left[\left(\mathbf{C}'(\mathbf{X})^{\top}\mathbf{D}'(\mathbf{C}(\mathbf{X}))^{\top} - I\right) \mathbf{\Delta}^{\mathrm{A1}}\right] \cdot \mathbf{\Delta}^{\mathrm{ND}}, \\
&= \left(\mathbf{\Delta}^{\mathrm{ND}}\right)^{\top} \left[\mathbf{C}'(\mathbf{X})^{\top}\mathbf{D}'(\mathbf{C}(\mathbf{X}))^{\top} - I\right] \mathbf{\Delta}^{\mathrm{A1}}, \\
&= \left(P(\mathbf{X})\mathbf{\Delta}^{\mathrm{A1}}\right)^{\top} \left[\mathbf{C}'(\mathbf{X})^{\top}\mathbf{D}'(\mathbf{C}(\mathbf{X}))^{\top} - I\right] \mathbf{\Delta}^{\mathrm{A1}}, \\
&= \left(P(\mathbf{X})\mathbf{\Delta}^{\mathrm{A1}}\right)^{\top} \mathbf{C}'(\mathbf{X})^{\top} \left(\mathbf{D}'(\mathbf{C}(\mathbf{X}))^{\top}\mathbf{\Delta}^{\mathrm{A1}}\right) - \left(P(\mathbf{X})\mathbf{\Delta}^{\mathrm{A1}}\right)^{\top} I\mathbf{\Delta}^{\mathrm{A1}}, \\
&= \left(P(\mathbf{X})\mathbf{\Delta}^{\mathrm{A1}}\right)^{\top} \mathbf{C}'(\mathbf{X})^{\top} \left(\mathbf{D}'(\mathbf{C}(\mathbf{X}))^{\top}\mathbf{\Delta}^{\mathrm{A1}}\right) - \left(\mathbf{\Delta}^{\mathrm{A1}}\right)^{\top} P(\mathbf{X})^{\top} \left(\mathbf{\Delta}^{\mathrm{A1}}\right). \qquad (3.2)
\end{aligned}$$

Several of the obvious possibilities for our choice of $P(\mathbf{X})$ were examined:

- $P(\mathbf{X}) \equiv \mathbf{C}'(\mathbf{X})$;
- $P(\mathbf{X}) \equiv -\mathbf{D}'(\mathbf{C}(\mathbf{X}))$;
- $P(\mathbf{X}) \equiv -\mathbf{D}'(\mathbf{C}(\mathbf{X}))^{\top}$.

Trying the first option. Setting $P(\mathbf{X}) \equiv \mathbf{C}'(\mathbf{X})$ at first looks good because it causes the right hand side of (3.2) to become negative. Using this our directional derivative is now equal

$$\begin{aligned}
&\left(\mathbf{C}'(\mathbf{X})\mathbf{\Delta}^{\mathrm{A1}}\right)^{\top} \mathbf{C}'(\mathbf{X})^{\top} \left(\mathbf{D}'(\mathbf{C}(\mathbf{X}))^{\top}\mathbf{\Delta}^{\mathrm{A1}}\right) - \left(\mathbf{\Delta}^{\mathrm{A1}}\right)^{\top} \mathbf{C}'(\mathbf{X})^{\top} \left(\mathbf{\Delta}^{\mathrm{A1}}\right), \\
= \underbrace{\left(\mathbf{\Delta}^{\mathrm{A1}}\right)^{\top} \left[\mathbf{C}'(\mathbf{X})^{\top}\mathbf{C}'(\mathbf{X})^{\top}\mathbf{D}'(\mathbf{C}(\mathbf{X}))^{\top}\right] \left(\mathbf{\Delta}^{\mathrm{A1}}\right)}_{\text{unsure of sign}} &\quad \underbrace{- \left(\mathbf{\Delta}^{\mathrm{A1}}\right)^{\top} \mathbf{C}'(\mathbf{X}) \left(\mathbf{\Delta}^{\mathrm{A1}}\right)}_{< 0 \text{ as } \mathbf{C}'(\mathbf{X}) \text{ strictly positive definite}}.
\end{aligned}$$

so this definition of $P(\mathbf{X})$ is not a guaranteed descent direction for $V^{\mathrm{A1}}(\mathbf{X})$.

Trying the second option. Setting $P(\mathbf{X}) \equiv -\mathbf{D}'(\mathbf{C}(\mathbf{X}))$ seems a plausible definition of $P(\mathbf{X})$ because in the right hand side of (3.2) we can see that it again causes negativity. It also looks like it may help the left hand side form into a product that may be negative due to strict negative definiteness. Using this definition our directional derivative is now equal

$$\begin{aligned}
&- \left(\mathbf{D}'(\mathbf{C}(\mathbf{X}))\mathbf{\Delta}^{\mathrm{A1}}\right)^{\top} \mathbf{C}'(\mathbf{X})^{\top} \left(\mathbf{D}'(\mathbf{C}(\mathbf{X}))^{\top}\mathbf{\Delta}^{\mathrm{A1}}\right) + \left(\mathbf{\Delta}^{\mathrm{A1}}\right)^{\top} \mathbf{D}'(\mathbf{C}(\mathbf{X}))^{\top} \left(\mathbf{\Delta}^{\mathrm{A1}}\right), \\
= \underbrace{- \left(\mathbf{D}'(\mathbf{C}(\mathbf{X}))\mathbf{\Delta}^{\mathrm{A1}}\right)^{\top} \mathbf{C}'(\mathbf{X})^{\top} \left(\mathbf{D}'(\mathbf{C}(\mathbf{X}))^{\top}\mathbf{\Delta}^{\mathrm{A1}}\right)}_{\text{unsure of sign because } \mathbf{D}'(\mathbf{C}(\mathbf{X})) \text{ doesn't match } \mathbf{D}'(\mathbf{C}(\mathbf{X}))^{\top}} &+ \underbrace{\left(\mathbf{\Delta}^{\mathrm{A1}}\right)^{\top} \mathbf{D}'(\mathbf{C}(\mathbf{X})) \left(\mathbf{\Delta}^{\mathrm{A1}}\right)}_{< 0 \text{ as strictly negative definite}}.
\end{aligned}$$

so this definition of $P(\mathbf{X})$ is not a guaranteed descent direction for $V^{\mathrm{A1}}(\mathbf{X})$.

Trying the third option. The definition $P(\mathbf{X}) \equiv -\mathbf{D}'(\mathbf{C}(\mathbf{X}))^\top$ was reached because of the results of the last comparison. It was noted that the only reason (3.2) had not become negative was because on the left hand side the $\mathbf{D}'(\mathbf{C}(\mathbf{X}))$ did not match up with the $\mathbf{D}'(\mathbf{C}(\mathbf{X}))^\top$. Because it would not affect the sign of the right hand side it was decided that the transpose be taken so that the parts of the left hand side would match up to form something negative. So now using this definition of $P(\mathbf{X})$ our directional derivative is now equal

$$\begin{aligned} &- \left(\mathbf{D}'(\mathbf{C}(\mathbf{X}))^\top \boldsymbol{\Delta}^{\mathrm{A1}}\right)^\top \mathbf{C}'(\mathbf{X})^\top \left(\mathbf{D}'(\mathbf{C}(\mathbf{X}))^\top \boldsymbol{\Delta}^{\mathrm{A1}}\right) + \left(\boldsymbol{\Delta}^{\mathrm{A1}}\right)^\top \mathbf{D}'(\mathbf{C}(\mathbf{X})) \left(\boldsymbol{\Delta}^{\mathrm{A1}}\right), \\ &= \underbrace{- \left(\mathbf{D}'(\mathbf{C}(\mathbf{X}))^\top \boldsymbol{\Delta}^{\mathrm{A1}}\right)^\top \mathbf{C}'(\mathbf{X})^\top \left(\mathbf{D}'(\mathbf{C}(\mathbf{X}))^\top \boldsymbol{\Delta}^{\mathrm{A1}}\right)}_{<0 \text{ as } \mathbf{C}'(\mathbf{X}) \text{ strictly positive definite}} + \underbrace{\left(\boldsymbol{\Delta}^{\mathrm{A1}}\right)^\top \mathbf{D}'(\mathbf{C}(\mathbf{X})) \left(\boldsymbol{\Delta}^{\mathrm{A1}}\right)}_{<0 \text{ as strictly negative definite}}. \end{aligned}$$

so is appears that this definition of $P(\mathbf{X})$ is a guaranteed descent direction for $V^{\mathrm{A1}}(\mathbf{X})$ for all cost and demand functions strictly positive and negative definite respectively.

So to recap our definition of the new search direction working in terms of flows only is

$$\boldsymbol{\Delta}^{\mathrm{ND}}(\mathbf{X}) = -\mathbf{D}'(\mathbf{C}(\mathbf{X}))^\top \boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}). \tag{3.3}$$

Hopefully this new direction that fixes the critical flaw that arose in *algorithm 1* will also prove to be an expedient route to the solution of variable demand equilibrium problems. We shall perform some experimental tests to determine how good our 'cure' has been.

3.5. Experimental results with new search direction

Implementation of the new search direction was extremely simple given that *algorithm 1* was already up and running. Because of the way the computer system was implemented only the input of the search direction required changing.

Using the new implementation experimental tests were carried out to compare the new direction's performance to algorithm 1 upon the three original test problems. All the tests were performed using the Armijo-like step-lengths in order to ensure they were as unbiased as possible.

In each of the following graphs, *algorithm 1* is plotted with a dashed line while the path of the new search direction is plotted using a solid line. The solution vector is also marked with a circle.

3.5.1. Problem 1. In this, the first problem, it happens that the new direction is always identical to the search direction of *algorithm 1.* The reasons for this are as follows; in problem 1 we defined

$$\mathbf{D}(\mathbf{Y}) = \begin{pmatrix} 2 \\ 3 \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \mathbf{Y},$$

so that

$$\mathbf{D}'(\mathbf{Y}) = - \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

This results in our search direction being

$$\begin{aligned} \boldsymbol{\Delta}^{\mathrm{ND}}(\mathbf{X}) &= -\mathbf{D}'(\mathbf{C}(\mathbf{X}))^\top \boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}), \\ &= - \left[- \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right]^\top \boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}), \\ &= I \boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}), \\ &= \boldsymbol{\Delta}^{\mathrm{A1}}(\mathbf{X}). \end{aligned}$$

So in this problem the performance of the two algorithms is identical. As we can see in Fig. 3.4 each algorithm was started at the point $\mathbf{X}^0 = (5,\, 5)^\top$ and clearly they follow the exact same path to the solution at $\mathbf{X}^{\text{sol}} = (\frac{2}{3},\, 1)^\top$.

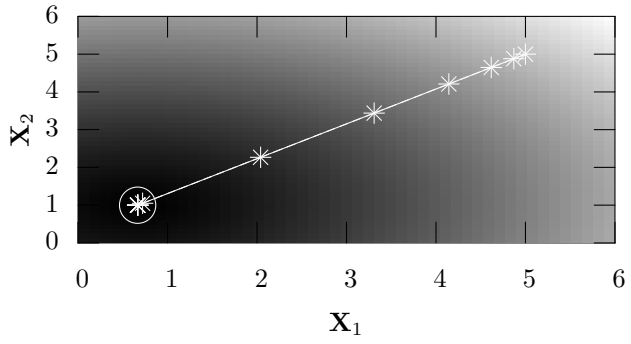


Figure 3.4: Comparison of algorithm 1 and new direction for solving problem 1

3.5.2. Problem 2. The results of *algorithm 1* and the new search direction attacking problem 2 are shown in Fig. 3.5. The formatting of the graph has been changed slightly because although the new search direction does find the solution, it takes a very large (> 1000) steps to find it. If each point were marked with a cross as in the previous graphs then the trajectory the algorithm follows would be unclear because the crosses would overlap causing a thick, imprecise line to be drawn. There is also an additional square mark on the graph this time indicating precisely where *algorithm 1* started to point uphill.

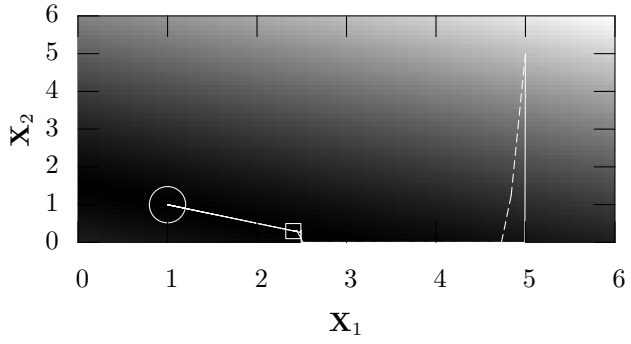


Figure 3.5: Comparison of algorithm 1 and new direction for solving problem 2

We can see that the flaw that existed in *algorithm 1* has caused it to stall at roughly the point $(2.4,\, 0.3)^\top$, which is on the line $y = \frac{1}{2}(3 - x)$, where $\boldsymbol{\Delta}^{\text{A1}}$ points uphill. Clearly the flaw that existed in *algorithm 1* has been somewhat 'ironed out' with the new search direction. It reaches the solution vector $\mathbf{X}^{\text{sol}} = (1,\, 1)^\top$

although it does take a very large number of iterations to do so. Upon closer inspection (see Fig. 3.6) we can see that the reason for this appears to be because it is zigzagging up and down repeatedly past the bottom of the valley of the objective function. It seems this is the price we pay for the guarantee of descent.

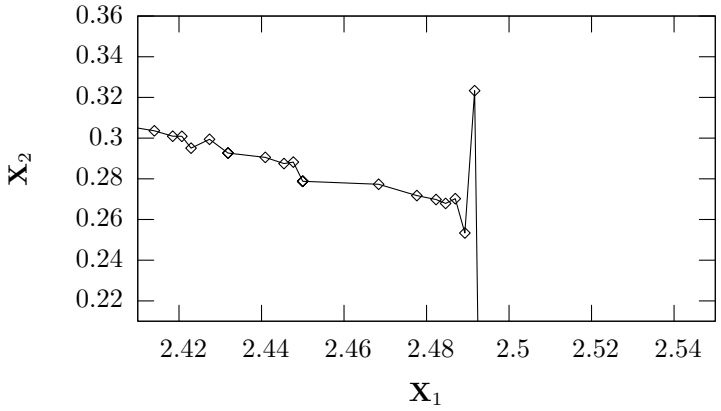


Figure 3.6: Zoomed in section of new search direction attacking problem 2

3.5.3. Problem 3. When attacking problem 3 the performance of the new search direction is again quite comparable to the performance of *algorithm 1.* We can see in Fig. 3.7 that each algorithm fairly quickly converges to the region the solution vector, $\mathbf{X}^{\text{sol}} = (1,\, 1)^\top$, lies in.

What we do see is that the new search direction moves toward the solution in a considerably less direct manner. It also takes several more iterations of the algorithm for it to get there. This loss in performance however, seems reasonably insignificant compared to the gain of a guaranteed descent direction.

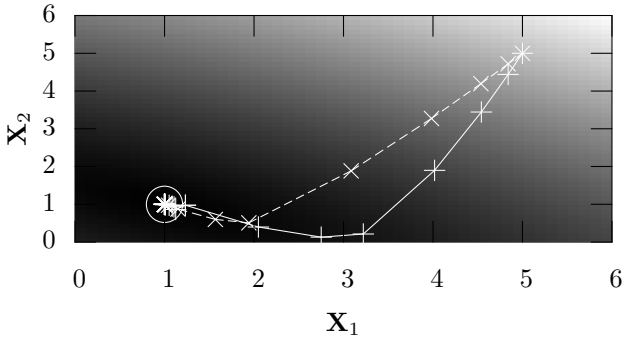


Figure 3.7: Comparison of algorithm 1 and new direction for solving problem 3

3.5.4. Comparison of directions over a small area. In this section we perform a comparison of the performance of *algorithm 1* against that of the new search direction over a region of the problem space close to the solution vector, $\mathbf{X}^{\text{sol}} = (1,\, 1)^\top$. For this test we selected problem 2 so that we could demonstrate the failure of *algorithm 1* for starting vectors even very close to the solution.

20 starting locations around the solution vector were chosen in a circular pattern. From each of these locations the algorithm was started once using the search direction from *algorithm 1* and once using the new search direction. The trajectories each run followed were recorded and plotted. In Fig. 3.8 the paths taken by the search direction of *algorithm 1* were plotted while in Fig. 3.9 the trajectories are all from the new search direction. For all runs the Armijo-like step-length selection technique was used so ensure a more fair test of the two directions.

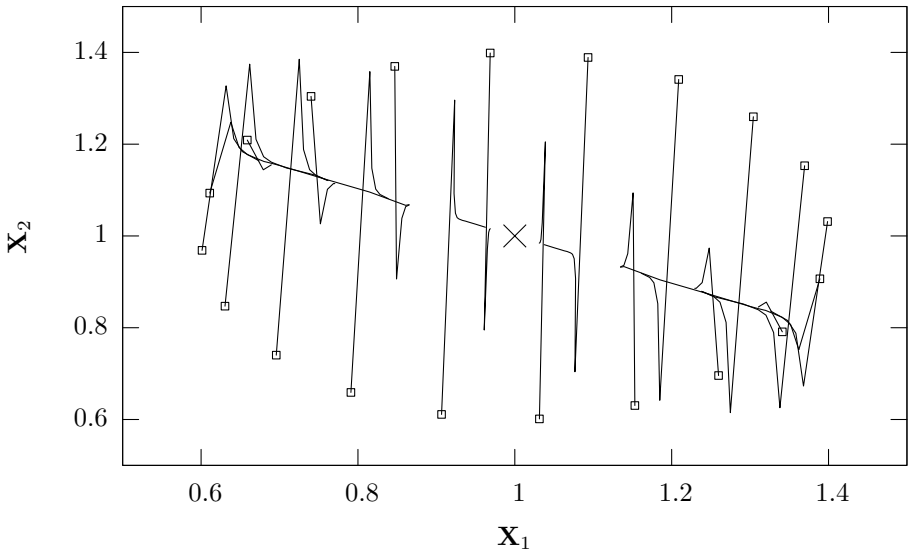


Figure 3.8: Several trajectories followed by algorithm 1 on problem 2

In the first figure we can clearly see *algorithm 1* stalling as soon as it reaches the line $(t,\, \frac{1}{2}(3-t))^\top$, demonstrating the flaw we noticed much earlier on. Even when the start vector is almost directly above the solution the algorithm will stall before it reaches it. It is the author's opinion, however, that this is not only the fault of the search direction, but also in the objective function V^{A1}. We can see very clearly in Fig. 3.10 this valley shape. One of the requirements of a good objective function for use in minimisation is for it to be basin-shaped. This helps to avoids the situation of a minimisation algorithm stalling at the bottom of the valley because it appears to be the minimum when it it actually far away.

In the second figure we see our 'cure' introduced into the new search direction works with the algorithm starting at all of the locations shown. Something strange, however, is the path that the new direction takes to the solution. Instead of curving naturally toward the solution it moves in a very straight line back and forward until it settles in the base of the valley along $(t,\, \frac{1}{2}(3-t))^\top$. It then turns a very sharp angle and zigzags its way toward the solution, managing to decrease by alternating along the lowest edges of the valley as it proceeds as we saw in Fig. 3.6.

What we still need to bear in mind is that however successful the new direction appears to be on that graph, it does take a great many iterations to converge. Due

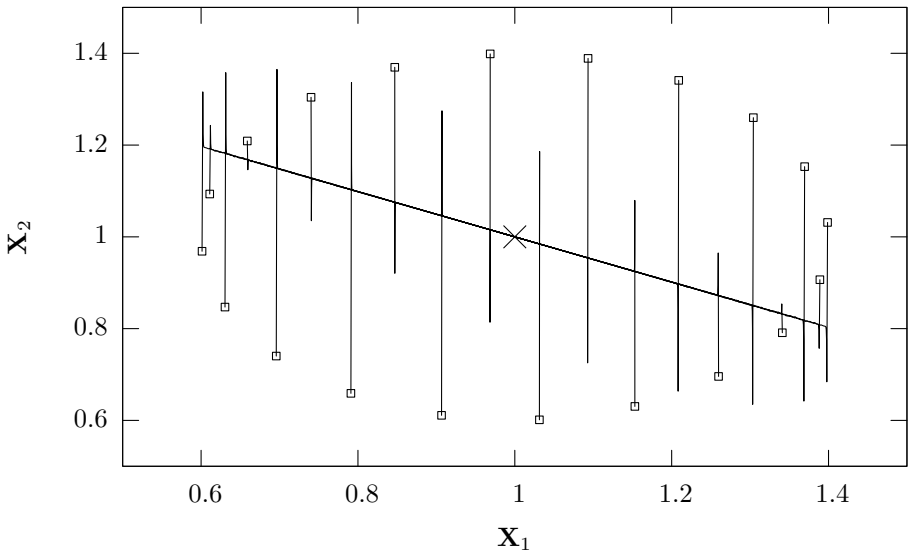


Figure 3.9: Several trajectories followed by the new direction on problem 2

to time constraints a new objective function to remove this valley problem could not be developed to determine if the *algorithm 1* search direction is truly flawed. It is quite possible that with a better objective function the original search direction could actually be superior to the new direction.

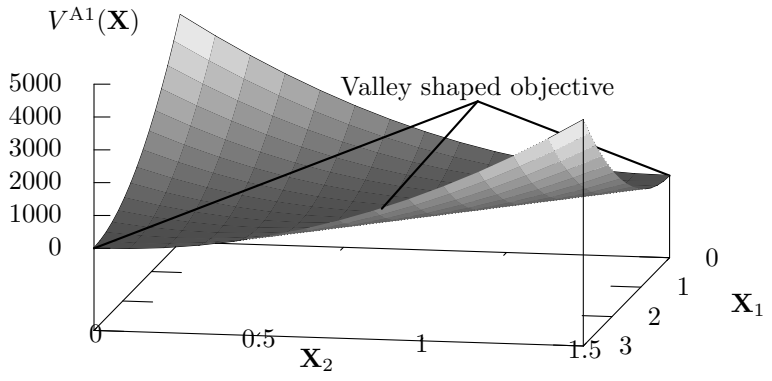


Figure 3.10: Demonstration of the valley shape V^{A1} forms for problem 2

3.6. More experimental results with new search direction

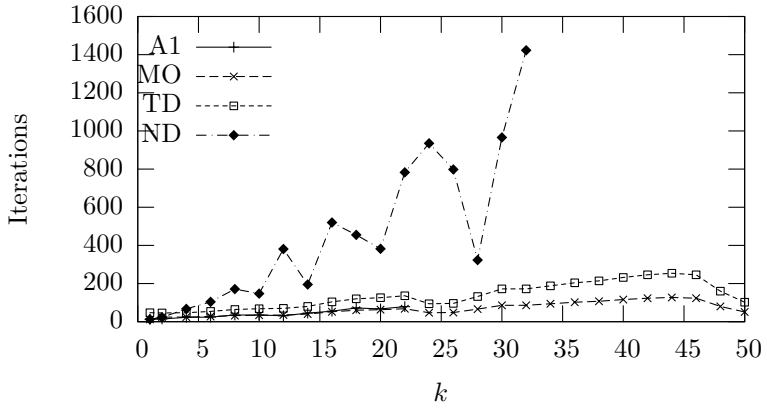


Figure 3.11: Tests with first problem using Armijo-like step-lengths now including the new search direction

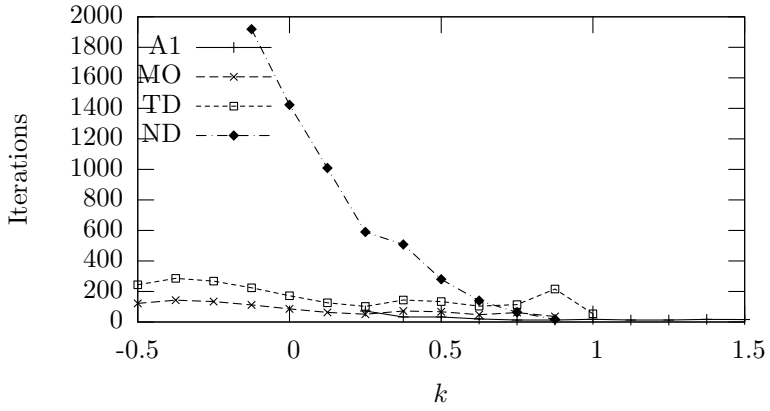


Figure 3.12: Tests with second problem using Armijo-like step-lengths now including the new search direction

CHAPTER 4

Conventional search techniques

4.1. Downhill simplex method

So far all the algorithms we have used have explicitly made use of one one-dimensional minimisation along some sort of search direction as a part of their computational strategy. In contrast to this we will now combine this with an algorithm in which one-dimensional minimisation does not figure at all.

The downhill simplex method was created by Nelder and Mead[**4**]. Its main advantage over the previous methods we have been studying is that it only requires function evaluations; it does not need derivative information of any kind. Unfortunately the simplex method is not very efficient in terms of the number of function evaluations it makes. Another more recent method, Powell's method, is widely accepted as being significantly faster. It does have the disadvantage, however, of being a *lot* more complicated to implement. So the simplex method is often used, as here, where the function evaluations are not too computationally intensive and speed of implementation is important.

The simplex is a natural geometric form that is the simplest 'solid' in the dimension we are working in. In N dimensions it is a geometrical figure consisting of $N + 1$ vertices and all interconnecting arcs and faces. In two dimensions, our simplex is a triangle. In three dimensions we have a tetrahedron. Quite importantly the simplex is not necessarily regular; it can be distorted in any fashion.

For the purposes of minimisation we are only interested in non-degenerate simplexes. By this we mean that the simplex has a finite N-dimensional volume.

When we are minimising in one dimension, it is possible to encapsulate a minimum such that we can guarantee finding it. Unfortunately we cannot do the same in multi-dimensions. We must continue with our loose idea of an algorithm we came up with in the first chapter. We have a starting guess at the solution, an N-vector in our search space, as our first point to examine. Our algorithm then attempts to make its way downhill through multidimensional space until it believes it has found a minimum. To guarantee finding the global minimum is very hard and the simplex is not particularly suited to this purpose. However the condition of monotonicity we impose upon our cost and demand functions mean that there will only be a single global minimum.

This single global minimum would mean that ideally we could apply the simplex method to directly solving the variable demand equilibrium problem. However the problem arises because of the polynomial nature of the algorithm's memory requirements. In the computer's memory we must maintain a representation of the simplex. This requires $N(N + 1)$ numbers to be represented. If we were to have, say, a one million dimensional space this would require

$$1,000,000 \times 1,000,0001 = 1,000,001,000,000 \approx 1 \times 10^{12}$$

floating point numbers to be stored in memory. Using double accuracy floating point storage on a standard desktop PC with an Intel instruction-set this would

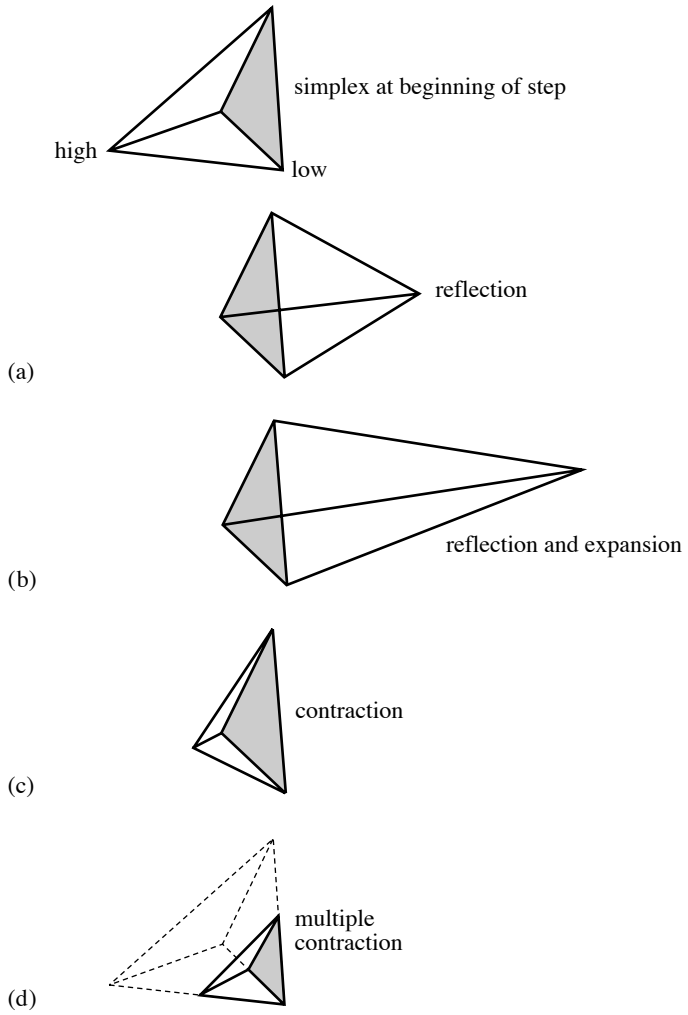


Figure 4.1: Possible outcomes for a simplex

require

$$\frac{48 \times 1,000,000 \times 1,000,0001}{8 \times 1024^3} \approx 5,588 \text{ GiB}$$

of main-store memory. In the current day these sorts of memory requirements are simply absurd.

What we have decided to do in order to avoid this problem in future is to have the simplex operate in only two dimensions. We will take the two search directions $\boldsymbol{\Delta}^{\mathrm{MO}}$ and $\boldsymbol{\Delta}^{\mathrm{AD}}$ and run a two-dimensional simplex method from the start vector with each component representing a linear combination of one of the search vectors. We know that at any vector $(\mathbf{X}, \mathbf{Y})^\top$ one of $\boldsymbol{\Delta}^{\mathrm{MO}}(\mathbf{X}, \mathbf{Y})$ and $\boldsymbol{\Delta}^{\mathrm{AD}}(\mathbf{X}, \mathbf{Y})$ is guaranteed to be a descent direction. So to have the simplex work in terms of each of them will likely point it toward the solution vector. When the simplex converges we will start the simplex again with a new evaluation of $\boldsymbol{\Delta}^{\mathrm{MO}}(\mathbf{X}, \mathbf{Y})$ and $\boldsymbol{\Delta}^{\mathrm{AD}}(\mathbf{X}, \mathbf{Y})$. This process will continue until the simplex fails to descend further than the point we start it at.

To continue our description of how the simplex works, however, we will return to multiple dimensions. To initialise the simplex we do not simply start it with a single vector; it requires $N + 1$ vectors, together defining the initial simplex. One of these points is typically the starting vector, which we will call $\mathbf{P}^0$ here. We will then take the other N points to be

$$\mathbf{P}^i = \mathbf{P}^0 + \lambda \mathbf{e}^i,\ i \in \{1, \ldots, N\} \tag{4.1}$$

where $\mathbf{e}^i$ is a unit N-vector. λ is a constant we use in order to guess the scale of the problem we are attempting to solve.

The downhill simplex method now takes a series of steps. The most common being to flip over the simplex by moving the vector where the result of the evaluation function is largest over to the opposite face of the simplex hopefully to a lower valued (better) point. This kinda of step is called a reflection and in implementation are constructed to conserve the volume of the simplex. When it is possible, the algorithm will also expand in one or more directions to allow it to take larger steps. If the simplex reaches a valley, it contracts itself in the transverse direction in order to squeeze itself to the bottom and flow down the valley floor. If the simplex finds itself in a tight basin it contracts in all directions with the centre of contraction about the lowest-valued vector. For this reason implementations typically call the algorithm 'amoeba' describing the nature of its movement. The moves we have discussed here are shown in Fig. 4.1. Due to time constraints this figure was copied from the excellent book Numerical Recipes in C[**5**].

What we need to determine now are the termination criteria. In multidimensional minimisation this is always a delicate issue. Because we have multiple independent variables we cannot simply specify a tolerance as we could for a single variable. At the end of each iteration of the multidimensional algorithm we calculate the magnitude of the vector distance we have travelled and if it is smaller than some tolerance then we stop.

Remark 4.1. It should be noted that the above termination criteria can easily be fooled by a single anomalous step that failed to move anywhere. Because of this it is often a wise precaution to restart the downhill simplex method at the point it has finished at. N of the $N + 1$ input vectors should be re-initialised by (4.1) using the newly discovered 'minimum' as $\mathbf{P}^0$.

Restarting the algorithm like this should never be too computationally intensive. If the minimum discovered actually was a minimum then the algorithm will very quickly re-converge there.

4.2. Experimental results with the two-direction simplex method

We now take our downhill simplex method implementation and apply it to our standard three test problems. Each problem attempt is started at the initial vector $(\mathbf{X}, \mathbf{Y}) = ((5, 5)^\top, (5, 5)^\top)$. At each step of the algorithm our simplex is initialised to $\{(0, 0)^\top, (0.01, 0)^\top, (0, 0.01)^\top\}$.

Problem 1. The trajectory the algorithm follows when attacking problem 1 is shown in Fig. 4.2a. Clearly this method of minimisation is well suited to our variable demand equilibrium problems. In a single run of the simplex method we come within 0.1 of the solution vector. It then takes a further two runs of the simplex minimisation to be precisely at the solution.

The downside, though, is that the algorithm requires 910 evaluations of the objective function overall. This is significantly higher than required for the previous style algorithms.

In Fig. 4.3a the actual simplex itself is drawn in 3 of the four dimensions for this problem. We can see how it flips its way over to the solution while also expanding to

increase its speed and then contracts around it as described in the previous section. The individual steps the simplex runs make up and marked by the dashed lines.

Problem 2. In Fig. 4.2b we see the path that the algorithm follows when applied to problem 2. In the first step we see that it attempts to leave the positive region of the search space and is clipped by the protection procedure. This behaviour is very similar to what we have seen the other algorithms do and it is quite intriguing to see it in a radically different search method.

The zigzagging following this causes the algorithm to utilise 2389 objective function evaluations. This performance is rather poor. It must be remembered, though, that this minimisation method is not making use of any gradient information at all.

As before, in Fig. 4.3b we have a three dimensional slice of the whole simplex. It becomes clearer now how the search is zigzagging in more than just the flow-vector space. We are also getting great fluctuations in cost-vector space.

Problem 3. Problem 3, like problem 1, is ideally suited to the downhill simplex minimisation method as we can see in Fig. 4.2c. The first simplex run takes us very close to the solution again and as with the first problem only three more simplex runs takes us almost exactly onto the solution.

Unfortunately the algorithm still takes a relatively large number of objective function evaluations with a total of 1532. It is possible this could be cut down by starting the simplex at a different size.

In Fig. 4.3c we see the whole simplex figure for problem 3. We can see the large (and therefore rapid) simplex move across to the left where the solution lies where it then contracts.

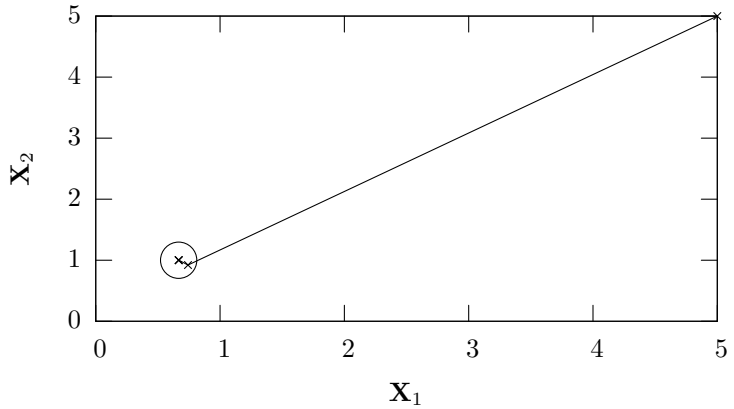
4.3. More experimental results with the two-direction/simplex method

In this section we will re-run the tests we did for the two problems with variable parameters now for the two-direction/simplex method. As the new algorithm differs significantly from the previous ones we will also include real-world time in our comparisons. We will perform a comparison against the *two-direction* method.

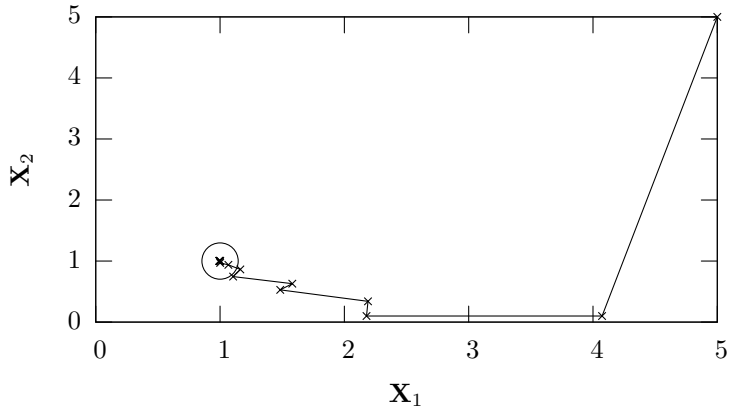
First problem. The problem is defined in (2.13). As before we test the algorithm for values of k through the range 1 to 50. For this problem both the new algorithm and the *two-direction* method successfully find the solution for all values of k. In terms of function evaluations, see Fig. 4.4a, the new algorithm takes approximately 6 times as many. The *two-direction* method takes around 100 to 200 iterations while the new method idles along with around 600 to 900 evaluations. Clearly this measure of performance makes the new algorithm seem somewhat undesirable.

When we look at the real times taken in milliseconds in Fig. 4.4b, however, the new two-direction/simplex method picks up a little. Although it is still slower than the *two-direction* method the difference is not nearly as great. On average we see the new algorithm taking 18 milliseconds to converge, which is not too far behind the 9 milliseconds for the *two-direction* method. This indicates that the new algorithm has lower computational overheads than the *two-direction* method. This could partly be caused by differing quality in the implementations of the algorithms.

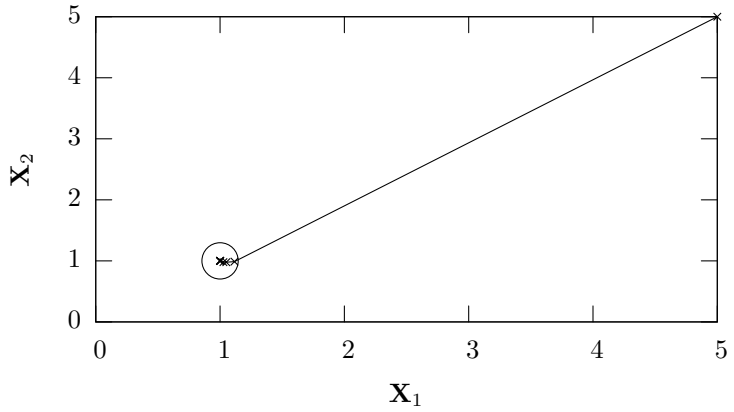
Second one. This equations for this problem are defined in (2.14). We use the same range of k values as in previous tests, i.e. $k \in [-0.5, 1.5]$. As we can see in Fig. 4.5a this problem both the new two-direction/simplex method and the original *two-direction* method fail to find the solution for $k > 1$. Clearly the simplex



(a) Problem 1

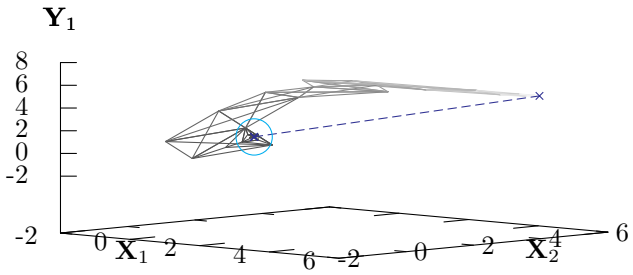


(b) Problem 2

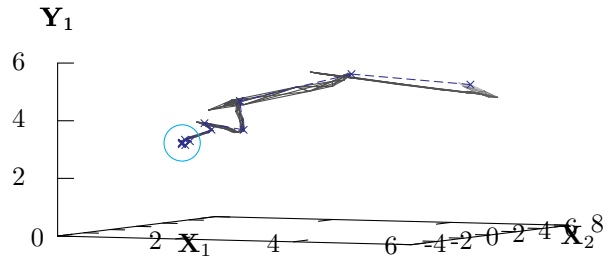


(c) Problem 3

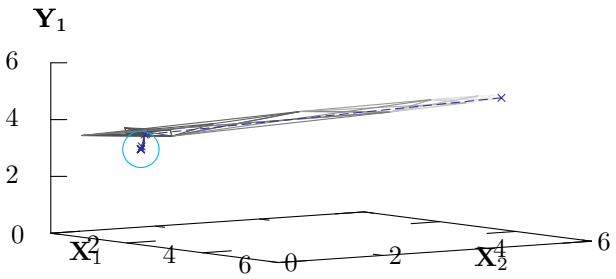
Figure 4.2: Two-direction simplex search method on different problems



(a) Problem 1

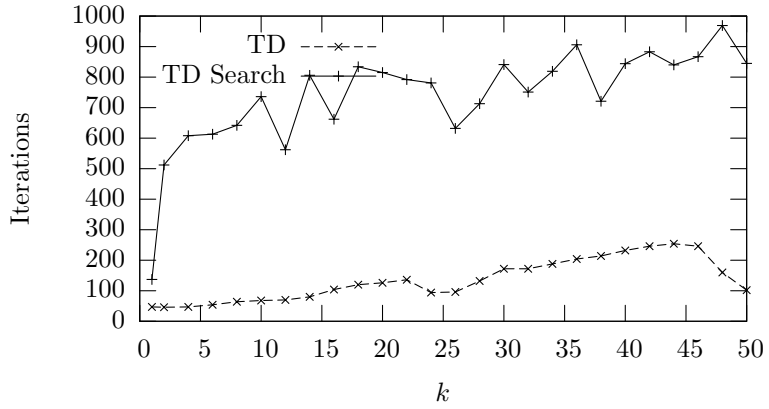


(b) Problem 2

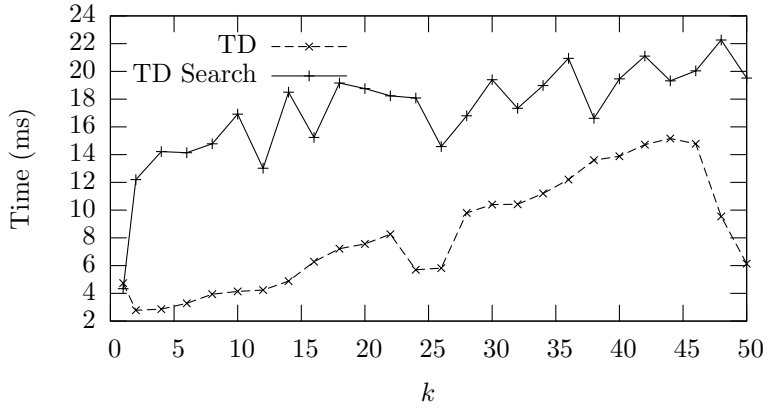


(c) Problem 3

Figure 4.3: Two-direction simplex search method on different problems showing simplex



(a) Iterations



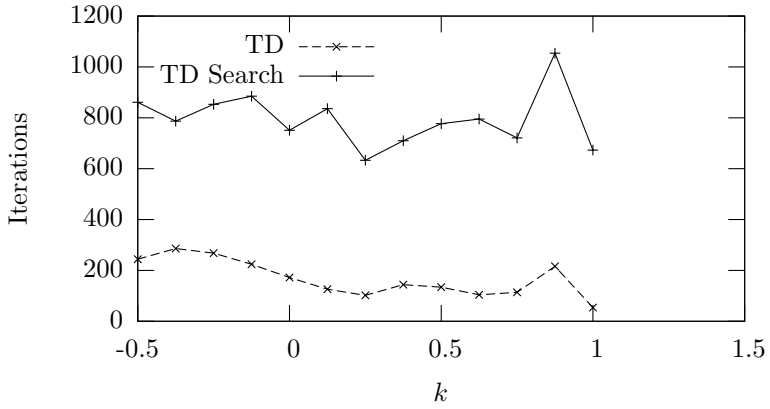
(b) Times in milliseconds

Figure 4.4: Two-direction/simplex method on first problem

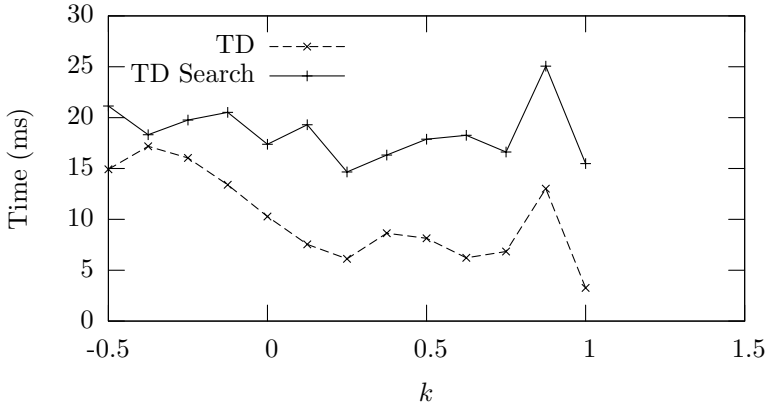
search method is 'thrown-off' by the failure of either $\mathbf{\Delta}^{\mathrm{MO}}$ and $\mathbf{\Delta}^{\mathrm{AD}}$ to be descent directions.

For the iteration counts we see the same story as told by the previous problem. The two-direction/simplex method takes in the region of 900 function evaluations to reach the solution vector, whilst the *two-direction* method is only in the region of 200. It is interesting to see that the spike at $k = 0.9$ is present on both the two-direction/simplex method and the *two-direction* method. Again this indicates that the search directions are having a very direct impact upon the performance of the simplex minimisation procedure.

As in the previous problem, Fig. 4.5b shows the real-world times for the two methods are much closer than their iteration counts suggest. In one case ($k = -0.4$) the performance of the two methods is almost identical at around 17 milliseconds. This is quite promising as it shows that it could be possible to construct an algorithm that performs almost as well as the Armijo-like algorithms but doesn't require gradient information.



(a) Iterations



(b) Times in milliseconds

Figure 4.5: Two-direction/simplex method on second problem

4.4. Simulated Annealing

The simulated annealing[**2, 1**] method of multidimensional minimisation is a technique that has proved itself across a great many engineering disciplines. It is a very suitable method for large scale optimisation problems particularly where there exist several minima. It has, for example, been successfully applied to such problems as the travelling salesman problem and while it may not guarantee finding the minimum-cost Hamiltonian-cycle it will get very close. It has also been applied to automatic the arrangement of the internal components of complicated integrated circuitry. The computer circuits are usually divided up into logical blocks and then the simulated annealing optimisation technique is used to determine the optimal layout of the interconnecting conducting pathways. One of the strong points of the simulated annealing method is that its implementation is fairly straightforward. It should also be noted that the applications noted so-far are all examples of discrete "combinatorial optimisation". In such problems we have an objective function as before, but now instead of the search space being a set of N-dimensional continuous-valued vectors it becomes a usually large set of combinations. For example the set of choices the travelling salesman makes along his way round the different cities.

Obvious the number of combinations available to us is going to grow factorially in relation to how many choices are available. This disallows us from practically exploring the whole space to examine all possible combinations, however we rely upon the assumption that a single change in choice should only result in a small change in objective function. In some sense this gives us a 'smooth' discrete search space.

The simulated annealing optimisation method can also be adapted to work with the more conventional N-dimensional vector spaces, so it will be of some use to us. Unfortunately this adaptation makes the algorithm more complicated than for combinatorial optimisation. This is caused by the problem that in many situations a step in a random direction is almost always going to be an uphill one. This requires some sophistication in order to be rectified.

The name and inspiration come from annealing in metallurgy, a technique involving the heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. At high temperatures the heat causes the atoms to become unstuck from their initial positions (a local minimum of the internal energy) and gives then the ability to wander randomly through states of higher energy. If the material is slowly cooled it is often possible for atoms to like themselves up and form a very pure completely ordered crystal which has a very low energy state. It is quite incredible that nature is able to do this itself when the distances the atoms must travel in order to form this crystal are billions of times larger than the atom itself. Conversely if the material is rapidly cooled the atoms are not given sufficient time to rearrange themselves and so we end up with a polycrystalline structure with a relatively high energy level. So the key to finding the state with the lowest energy is in the slow cooling of the material.

The algorithms so far, in some sense, have all corresponded to the behaviour of the atoms in a rapidly cooled annealing process. They have all been 'greedy' algorithms and in each case follow a downhill path as far as they can go. Fortunately our conditions on the cost and demand functions have guaranteed that this is not a problem so far. However if we wish to relax this constraint we need a method that does not simply find a local minimum, but is capable of searching for the likely global minimum. Our nature-inspired minimisation algorithm is based upon a different concept. The Boltzman probability distribution

$$\mathcal{P}(E) \sim \exp\left(-\frac{E}{kT}\right) \tag{4.2}$$

represents the idea that at a particle in a thermal system at temperature T has its energy state probabilistically distributed according to this distribution. While the temperature is high, the chance of a particle in the system being in a high energy state is also quite high so that most particles are very free to move where they feel in the system, allowing them the chance to find the lowest energy state. When the temperature is low there is still a chance although it is a lot smaller that a particle is in a high energy state and thus able to move, so that smaller moves are made. The Boltzman constant k determines our relation between temperature and energy level. What this roughly translates to is that the system can always go downhill, but uphill moves are accepted depending upon the temperature. When the temperature is high it is more likely an uphill move will be accepted while a low temperature means it will be unlikely.

Metropolis and his co-workers[**3**] built these ideas into a mathematical model of a thermodynamic system in 1953. When offered a succession of potential moves the simulated system was designed to change its configuration from energy E to

energy F with probability defined by the equation

$$p = \begin{cases} 1, & \text{if } F < E; \\ \exp\left(-\frac{F-E}{kT}\right), & \text{otherwise.} \end{cases} \tag{4.3}$$

Clearly what this translates to is that the system will always accept a downhill move (corresponding to the probability of 1) and accepts uphill moves depending upon the temperature and how far uphill the move takes us. The general scheme of always accepting downhill moves and probabilistically accepting uphill moves has become known as the *Metropolis* algorithm.

In order to fit other minimisation problems into the framework of the *Metropolis* algorithm the following key elements must be extracted:

- A concept of what is a valid system configuration;
- An objective function defining the energy-level of a given configuration;
- A function to generate random permutations of a given configuration. Each of these permutations is equivalent one of the movement choices for the algorithm;
- A parameter T representing the system's current 'temperature' with an appropriate scaling.
- An annealing schedule defining exactly how the temperature should be lowered. It should define how long the system should stay at a temperature T (how many iterations), how much T should be decreased by (usually some sort of multiplicative value) and some concept of when the system is determined to have cooled completely.

4.4.1. Continuous minimisation by simulated annealing. We now discuss how to turn the discrete idea of simulated annealing into a full continuous N-dimensional minimisation problem. In our particular example we wish to find the N-vector $\mathbf{X}$ corresponding to the minimum value of $V^{\mathrm{A1}}(\mathbf{X})$. The five elements defined earlier are now represented as follows:

- Our valid space of vectors is $\mathbf{X} \in [0, +\infty)^N$;
- The value of $V^{\mathrm{A1}}(\mathbf{X})$ is our objective function;
- A generator of random perturbations of the configuration vector describing a random step from $\mathbf{X}$ to $\mathbf{X} + \boldsymbol{\Delta}$;
- A real-valued variable T which will be our analogue of temperature;
- An annealing schedule that gradually reduces T.

The most difficult of these elements to define well is the generator of random perturbations. The difficulty arises as usual when the algorithm finds itself in a narrow valley. In this situation almost all directions randomly chosen will be uphill making this sort of generator extremely inefficient. More precisely a generator of random moves is inefficient if even when downhill moves exist it commonly produces an uphill result. Up to the current date there are perhaps only two likely contenders for efficient schemes; one detailed by Vanderbilt and Louie[**8**] in 1984 and a newer one in the book Numerical Recipes in C[**5**].

The method we will examine for performing continuous minimisation by simulated annealing is the one described by Numerical Recipes. Their method uses a modified version of the downhill simplex method that we described earlier in Section 4.1. This does unfortunately require us to maintain a simplex of $N + 1$ points. The movement of the simplex will be performed by the same procedure we described in Section 4.1, i.e. reflections, expansions and contractions. In order to add the randomness a rather clever technique is employed. A positive, logarithmically distributed random variable, proportional to the temperature T is added to

the stored function value of each of the points in the simplex and a similar random variable is subtracted from the objective function value of each new point we consider accepting as a replacement. This fits the original *Metropolis* algorithm by always accepting a downhill move but only sometimes accepting an uphill one. As $T \to 0$ the algorithm reduces to identical behaviour to the simplex method and will behave in its characteristic greedy manner converging on the closest local minimum.

When T is non-singular it causes the simplex to expand to size that approximately determines the region over which the system can reach at this temperature. It then moves according to a tumbling Brownian motion sampling new randomly chosen points inside this region. Because of the simplex-like nature of this implementation the region is explored efficiently even if it has a difficult shape, for example a narrow valley, or orientation. The idea then is that if we decrease T slowly enough then the simplex will shrink to cover the particular minimum that looks lowest of all the minima overall.

The choice of annealing schedule can require a fair amount of experimentation. What works well for one problem may not work well for another, in-fact a poor choice of schedule can cause the simulated annealing to fail to find the solution for a given problem. Again from Numerical Recipes[**5**], here some good suggestions for annealing schedule:

- Reduce T to ϵT after every m moves, where ϵ and m are determined by experiment. This is the method that was employed in the implementation here for solving variable demand equilibrium problems.
- Budget a total of K moves, and reduce T after every m moves to a value $T = T_0(1 - k/K)^\alpha$, where k is the cumulative number of moves thus far, and α is a constant, say 1, 2, or 4. The optimal value for α depends on the statistical distribution of relative minima of various depths. Larger values of α spend more iterations at lower temperature.
- After every m moves, set T to β times $f_1 - f_b$, where β is an experimentally determined constant of order 1, f_1 is the smallest function value currently represented in the simplex, and f_b is the best function ever encountered. However, never reduce T by more than some fraction γ at a time.

As we determined that in our problems there would only be a single minimum it was decided that the choice of annealing schedule would not be too important for us, so we simply picked the simplest of the three.

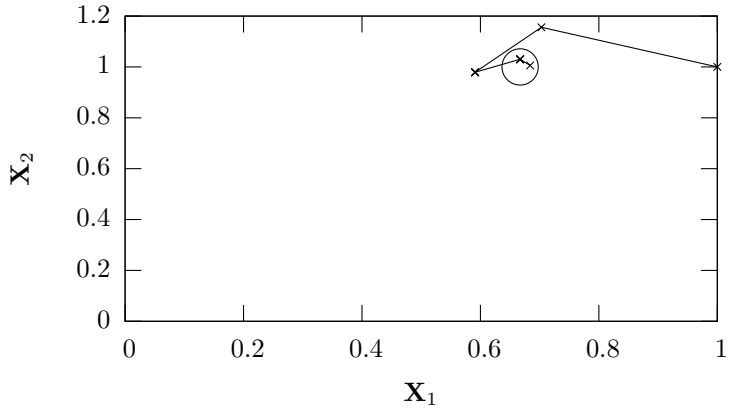
Simulated annealing has shown its worth across a great many disciplines. It has even been employed for such varied purposes as cryptanalysis, where problems previously thought to be practically insoluble have been cracked by home computers. This simply using an appropriate annealing schedule and cost function for the various combinations of bits in the encryption key.

The analogies to thermodynamics may be pursued to a greater extent than we have done here. Quantities analogous to specific heat and entropy may be defined, and these can be quite useful in monitoring the progress of the algorithm towards an acceptable solution. Information on this subject is found in [**2**].

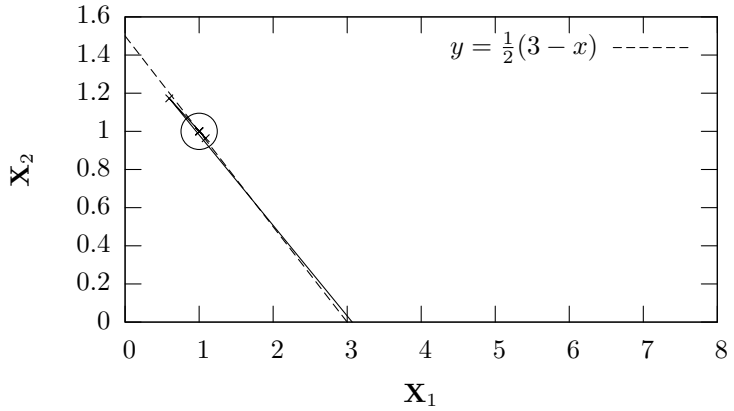
4.5. Experimental results with simulated annealing

We shall now take our simulated annealing implementation and apply it to the three standard test problems we have use throughout the document. The simulated annealing code works only in terms of flow vectors for the moment, so our initial solution vector will is $\mathbf{X}^{\text{init}} = (5, 5)^\top$. We create our initial simplex to by adding vectors of unity in each dimension's base vector. So our set of points is

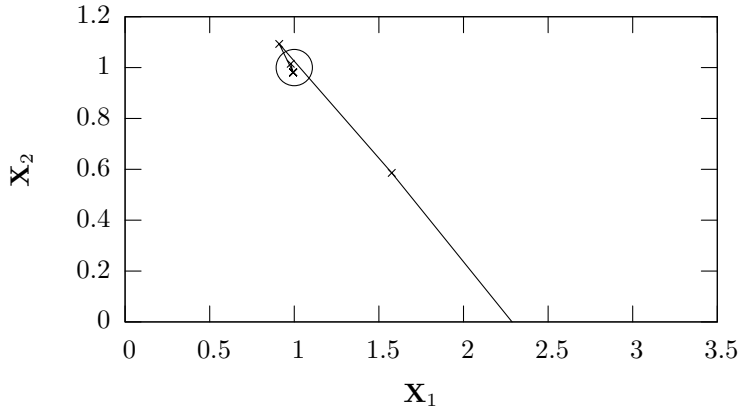
$$\left\{ \begin{pmatrix} 5 \\ 5 \end{pmatrix}, \begin{pmatrix} 5 \\ 6 \end{pmatrix}, \begin{pmatrix} 6 \\ 5 \end{pmatrix} \right\}.$$



(a) Problem 1



(b) Problem 2



(c) Problem 3

Figure 4.6: Simulated annealing method on various problems

Now our annealing schedule was defined with an initial temperature of 0.3 and a scaling factor $\epsilon = 0.9$. 10 iterations were carried out at each temperature level and the stopping temperature was defined as 0.03. In each graph the best point found so far is plotted after each batch of 10 iterations.

Problem 1. The simulated annealing code performs well against this problem as we can see in Fig. 4.6a. After only two sets of 10 iterations is is quite close to the solution. After 30 it is practically there. This level of performance is significantly better than was anticipated.

It must be remembered, though, that this particular problem was meant to be easy to solve and, for example, *algorithm 1* solved it in a single step when using optimal step-lengths.

Problem 2. When set against this problem we can see in Fig. 4.6b that after the first 10 iterations the simulated annealing method has dived straight down to the bottom of the valley along the line $y = \frac{1}{2}(3 - x)$. The simplex then contracts transversely to the direction of the valley, as we described previously, and makes its way easily along to the solution in a further 20 steps. Clearly the simplex hybridising here helps immensely.

Again it must be remembered that for more practical real-world problems the dimension of the space will be much higher. This would disallow us from using the simplex to determine our search directions, so we would have to use a different method; most likely the one discussed in [**8**].

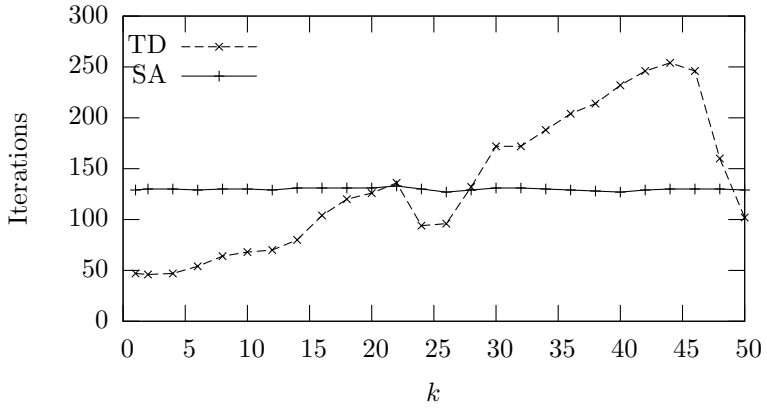
Problem 3. In Fig. 4.6c we can see that the simulated annealing code actually ventures outside the safe positive region briefly. After this brief mistake, though, it proceeds to the region of the solution in another 20 iterations. As with problem 1 this problem is quite simple to solve so we do not expect any unusual behaviour from our algorithms.

4.6. More experimental results with simulated annealing

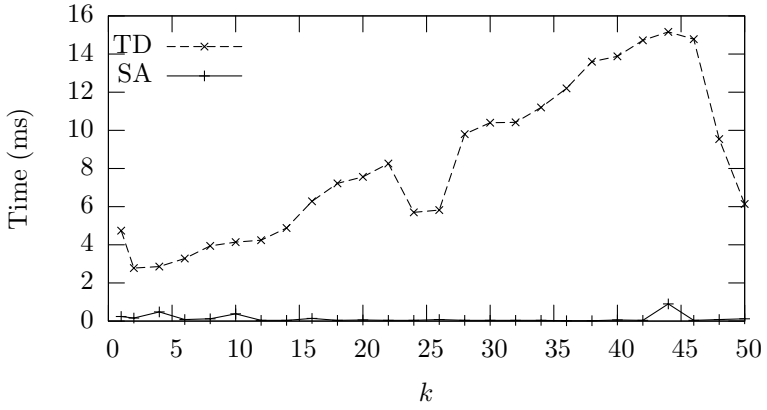
In this section we will re-run the tests we did for the two problems with variable parameters but this time employ the simulated annealing minimisation technique. As this algorithm differs significantly from the search direction based ones we will also include real-world time in our comparisons. For our tests we will perform the comparison against the *two-direction* method.

First problem. The problem is defined in (2.13). As previously an algorithm run is performed for several values of k ranging from 1 to 50. For this problem both the simulated annealing algorithm and the *two-direction* method successfully find the solution for all values of k. The number of function evaluations required for each technique are shown in Fig. 4.7a. As we would expect, the number of iterations the simulated annealing method takes is almost constant for all values of k, while the *two-direction* method's performance is heavily determined by the exact form of the problem. This is caused by the nature of the simulated annealing algorithm; we budget iterations at the start and for each temperature level.

What is also interesting now are the real-world times taken by the simulated annealing code to find the solution vector. These can be seen in Fig. 4.7b (all times are averaged over 100 algorithm runs). Clearly an efficient implementation and good choice of parameters has helped the simulated annealing code to take very little time at all, taking under 1 millisecond for all values of k! We also see a slight spike at $k = 44$, which is the same place where the *two-direction* method takes longest. Although we cannot be certain without proper analysis it is conjectured that these spikes in performance are related.



(a) Iterations



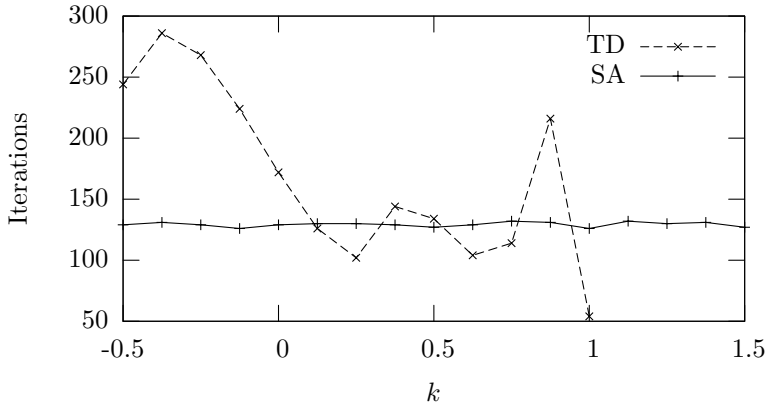
(b) Times in milliseconds

Figure 4.7: Simulated annealing method on first problem

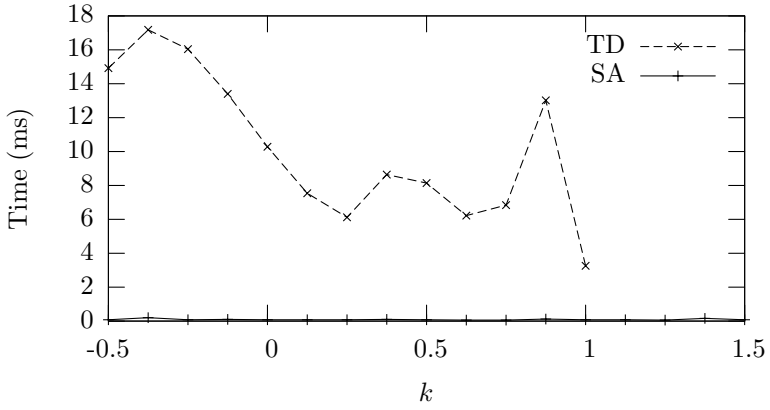
Second problem. The problem is defined in (2.14). As before we test the algorithm for values of k through the range -0.5 to $+1.5$. For this problem the simulated annealing algorithm once again finds the solution vector for all values of k while the *two-direction* method ceases to function once $k > 1$. Reliability appears to be a strong-point in the performance of the simulated annealing method. Once again the number of iterations is roughly constant for the simulated annealing algorithm across all values of k.

We again see in Fig. 4.8b that the real-world times (averaged over 100 runs) for the simulated annealing algorithm are significantly lower ($t < 1$ ms) than for the *two-direction* algorithm. This is despite them both requiring a comparable number of iterations for convergence. This again re-enforces the author's belief of the efficiency of the implementation of the simulated annealing code. It cannot be inherently this much faster than the search direction based algorithms.

Although due time constraints higher-dimensional spaces were not explored, it is suspected that performance of the simulated annealing algorithm will decline significantly in higher dimensions. Especially if we cannot use the simplex construction.



(a) Iterations



(b) Times in milliseconds

Figure 4.8: Simulated annealing method on second problem

Concluding remarks

The example problems covered in this dissertation barely begin to scratch the surface of the possible configurations of functions that make up *variable demand equilibrium* problems. Real-world problems modelling even the smallest subsection of a road network will be significantly more complicated. Fortunately even though the problems we have examined are from a very narrow branch the algorithms we have analysed should, in theory, be applicable to significantly more diversity. This is especially true of the later more conventional minimisation techniques, specifically simulated annealing.

There are many algorithms, too, that were not covered by this report even including only those specifically aimed at solving *variable demand equilibrium* problems. We did learn some simple facts, however, about the relative performance of the algorithms. For example it seems that *algorithm 1* performs extremely rapidly when it is appropriate for a particular problem, but it does have the unfortunate requirement that $-\mathbf{C}'(\mathbf{X})\mathbf{D}'(\mathbf{C}(\mathbf{X}))$ be positive definite in order for it to converge successfully to the solution. We also found that the so-caled *most obvious* search direction performs very well on a great many problems which trip *algorithm 1*, however it also has a flaw in that it is wholly inappropriate for problems with rigid demand. This flaw is fixed to a large extent by the introduction of an alternate search direction which the algorithm can alternate to to help prevent the single *most obvious* direction from getting stuck.

We also examined a relatively new Armijo-like method for determining step-lengths when gradient information is available to us. It performed extremely well when compared to the explicitly calculated optimal step-lengths available for linear functions.

We applied a combination between the *two-direction* method and the downhill simplex method where we gave the simplex the two search directions as component to work with. This actually performed quite well and further refinement of the method would likely be useful. We also examined a simulated annealing algorithm and, as would be expected, it performed extremely well. However it is likely its performance may not be sustainable at higher dimensions.

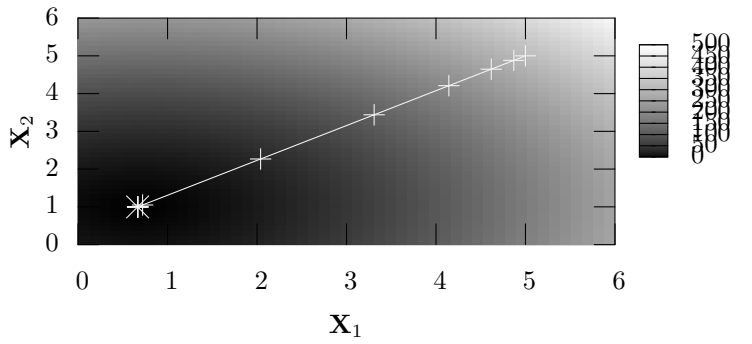
Some further work is suggested here:

- The examination of higher dimensional spaces. This would give us a greater impression of how the algorithms perform when applied to a more realistic network problem.
- The new search direction invented in Chapter 3 warrants further investigation. If it is possible to perform optimisation based only upon the flows then this reduces our overheads by approximately half.
- Although extremely good in lower dimensions, the simplex based simulated annealing code would not function at all in higher spaces. Thus we must find a replacement for the simplex algorithm it is based upon and see if we can maintain its performance.

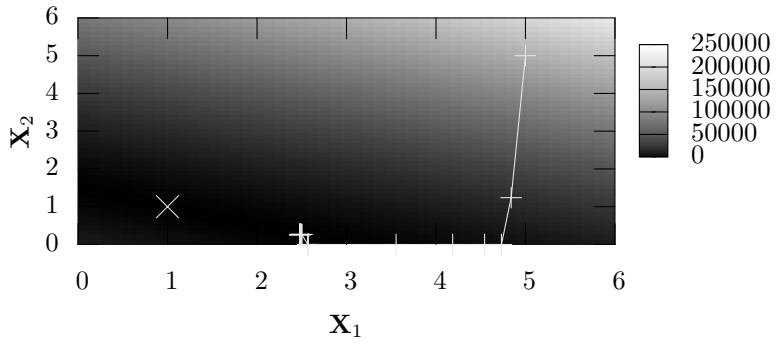
- Examining the performance of Powell's method where we employed the downhill simplex method to judge the relative performances would be interesting.
- Testing all of the algorithms with several non-linear problems to see how they fare would be a good idea.
- A replacement for the objective function V^{A1} would be highly beneficial. It is possible we could then keep the fast-performing *algorithm 1* direction if could create an objective function that didn't have unpleasant narrow valleys.
- It would be a good idea to spend a little time improving the efficiency of implementation of search direction based algorithms. They are significantly less efficient than the implementations of the conventional minimisation techniques.

APPENDIX A

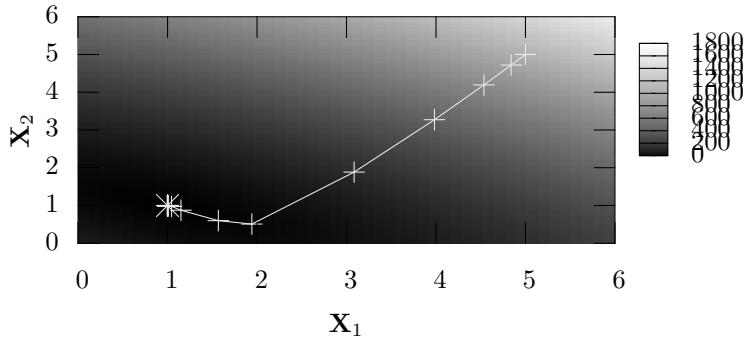
Remaining un-commented upon figures



(a) Problem 1

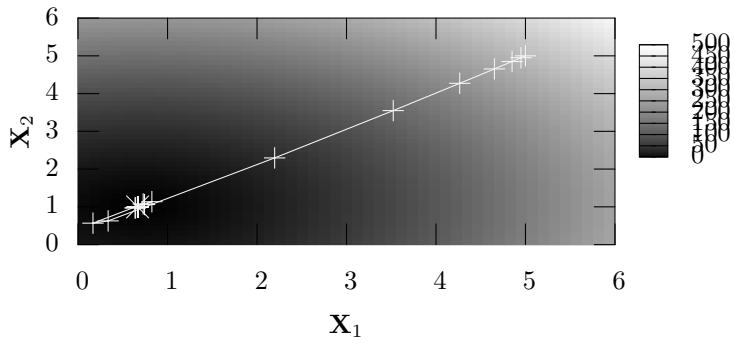


(b) Problem 2

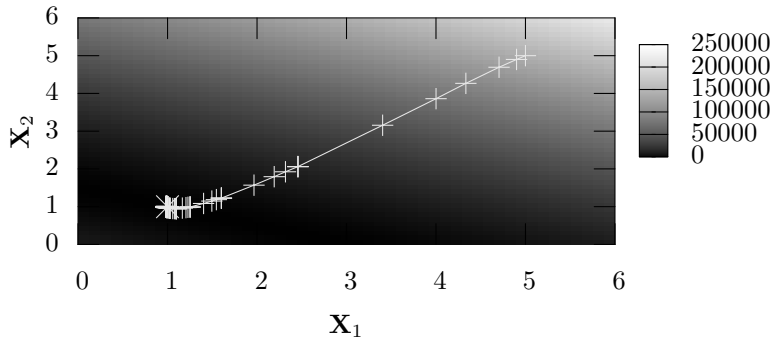


(c) Problem 3

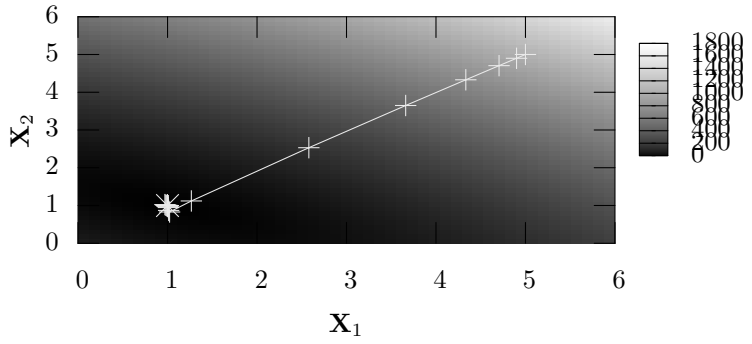
Figure A.1: 'Algorithm 1' using armijo-like step-lengths on linear flow/cost functions



(a) Problem 1

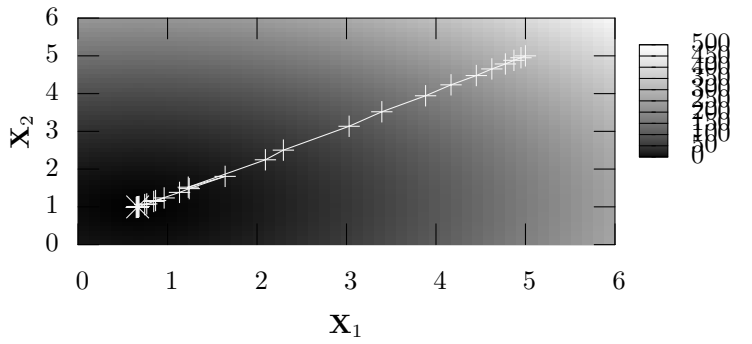


(b) Problem 2

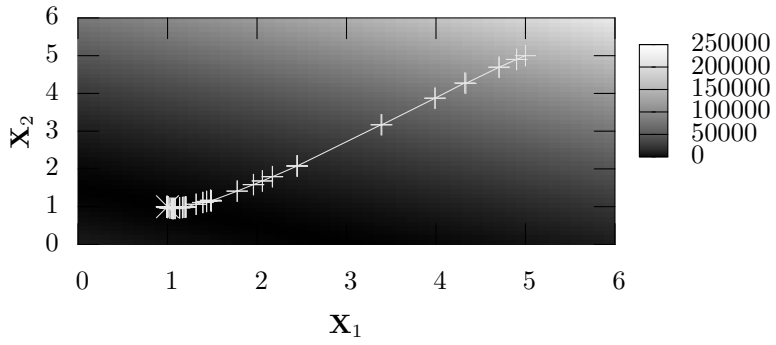


(c) Problem 3

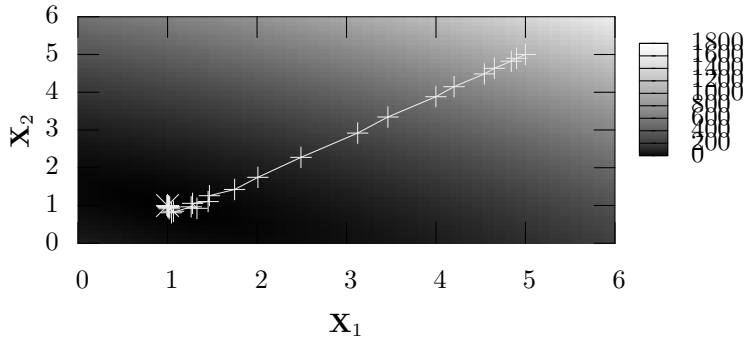
Figure A.2: 'Most obvious' using armijo-like step-lengths on linear flow/cost functions



(a) Problem 1



(b) Problem 2



(c) Problem 3

Figure A.3: 'Two-direction' using armijo-like step-lengths on linear flow/cost functions

APPENDIX B

Noteworthy code extracts

B.1. Flow and cost optimal step-length method

```
private void run(LinearFunction F, Matrix X, Matrix Y, final int decpts,
    final boolean logging)
{
    this.iterations = 0;

    boolean[] stalled = new boolean[this.numParticulars()];
    for(int i = 0; i < this.numParticulars(); i++)
        stalled[i] = false;

    double tol = Math.pow(10, -decpts-1);

    this.currentParticular = 0;

    this.lastX = X.copy();
    this.lastY = Y.copy();

    Matrix tmpX;
    Matrix tmpY;

    while(true)
    {
        Matrix[] delta = this.getParticular(this.currentParticular).delta
            (F, this.lastX, this.lastY);
        Matrix deltaX = delta[0];
        Matrix deltaY = delta[1];

        for(int i = 0; i < deltaX.getRowDimension(); i++)
        {
            if(deltaX.get(i, 0) < 0.)
                if(this.lastX.get(i, 0) < 0.1)
                    deltaX.set(i, 0, 0.001);
            if(deltaY.get(i, 0) < 0.)
                if(this.lastY.get(i, 0) < 0.1)
                    deltaY.set(i, 0, 0.001);
        }

        double L =
            this.getParticular(this.currentParticular).lambda
                (F, X, Y, delta); // LINEARISM

        tmpX = this.lastX.plus(deltaX.times(L));
```

```
        tmpY = this.lastY.plus(deltaY.times(L));

        boolean[] fixed = new boolean[1];
        L = FixVector.fix(lastX, lastY, tmpX, tmpY,
            deltaX, deltaY, L, fixed);

        if (tmpX.minus(lastX).abs().max() < tol)
            stalled[this.currentParticular] = true;
        else
            for(int i = 0; i < this.numParticulars(); i++)
                stalled[i] = false;

        boolean stop = true;
        for(int i = 0; i < this.numParticulars(); i++)
            if(!stalled[i])
            {
                stop = false;
                break;
            }
        if(stop)
            break;

        lastX = tmpX;
        lastY = tmpY;

        this.currentParticular++;
        this.currentParticular %= this.numParticulars();

        this.iterations++;
        if (this.iterations > this.iterationCap) break;
    }
}
```

B.2. Flow and cost Armijo-like method

```
private void run(Function F, Matrix X, Matrix Y,
    final double lambda, final int decpts)
{
    double tol = Math.pow(10, -decpts-1);
    this.iterations = 0;

    boolean[] stalled = new boolean[this.numParticulars()];
    for(int i = 0; i < this.numParticulars(); i++)
        stalled[i] = false;

    this.currentParticular = 0;
    double[] L = new double[this.numParticulars()];
    for(int i = 0; i < this.numParticulars(); i++)
        L[i] = lambda;

    this.lastX = X.copy();
    this.lastY = Y.copy();
```

```
Matrix tmpX;
Matrix tmpY;

while(true)
{
    Matrix[] delta = this.getParticular(this.currentParticular).delta
        (F, this.lastX, this.lastY);
    Matrix deltaX = delta[0];
    Matrix deltaY = delta[1];

    for(int i = 0; i < deltaX.getRowDimension(); i++)
    {
        if(deltaX.get(i, 0) < 0.)
            if(this.lastX.get(i, 0) < 0.1)
                deltaX.set(i, 0, 0.001);
        if(deltaY.get(i, 0) < 0.)
            if(this.lastY.get(i, 0) < 0.1)
                deltaY.set(i, 0, 0.001);
    }

    tmpX = this.lastX.plus(deltaX.times(L[this.currentParticular]));
    tmpY = this.lastY.plus(deltaY.times(L[this.currentParticular]));

    boolean[] fixed = new boolean[1];
    L[this.currentParticular] = FixVector.fix(lastX, lastY,
        tmpX, tmpY, deltaX, deltaY,
        L[this.currentParticular], fixed);

    if (tmpX.minus(lastX).abs().max() < tol)
        stalled[this.currentParticular] = true;
    else
        for(int i = 0; i < this.numParticulars(); i++)
            stalled[i] = false;

    boolean stop = true;
    for(int i = 0; i < this.numParticulars(); i++)
        if(!stalled[i])
        {
            stop = false;
            break;
        }
    if(stop)
        break;

    Matrix[] gradV = this.getParticular(this.currentParticular).gradV
        (F, lastX, lastY);
    Matrix gradV_X = gradV[0];
    Matrix gradV_Y = gradV[1];

    double g = 0.;

    for(int i = 0; i < gradV_X.getRowDimension(); i++)
```

```
    {
        g += gradV_X.get(i, 0) * deltaX.get(i, 0);
        g += gradV_Y.get(i, 0) * deltaY.get(i, 0);
    }

    double Vcurr =
        this.getParticular(this.currentParticular).V(F, lastX, lastY);
    double Vnext =
        this.getParticular(this.currentParticular).V(F, tmpX, tmpY);

    double c = (Vnext - Vcurr) / L[this.currentParticular];

    if(g > 0.)
    {
        // going uphill so stop
        L[this.currentParticular] = 0.;
    }
    else if(c > 0.)
    {
        // stay and halve L
        L[this.currentParticular] /= 2.;
    }
    else if(c > .33333333*g)
    {
        // move and halve L
        lastX = tmpX; lastY = tmpY;
        L[this.currentParticular] /= 2.;
    }
    else if(c > .66666667*g)
    {
        // move and preserve L
        lastX = tmpX; lastY = tmpY;
    }
    else if(c > g)
    {
        // move and double L
        lastX = tmpX; lastY = tmpY;
        L[this.currentParticular] *= 2.;
    }
    else
    {
        // SHOULDN'T HAPPEN
        L[this.currentParticular] = 0.;
    }

    this.iterations++;

    this.currentParticular++;
    this.currentParticular %= this.numParticulars();

    if (this.iterations > this.iterationCap) break;
}
```

```
}
```

B.3. Two-direction/simpex method

```
private void run(Function F, Matrix X, Matrix Y, final int decpts)
{
    double tol = Math.pow(10, -decpts-1);

    this.iterations = new int[1];
    this.iterations[0] = 0;

    lastX = X.copy();
    lastY = Y.copy();

    Matrix tmpX;
    Matrix tmpY;

    while(true)
    {
        Matrix[] delta0 = this.getParticular(0).delta(F, lastX, lastY);
        Matrix[] delta1 = this.getParticular(1).delta(F, lastX, lastY);
        double[] L = this.search.go(F, this.getParticular(0),
            lastX, lastY, delta0, delta1, tol, filename,
            (this.iterations[0]==0)?false:true, this.iterations);

        Matrix deltaX = delta0[0].times(L[0]).plus(delta1[0].times(L[1]));
        Matrix deltaY = delta0[1].times(L[0]).plus(delta1[1].times(L[1]));

        tmpX = lastX.plus(deltaX);
        tmpY = lastY.plus(deltaY);

        for(int i = 0; i < tmpX.getRowDimension(); i++)
        {
            if(tmpX.get(i, 0) < 0.)
                tmpX.set(i, 0, 0.1);
            if(tmpY.get(i, 0) < 0.)
                tmpY.set(i, 0, 0.1);
        }

        if (tmpX.minus(lastX).abs().max() < tol)
            break;

        if (this.iterations[0] > this.iterationCap) break;

        lastX = tmpX;
        lastY = tmpY;
    }
}
```

Acknowledgements

Most of all I would like to thank the legendary Professor Mike Smith for the continuous inspiration he provided me with. I would also like to thank my wonderful friend Lizzy for keeping me sane and helping me to believe in myself. I would probably be significantly more of a wreck than I am right now if not for her. Thanks also to James Springham for producing an excellent dissertation last year full of excellent ideas for me to improve upon and work with. If I ever meet him I most definitely owe him a pint.

My gratitude also goes to the EPSRC for giving me the money to allow me to attend this MSc course in Mathematics.

Bibliography

1. S. Kirkpatrick, *Simulated annealing*, Journal of Statistical Physics **34** (1984), 975–986.
2. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, *Optimization by Simulated Annealing*, Science **220** (1983), no. 4598, 671–680.
3. Nicholas Constantine Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller, *Equation of state calculations by fast computing machines*, Journal of Chemical Physics **21** (1953), 1087–1092.
4. J. A. Nelder and R. Mead, *A simplex method for function minimization*, Computer Journal, vol. 7, 1965, pp. 308–313.
5. W.H. Press, B.P. Flannery, S.A Teukolsky, and W.T. Vetterling, *Numerical recipies in c: The art of scientific computing*, Cambridge University Press, New York, New York, USA, 1988.
6. M. J. Smith, *Two-direction methods of calculating variable demand equilibria: objective function, search direction and proof of descent.*, DIADEM research note, 2004.
7. J. Springham, *On solving variable demand equilibrium problems*, Thesis submitted for the degree of MSc at the University of York (2004).
8. D. Vanderbilt and S. G. Louie, *Simulated annealing*, Journal of Computational Physics **56** (1984), 259—271.