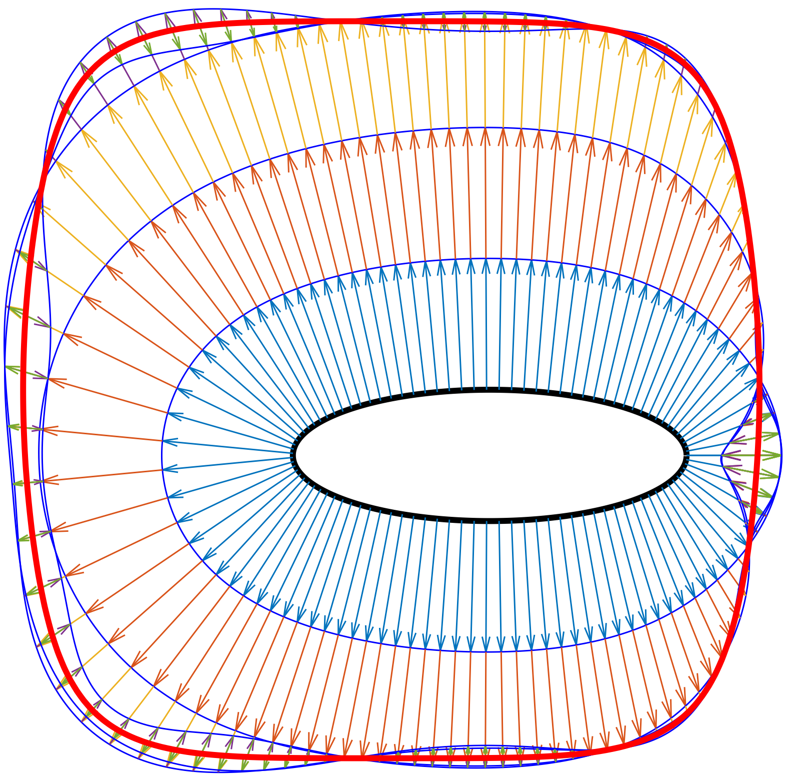
Shape Optimsation of Two-Dimensional Level Set Functions

Mike Fuller (michael.fuller@pmb.ox.ac.uk)
Supervised by Dr Alberto Paganini (paganini@maths.ox.ac.uk)

9th August 2018



Abstract

Shape optimisation is typically concerned with finding the shape Ω which is optimal in the sense that it minimises a certain cost functional $\mathcal{J}[\Omega]$. In this report we briefly discuss unconstrained optimisation methods for functions before introducing the notion of a shape derivative, which we will use to create a shape optimisation method that minimises functionals of the form

$$\mathcal{J}[\Omega] := \int_{\Omega} f \, \mathrm{d}x.$$

We will perform numerical and theoretical investigations in MATLAB to test the success of our shape optimisation method, firstly by considering the functional with f as the unit disk

$$f(x) = x_1^2 + x_2^2 - 1,$$

then by considering an interesting test case with a non-convex level set function.

Acknowledgements

I would like to thank my supervisor Alberto for allowing me to pursue this summer project, and for regularly giving me his time and advice throughout the 6-week period. I would also like to thank my project colleagues David and Christopher - I am deeply grateful to have had the opportunity to discuss our ideas together.

I want to thank Chris Rokos for his generosity in providing the funding towards my project by means of the Rokos Awards - without it, I would have not had the chance to write this report and gain first hand experience in scientific research. Finally I would like to thank my family and friends for their continuous support.

Contents

1 Unconstrained Optimisation

We first recap on some iterative methods for solving unconstrained optimisation problems. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function. Suppose we want to find a value x^* such that f is minimal, i.e

$$x^* \in \arg\min_{x \in \mathbb{R}^n} f(x).$$

We can find such a value by means of an iterative algorithm. That is, given an initial guess $x^{(0)}$ for x^*, we construct a *minimising sequence* $(x^{(k)})$ of points such that

$$f(x^{(k)}) \longrightarrow f(x^*) \quad \text{as } k \to \infty.$$

Often we only require an approximate solution, so we can reduce computational effort by choosing a tolerance $\varepsilon > 0$ such that when $f(x^{(k)}) - f(x^*) \leqslant \varepsilon$, we terminate the algorithm.

1.1 Descent Methods

The minimising sequences we are going to discuss in this section are all of the form

$$x^{(k+1)} = x^{(k)} + s^{(k)} \Delta x^{(k)},$$

where

- $\Delta x^{(k)}$ represents a *search direction* - which direction we travel in to approach x^*,
- $s^{(k)} \geqslant 0$ represents a *step length* - how far we travel along the vector $\Delta x^{(k)}$.

Recall that our aim is to minimise f, so we would like the sequence to have the property

$$f(x^{(k+1)}) < f(x^{(k)}) \tag{1}$$

for all k, unless $x^{(k)}$ is optimal. How can we ensure that our choice of search direction results in this property being satisfied? If f is differentiable, we can Taylor expand up to first-order and write

$$f(x^{(k+1)}) = f(x^{(k)} + s^{(k)} \Delta x^{(k)}) \approx f(x^{(k)}) + s^{(k)} \mathrm{d}f(x^{(k)}, \Delta x^{(k)}),$$

where $\mathrm{d}f$ denotes the *directional derivative*

$$\mathrm{d}f(x, v) := \lim_{t \to 0} \frac{f(x + tv) - f(x)}{t}.$$

Hence, if $x^{(k)}$ is not optimal, $s^{(k)} \neq 0$ and we can ensure (1) holds up to first-order if

$$\mathrm{d}f(x^{(k)}, \Delta x^{(k)}) < 0. \tag{2}$$

A search direction $\Delta x^{(k)}$ such that (2) holds is called a *descent direction.*

A generic descent method then goes as follows:

```
1 Given starting point $x$
2 repeat
3     (1) Determine descent direction, $\Delta x$
4     (2) Choose step size, $s > 0$
5     (3) Set $x \leftarrow x + s\Delta x$
6 until stopping criterion satisfied
```

At the moment this is rather vague. We have a few questions to answer:

- How do we determine the descent direction?
- How do we choose the step size?
- When do we stop the algorithm?

1.2 Finding a Descent Direction

By analysing the Taylor expansion of $f(x + s\Delta x)$ about x up to first-order, we can find a suitable descent direction (which will be used later in the *Steepest Descent* method). The expansion reads

$$f(x + s\Delta x) \approx f(x) + s\,\mathrm{d}f(x, \Delta x),$$

so for fixed $s \geqslant 0$, the approximation is smallest when we minimise $\mathrm{d}f(x, \Delta x)$ over all descent directions Δx. We define a *normalised steepest descent direction* Δx_{nsd} to be a descent direction of unit length that minimises the directional derivative, i.e

$$\Delta x_{\text{nsd}} \in \operatorname*{arg\,min}_{\Delta x\,:\,||\Delta x||=1} \ \mathrm{d}f(x, \Delta x)$$

for some norm $||\cdot||$ on $\mathbb{R}^n$.

To piece together the *Steepest Descent* method, it will be useful to introduce the concept of inner products and gradients, particularly when it comes to defining stopping criteria. For a real vector space V, an *inner product* is a function $(\cdot,\cdot) : V^2 \to \mathbb{R}$ which satisfies the following three properties:

(A) *Linearity:* $(\alpha u + \beta v, w) = \alpha(u, w) + \beta(v, w) \quad \forall u, v, w \in V,\ \alpha, \beta \in \mathbb{R}$

(B) *Symmetry:* $(u, v) = (v, u) \quad \forall u, v \in V$

(C) *Positive definitenesss:* $(u, u) \geqslant 0 \quad \forall u \in V, \quad (u, u) = 0 \Longleftrightarrow u = 0.$

We define $\nabla f(x)$ to be the *gradient* of f at x with respect to an inner product $(\cdot,\cdot)$ if

$$(\nabla f(x), v) = \mathrm{d}f(x, v) \quad \forall v.$$

We occasionally write $\nabla_{(\cdot,\cdot)} f(x)$ to make it clear what the associated inner product is.

The following lemma uses these definitions to write Δx_{nsd} explicitly.

Lemma 1.1: If a norm $||\cdot||$ is induced by an inner product $(\cdot,\cdot)$, i.e $||v|| = \sqrt{(v, v)}$, then

$$\Delta x_{\text{nsd}} = -\frac{\nabla f(x)}{||\nabla f(x)||}. \tag{3}$$

Proof: We use the method of Lagrange multipliers. Let

$$\mathcal{L}(\Delta x; \lambda) := \mathrm{d}f(x, \Delta x) + \frac{\lambda}{2}(||\Delta x||^2 - 1).$$

Then by definition of Δx_{nsd},

$$\begin{aligned} & 0 = \mathrm{d}\mathcal{L}(\Delta x_{\text{nsd}}, v; \lambda) = \mathrm{d}f(x, v) + \lambda(\Delta x_{\text{nsd}}, v) \quad \forall v \\ \Longleftrightarrow \quad & (-\lambda \Delta x_{\text{nsd}}, v) = \mathrm{d}f(x, v) \quad \forall v \\ \Longleftrightarrow \quad & -\lambda \Delta x_{\text{nsd}} = \nabla f(x). \end{aligned}$$

Taking norms of both sides, and using the fact that $||\Delta x_{\text{nsd}}|| = 1$, we obtain $|\lambda| = ||\nabla f(x)||$. The result follows, using the fact that Δx_{nsd} is a descent direction.

□

1.3 Line Search

To answer the second question from Section 1.1, we introduce two methods for choosing the step size: *exact line search* (ELS) and *backtracking line search* (BLS). Here we assume that Δx is any descent direction.

In ELS, we choose s such that f is minimised along the ray $\{x + t\Delta x : t \geqslant 0\}$, i.e

$$s = \arg\min_{t \geqslant 0} f(x + t\Delta x).$$

It may be difficult to find s exactly, so it is generally cost effective to choose a step size such that f is only approximately minimised along the ray. For this reason we may choose to apply BLS, which generates the step size by the following algorithm:

Algorithm 1: Backtracking Line Search

```
1 Given descent direction $\Delta x$, $\alpha \in (0, 0.5), \beta \in (0, 1)$
2 Set $s \leftarrow 1$
3 while $f(x + s\Delta x) > f(x) + \alpha s \,\mathrm{d}f(x, \Delta x)$ do
4     Set $s \leftarrow \beta s$
5 end
```

The resulting value of s is chosen as the step size. The line search is called "backtracking" as it starts with unit step size, then reduces it by the factor β until the stopping criterion holds. Since Δx is a descent direction, we have $\mathrm{d}f(x, \Delta x) < 0$, so

$$f(x + s\Delta x) \approx f(x) + s\,\mathrm{d}f(x, \Delta x) < f(x) + \alpha s\,\mathrm{d}f(x, \Delta x)$$

for small enough s, demonstrating that BLS eventually terminates. The choice of parameters α, β has a noticeable but not dramatic effect on convergence [1, page 475]; though ELS sometimes improve convergence it is generally not worth the computational effort of using it [1, page 464].

1.4 Steepest Descent

From Lemma 1.1, we see that Δx_{nsd} is a step in the unit ball of $||\cdot||$ which extends farthest in the direction of $-\nabla f(x)$. It is more convenient in practice to use an unnormalised descent direction

$$\Delta x_{\text{sd}} := ||\nabla f(x)||_* \Delta x_{\text{nsd}},$$

where $||\cdot||_*$ denotes the *dual norm*

$$||v||_* := \sup\{v^T x : ||x|| \leqslant 1\}.$$

We find that Δx_{sd} is simply the unnormalised version of (3), that is

$$\Delta x_{\text{sd}} = -\nabla_{(\cdot,\cdot)} f(x),$$

and we arrive at the *Steepest Descent* method, which uses Δx_{sd} as a descent direction:

Algorithm 2: Steepest Descent

```
1 Given starting point $x$
2 repeat
3     (1) Set $\Delta x \leftarrow \Delta x_{\text{sd}}$
4     (2) Choose $s$ via ELS/BLS
5     (3) Set $x \leftarrow x + s\Delta x$
6 until stopping criterion satisfied
```

To define a stopping criterion, recall that when x^* is optimal we have

$$\mathrm{d}f(x^*, v) = 0 \quad \forall v,$$

so for a sufficiently smooth f, we expect the gradient to be small for $x \approx x^*$. For this reason, we can define a stopping criterion of the form

$$||\nabla f(x)|| \leqslant \varepsilon$$

for some small $\varepsilon > 0$. Most practical uses of Steepest Descent check the stopping criterion after the first step in the repeat loop - we will see this shortly in *Newton's Method.*

1.5 Newton's Method

Which inner product do we use to define Δx_{sd}? An interesting choice, which forms the basis of *Newton's Method*, is to consider the inner product induced by the second derivative

$$\mathrm{d}^2 f(x, u, v) := \lim_{t \to 0} \frac{\mathrm{d}f(x + tv, u) - \mathrm{d}f(x, u)}{t}.$$

If it exists, we can extend our Taylor approximation up to second-order: expanding $f(x+\Delta x)$ gives

$$f(x + \Delta x) \approx f(x) + \mathrm{d}f(x, \Delta x) + \frac{1}{2}\,\mathrm{d}^2 f(x, \Delta x, \Delta x) =: g(\Delta x).$$

Assume that g is *strictly convex** so that g has a unique minimiser, Δx_{nt}. In this case, $\mathrm{d}^2 f(x, \cdot, \cdot)$ is positive definite [2, page 640]. The second derivative is also symmetric and linear, making it an inner product. By minimality, Δx_{nt} must satisfy

$$\begin{aligned} & \mathrm{d}g(\Delta x_{\mathrm{nt}}, v) = 0 \quad \forall v \\ \iff \quad & \mathrm{d}f(x, v) + \mathrm{d}^2(x, \Delta x_{\mathrm{nt}}, v) = 0 \quad \forall v \\ \iff \quad & \mathrm{d}^2(x, \Delta x_{\mathrm{nt}}, v) = -\mathrm{d}f(x, v) \quad \forall v \end{aligned}$$

so Δx_{nt} is the negative gradient with respect to the inner product induced by $\mathrm{d}^2 f$, that is,

$$\Delta x_{\mathrm{nt}} = -\nabla_{\mathrm{d}^2 f(x,\cdot,\cdot)} f(x).$$

If f is quadratic then $f(x + \Delta x) = g(\Delta x)$, so $x + \Delta x_{\mathrm{nt}}$ is the *exact* minimiser of f. If we assume that the second derivative of f is continuous, then the quadratic model of f is very accurate for $x \approx x^*$, so $x + \Delta x_{\mathrm{nt}}$ is a very good estimate for x^*.

Before we introduce *Newton's method*, we will construct a stopping criterion using the quadratic appoximation of f, as demonstrated in the following lemma:

Lemma 1.2: $\frac{1}{2}\mathrm{d}f(x, \Delta x_{\mathrm{nt}})$ estimates the error $f(x) - f(x^*)$, based on the quadratic approximation of f.

Proof: Up to second-order, we have $f(x^*) = \inf_y g(y) = g(\Delta x_{\mathrm{nt}})$. Thus the error is

$$\begin{aligned} f(x) - g(\Delta x_{\mathrm{nt}}) &= f(x) - \left(f(x) + \underbrace{\mathrm{d}f(x, \Delta x_{\mathrm{nt}})}_{=0} + \frac{1}{2}\mathrm{d}^2 f(x, \Delta x_{\mathrm{nt}}, \Delta x_{\mathrm{nt}}) \right) \\ &= f(x) - \left(f(x) - \frac{1}{2}\mathrm{d}f(x, \Delta x_{\mathrm{nt}}) \right) \\ &= \frac{1}{2}\mathrm{d}f(x, \Delta x_{\mathrm{nt}}). \end{aligned}$$

□

*A function g is said to be *strictly convex* if the line segment connecting any two distinct points on the surface of g lies strictly above g, except at the endpoints.

We use this as a stopping criterion for *Newton's Method*, which goes as follows:

Algorithm 3: Newton's Method

```
1 Given starting point $x$, tolerance $\varepsilon > 0$
2 repeat
3     (1) Set $\Delta x \leftarrow \Delta x_{\mathrm{nt}}$
4     (2) Stop if $\mathrm{d}f(x, \Delta x)/2 \leqslant \varepsilon$
5     (3) Choose $s$ via BLS
6     (4) Set $x \leftarrow x + s\Delta x$
```

Observe that this is more or less a general descent method, with the difference that the stopping criterion is checked after computing Δx_{nt}, rather than after updating the value of x.

Newton's Method has many advantages over Steepest Descent. Most importantly it has *quadratic convergence* near x^* [1, page 496] - roughly speaking, this means that the number of correct digits doubles after each iteration. As a result, Newton's Method also scales with the size of the problem: it will perform just as well for a problem in $\mathbb{R}^{10000}$ as it would for problems in $\mathbb{R}^{10}$ say, with only a moderate increase in the number of iterations required.

A pitfall of Newton's Method is the cost of computing Δx_{nt}, which requires solving a system of linear equations. *Quasi-Newton* methods require less cost to form the search direction, sharing some advantages of Newton's Method such as rapid convergence near x^*, but we will not discuss such methods here.

2 Introduction to Shape Optimisation

In the previous section, we had a function $f : \mathbb{R}^n \to \mathbb{R}$ and we used its derivatives to find the points x^* which minimised $f(x)$. What if we were instead given a cost functional $\mathcal{J}[\Omega]$, with the aim of minimising $\mathcal{J}$ over bounded domains (shapes) Ω? We would need a notion of *shape differentiation.*

In this section we define the *shape derivative.* We will ultimately use such derivatives to find a domain Ω^* in a collection of admissible shapes $\mathcal{U}_{\mathrm{ad}}$ which minimises a given cost functional $\mathcal{J}$. For simplicity, we restrict ourselves to functionals of the form

$$\mathcal{J}[\Omega] := \int_\Omega f \, \mathrm{d}x.$$

In the style of Section 1.1, we write

$$\Omega^* \in \operatorname*{arg\,min}_{\Omega \in \mathcal{U}_{\mathrm{ad}}} \mathcal{J}[\Omega].$$

2.1 Defining the Shape Derivative

Let $T : \mathbb{R}^n \to \mathbb{R}^n$ be a sufficiently smooth vector field. Then

$$\mathcal{J}[T(\Omega)] = \int_{T(\Omega)} f \, \mathrm{d}x = \int_\Omega (f \circ T) |\det \mathbf{D}T| \, \mathrm{d}x,$$

where $\mathbf{D}T$ denotes the *Jacobian* of T. Note that $T(\Omega) \neq \Omega$ in general, and similarly $\mathcal{J}[T(\Omega)] \neq \mathcal{J}[\Omega]$.

Furthermore let $\mathcal{V} : \mathbb{R}^n \to \mathbb{R}^n$ be a vector field, and define a duality pairing

$$\left\langle \frac{\mathrm{d}}{\mathrm{d}T} \mathcal{J}[T(\Omega)], \mathcal{V} \right\rangle := \lim_{t \to 0} \frac{\mathcal{J}[(T + t\mathcal{V})(\Omega)] - \mathcal{J}[T(\Omega)]}{t}.$$

We then define the *Eulerian (shape) derivative* of $\mathcal{J}$ in the direction $\mathcal{V}$ as

$$\mathrm{d}\mathcal{J}(\Omega,\mathcal{V}) := \left\langle \frac{\mathrm{d}}{\mathrm{d}T}\mathcal{J}[T(\Omega)],\mathcal{V}\right\rangle\Bigg|_{T=I},$$

where I is the identity map $I(\Omega)=\Omega$. Therefore, we can write

$$\begin{aligned}\mathrm{d}\mathcal{J}(\Omega,\mathcal{V}) &= \left\langle \frac{\mathrm{d}}{\mathrm{d}T}\left(\int_\Omega (f\circ T)|\det \mathbf{D}T|\,\mathrm{d}x\right),\mathcal{V}\right\rangle\Bigg|_{T=I}\\ &= \int_\Omega \left\langle \frac{\mathrm{d}}{\mathrm{d}T}((f\circ T)\det\mathbf{D}T),\mathcal{V}\right\rangle\Bigg|_{T=I}\mathrm{d}x. \qquad (4)\end{aligned}$$

Ideally we want $\mathrm{d}\mathcal{J}$ to exist for all $\mathcal{V}$. We say that $\mathcal{J}$ is *shape differentiable* at Ω if the map

$$\mathcal{V}\mapsto \mathrm{d}\mathcal{J}(\Omega,\mathcal{V})$$

is linear and bounded on $C^1(\mathbb{R}^n,\mathbb{R}^n)$, the set of continuously differentiable maps from $\mathbb{R}^n$ to $\mathbb{R}^n$ [3].

2.2 Computing the Shape Derivative

Generally we do not know the vector field T, so (4) is not very useful. We now prove two formulae for the shape derivative $\mathrm{d}\mathcal{J}$ that do not depend on T, which will help us shortly to compute shape derivatives in practice.

Theorem 2.1: Let $\nabla\cdot\mathcal{V}$ denote the *divergence* of $\mathcal{V}$. Then

$$\mathrm{d}\mathcal{J}(\Omega,\mathcal{V}) = \int_\Omega \nabla f\cdot\mathcal{V} + f\nabla\cdot\mathcal{V}\,\mathrm{d}x.$$

Proof: By applying the product rule to (4), we have

$$\mathrm{d}\mathcal{J}(\Omega,\mathcal{V}) = \int_\Omega \left\{\left\langle\frac{\mathrm{d}}{\mathrm{d}T}(f\circ T),\mathcal{V}\right\rangle\det\mathbf{D}T + \left\langle\frac{\mathrm{d}}{\mathrm{d}T}\det\mathbf{D}T,\mathcal{V}\right\rangle(f\circ T)\right\}\Bigg|_{T=I}\mathrm{d}x.$$

Considering the first term, we have

$$\left\langle\frac{\mathrm{d}}{\mathrm{d}T}(f\circ T),\mathcal{V}\right\rangle = (\nabla f\circ T)\cdot\mathcal{V} \quad\Longrightarrow\quad \left\langle\frac{\mathrm{d}}{\mathrm{d}T}(f\circ T),\mathcal{V}\right\rangle\det\mathbf{D}T\Bigg|_{T=I} = \nabla f\cdot\mathcal{V}.$$

Considering the second term and using Jacobi's formula [4] to expand $\det\mathbf{D}T$, we have

$$\begin{aligned}\left\langle\frac{\mathrm{d}}{\mathrm{d}T}\det\mathbf{D}T,\mathcal{V}\right\rangle &= \lim_{t\to0}\frac{\det\mathbf{D}(T+t\mathcal{V})-\det\mathbf{D}T}{t}\\ &= \lim_{t\to0}\frac{\det(\mathbf{D}T+t\mathbf{D}\mathcal{V})-\det\mathbf{D}T}{t}\\ &= \lim_{t\to0}\frac{(\det\mathbf{D}T+t\,\mathrm{Tr}(\mathrm{adj}(\mathbf{D}T)\mathbf{D}\mathcal{V})+O(t^2))-\det\mathbf{D}T}{t} = \mathrm{Tr}(\mathrm{adj}(\mathbf{D}T)\mathbf{D}\mathcal{V}),\end{aligned}$$

where $\mathrm{Tr}(A),\mathrm{adj}(A)$ denote the *trace* and *adjugate*[†] of A respectively. Hence

$$\left\langle\frac{\mathrm{d}}{\mathrm{d}T}\det\mathbf{D}T,\mathcal{V}\right\rangle(f\circ T)\Bigg|_{T=I} = \mathrm{Tr}(\mathrm{adj}(\mathrm{Id})\mathbf{D}\mathcal{V})f = \mathrm{Tr}(\mathbf{D}\mathcal{V})f = f\nabla\cdot\mathcal{V}$$

since $\mathrm{adj}(\mathrm{Id})=\mathrm{Id}$ is the identity matrix, and the result follows. □

[†]For a matrix A, the *adjugate* of A is the transpose of the *cofactor matrix* C, where $c_{ij}=(-1)^{i+j}\det A_{ij}$, and A_{ij} is the matrix A with row i and column j removed. For an invertible matrix, this is simply $\mathrm{adj}A=(\det A)A^{-1}$.

Theorem 2.2: The shape derivative is equal to the surface integral

$$\mathrm{d}\mathcal{J}(\Omega, \mathcal{V}) = \int_{\partial\Omega} f(\mathcal{V} \cdot n)\,\mathrm{d}S,$$

where n is the outward pointing unit normal of the boundary $\partial\Omega$.

Proof: Using Theorem 2.1 and the Divergence Theorem,

$$\begin{aligned}\mathrm{d}\mathcal{J}(\Omega, \mathcal{V}) &= \int_{\Omega} \nabla f \cdot \mathcal{V} + f\nabla \cdot \mathcal{V}\,\mathrm{d}x \\ &= \int_{\Omega} \nabla \cdot (f\mathcal{V})\,\mathrm{d}x \\ &= \int_{\partial\Omega} (f\mathcal{V}) \cdot n\,\mathrm{d}S\end{aligned}$$

where the second equality follows from vector calculus. The result follows. □

3 Shape Optimisation in Matlab

We will use Theorem 2.2 to help us develop an efficient shape optimisation method in Matlab for two-dimensional problems. To construct a descent method for shapes, we first need a minimising sequence of shapes, say

$$\Omega^{(k+1)} = T^{(k)}(\Omega^{(k)})$$

for a sequence of smooth vector fields $T^{(k)}$. As mentioned in Section 2.2, the $T^{(k)}$ are generally unknown, certainly when doing computations in Matlab, so we need to represent the problem in a different manner.

3.1 Descent Methods for Shapes

How could we define the next shape in the sequence in a similar manner to Section 1.1? We could define the shape by its boundary, then *add* shape boundaries by taking their parametrisations and summing component-wise. It then makes sense to write

$$\partial\Omega^{(k+1)} = \partial\Omega^{(k)} + s^{(k)}\Delta\partial\Omega^{(k)},$$

where

- $\Delta\partial\Omega^{(k)}$ represents a *(boundary) search direction*,
- $s^{(k)} \geqslant 0$ represents a *step length*.

In Matlab, we can illustrate this with the help of the *Chebfun* package. For a 2D problem, we can define the boundary $\partial\Omega$ as being in the complex plane, as the following example demonstrates.

Example 3.1 (Superposition of shape boundaries): Let R_1 be the ellipse

$$R_1 := \left\{ x : \frac{(x_1 - 0.1)^2}{0.6^2} + \frac{(x_2 - 1.2)^2}{1.05^2} \leqslant 1 \right\}.$$

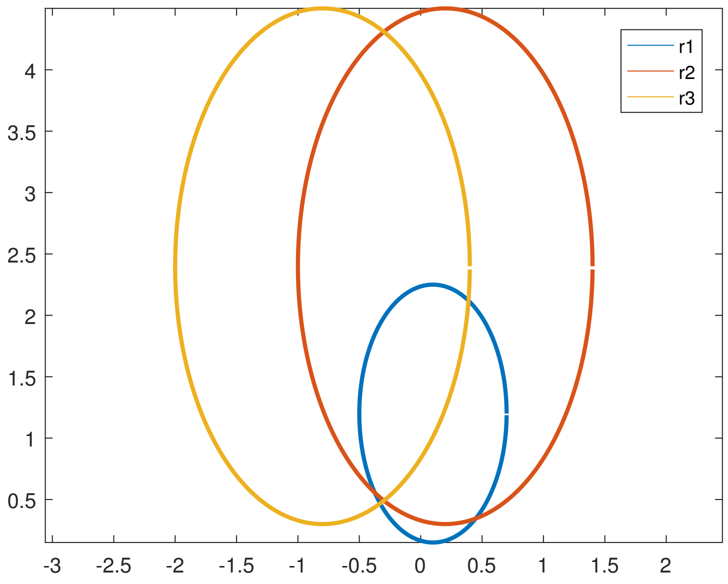
Defining the shape by its boundary using *Chebfun*, we can write the following:

```
r = @(t) [0.1 + 1.2i] + 0.6*cos(t) + 1.05*1i*sin(t);
t = linspace(0,2*pi);
r1 = chebfun(@(t) r(t), [0, 2*pi], 'trig'); %Parametrised boundary
```

Now consider scaling R_1 by a factor of 2, and translating the resulting shape in the direction of the negative real axis by 1. Call these shapes R_2 and R_3 respectively. In *Chebfun*, functions are treated as variables [5] and we can simply write the following:

```
r2 = 2*r1; %Scaled boundary
r3 = r2 - [1 + 0i]; %Scaled and translated boundary
```

Finally, we check our intuition by plotting `r1`, `r2` and `r3`:



3.2 Finding a Boundary Search Direction

We can find a suitable search direction by using the gradient definition in Section 1.2 for functionals instead of functions. If the shape derivative is linear with respect to $\mathcal{V}$, which is true for differentiable shapes, then there is a unique element of $\nabla\mathcal{J} \in L^2(\partial\Omega)$ [6], called the *shape gradient* of $\mathcal{J}$, and

$$\mathrm{d}\mathcal{J}(\Omega, \mathcal{V}) = (\nabla\mathcal{J}, \mathcal{V})_{\partial\Omega} := \int_{\partial\Omega} \nabla\mathcal{J} \cdot \mathcal{V} \,\mathrm{d}S,$$

where $(\cdot,\cdot)_{\partial\Omega}$ is the L^2-inner product on $\partial\Omega$. Using Theorem 2.2, we deduce that

$$\nabla\mathcal{J}\big|_{\partial\Omega} = fn.$$

So if we choose our boundary search direction to be the negative gradient

$$\Delta\partial\Omega := -\nabla\mathcal{J}\big|_{\partial\Omega} = -fn,$$

then the shape derivative of $\mathcal{J}$ in the direction $\Delta\partial\Omega$ is

$$\mathrm{d}\mathcal{J}(\Omega, \Delta\partial\Omega) = -\int_{\partial\Omega} f(fn \cdot n)\,\mathrm{d}S = -\int_{\partial\Omega} f^2 \,\mathrm{d}S, \tag{5}$$

and we deduce that the shape derivative vanishes when $f = 0$ on $\partial\Omega$. This intuitively makes sense; to minimise $\mathcal{J}$, we should integrate over the shape $\Omega^* = \{x : f(x) \leqslant 0\}$, with the boundary being the zero level set of f. Integrating over shapes outside of Ω^* would only increase the value of $\mathcal{J}$. Furthermore, if $f \neq 0$ on $\partial\Omega$, (5) tells us that $\mathrm{d}\mathcal{J}(\Omega, \Delta\partial\Omega) < 0$. We will call any search direction $\Delta\partial\Omega$ with this property a *(boundary) descent direction.*

With all this in mind, we turn to our first example of a shape optimisation method in MATLAB.

Example 3.2 (Simple shape optimisation method): Let $f : \mathbb{R}^2 \to \mathbb{R}$ be the function defined by

$$f(x) = x_1^2 + x_2^2 - 1.$$

We know that $\mathcal{J}(\Omega)$ is minimised when integrated over the unit disk $\Omega = \{x : x_1^2 + x_2^2 \leqslant 1\}$.

We implement a basic shape optimisation method in MATLAB, with initial guess

$$\Omega^{(0)} := \left\{ x : \frac{(x_1 - 0.2)^2}{0.6^2} + \frac{(x_2 - 0.3)^2}{0.2^2} \leqslant 1 \right\},$$

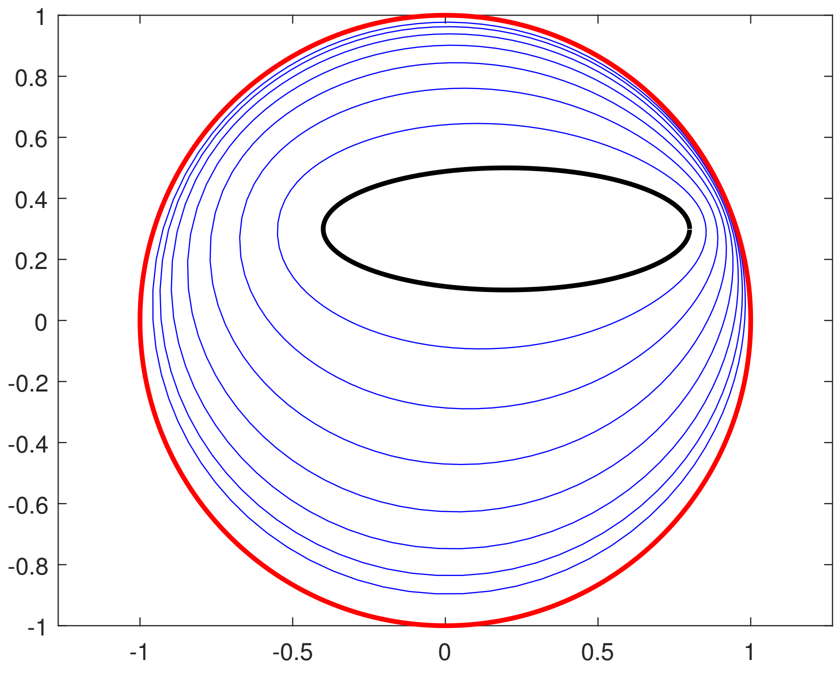
boundary search direction $\Delta\partial\Omega = -fn$ and fixed step size $s = 0.2$.

```
g_ = @(t) [0.2+0.3i] + 0.6*cos(t) + 0.2*1i*sin(t);
t = linspace(0,2*pi);
g = chebfun(@(t) g_(t), [0, 2*pi], 'trig'); %Initial boundary
init = plot(g, 'k')
set(init,'LineWidth',2);

f_ = @(x,y) x.^2 + y.^2 - 1;
f = chebfun2(@(x,y) f_(x,y), [-5 5 -5 5]);
hold on

for k=1:7
    dg = diff(g); n_ = -1i*dg; n = n_./abs(n_); %Unit normal to boundary
    fn = n.*f(real(g), imag(g)); %Boundary search direction
    g = g - 0.2*fn; %Updated boundary
    plot(g, 'b')
end

exact = fimplicit(f_, 'r');
set(exact,'LineWidth',2);
axis equal
```



The above figure shows the first 7 iterations $\partial\Omega^{(k)}$ in blue, together with the initial shape boundary $\partial\Omega^{(0)}$ in black and the optimum shape boundary $\partial\Omega^{*}$ in red.

3.3 Line Search for Shapes

In the previous example we fixed the step size s, but we could try to implement a line search-like method to reduce the number of iterations. In an ideal world, we could use an ELS-like method: choosing s such that $\mathcal{J}$ is minimised over all shapes in the set $\{\Omega + t\Delta\Omega : t \geqslant 0\}$, i.e

$$s = \arg\min_{t \geqslant 0} \mathcal{J}(\Omega + t\Delta\Omega).$$

Here we write $\Omega + t\Delta\Omega$ to mean the shape with boundary $\partial\Omega + t\Delta\partial\Omega$.

However, recall that in Section 1.3 it was difficult to find the step size exactly. We will consider a BLS-like algorithm to find s, and observe the improvements (if any) to our shape optimisation method. In a similar style to Section 1.3, we present the following backtracking algorithm for shapes:

Algorithm 4: Backtracking Line Search for Shapes

```
1 Given boundary descent direction $\Delta\partial\Omega$, $\alpha \in (0, 0.5), \beta \in (0, 1)$
2 Set $s \leftarrow 1$
3 while $\mathcal{J}(\Omega + s\Delta\Omega) > \mathcal{J}(\Omega) + \alpha s\, \mathrm{d}\mathcal{J}(\Omega, \Delta\partial\Omega)$ do
4     Set $s \leftarrow \beta s$
5 end
```

We can compute the functionals $\mathcal{J}(\Omega)$, $\mathcal{J}(\Omega + s\Delta\Omega)$ and shape derivatives $\mathrm{d}\mathcal{J}(\Omega, \mathcal{V})$ with *Chebfun* commands `integral2` and `integral` respectively. We could then replace the `for` loop in Example 3.2 with a `while` loop involving BLS for Shapes. Below is an example of the modified method with $\alpha = 0.1$ and $\beta = 0.7$:

```
J = @(f,g) integral2(f,g); %Cost functional, J(\Omega)
dJ = @(f,g,V) integral(f.*dot(V,n),g); %Shape derivative, dJ(\Omega,V)

hold on
tol = 0.05;
iteration = 0;

while abs(dJ(f,g,fn)) ≥ tol

    iteration  = iteration + 1

    dg = diff(g); n_ = -1i*dg; n = n_./abs(n_); %Unit normal to boundary
    fn = n.*f(real(g), imag(g)); %Boundary search direction

    % BLS for shapes; step size algorithm
        alpha = 0.1; beta = 0.7; s = 1;
        while J(f,g-s*fn) > J(f,g) + alpha*s*dJ(f,g,-fn);
                s = beta*s;
        end

    step = s

    g = g - s*fn; %Updated boundary
    plot(g, 'b')

end
```

Note that $\mathcal{J}$ and $\mathrm{d}\mathcal{J}$ are computed using the shape boundary instead of the shape itself. We use `iteration` and `step` to track how many iterations the algorithm takes, and what step size is used at each iteration.

3.4 Steepest Descent for Shapes

The MATLAB code in Section 3.3 bears close resemblance to the *Steepest Descent* algorithm we saw in Section 1.4. We could formally write the corresponding algorithm for shapes as follows:

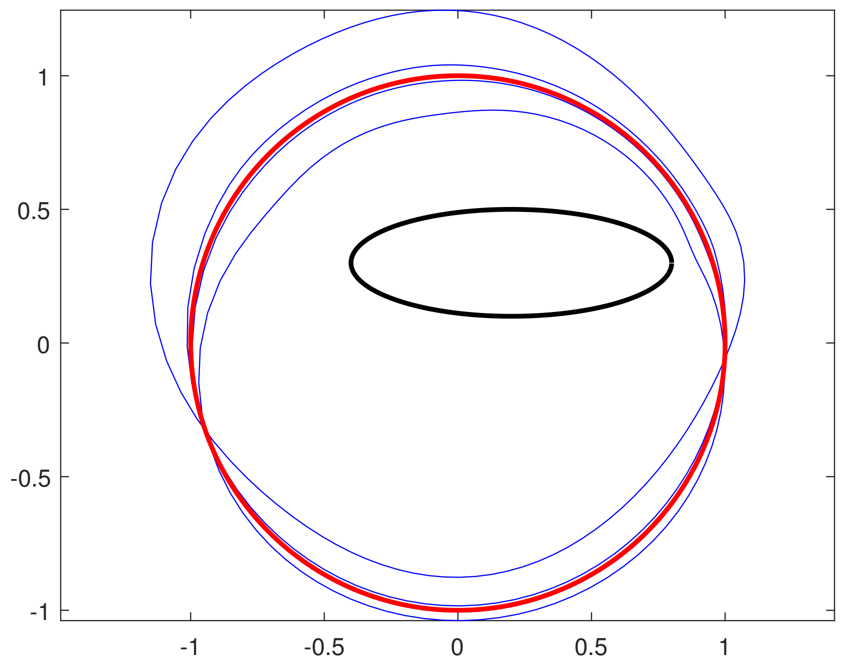
```
Algorithm 5: Steepest Descent for Shapes
1 Given starting shape $\Omega$
2 repeat
3     (1) Set $\Delta\partial\Omega \leftarrow -fn$
4     (2) Choose $s$ via BLS for Shapes
5     (3) Set $\partial\Omega \leftarrow \partial\Omega + s\Delta\partial\Omega$
6 until stopping criterion satisfied
```

We could use a stopping criterion of the form $\int_{\partial\Omega} f^2 \, \mathrm{d}S < \varepsilon$ here, which was used in the MATLAB code with $\varepsilon = 0.05$. The next example implements *Steepest Descent for Shapes* with backtracking.

Example 3.3 (Steepest Descent with BLS for Shapes): Running the code with the level set function

$$f(x) = x_1^2 + x_2^2 - 1$$

outputs `iteration` = 4 with step sizes 1, 0.7, 0.7, 0.7 respectively, as well as the following figure:



Though the first iteration shoots outside the level set, we achieve an extremely good approximation when the algorithm terminates after only 4 iterations. The first iteration uses step size 1; if we chose the initial step size for BLS to be such that the iteration stays within the level set, say $s = 0.5$, the algorithm terminates in just a fraction of the time.

After comparing the two descent methods in Examples 3.2 and 3.3 using `tic` and `toc`, we find that the non-BLS method takes 0.635 seconds, compared to 19.108 seconds with BLS[‡]. Though backtracking helps to reduce the number of iterations required, the effort of computing $\mathcal{J}$ and $\mathrm{d}\mathcal{J}$ makes Steepest Descent a lot slower, and for our purposes, impractical. We will revert to using a (small) fixed step size in our method, and we will not pursue a *Newton's Method for Shapes*.

[‡]Computed on a Toshiba Satellite P50-C-18L with Intel Core i7.

4 Complications in Shape Optimisation Methods

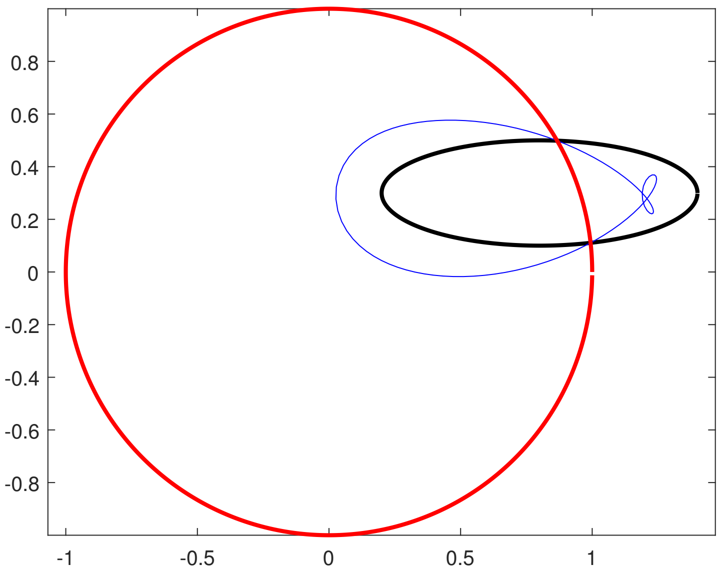
Recall that the optimum shape boundary is the set of points such that f vanishes. If the initial shape boundary intersects the optimum boundary, then at the intersection points we have

$$\Delta\partial\Omega = -fn = 0,$$

so each subsequent boundary iteration will always include those points as $\partial\Omega^{(k+1)} = \partial\Omega^{(k)}$ there. This can be problematic - suppose we instead took the initial boundary in Example 3.2 to be

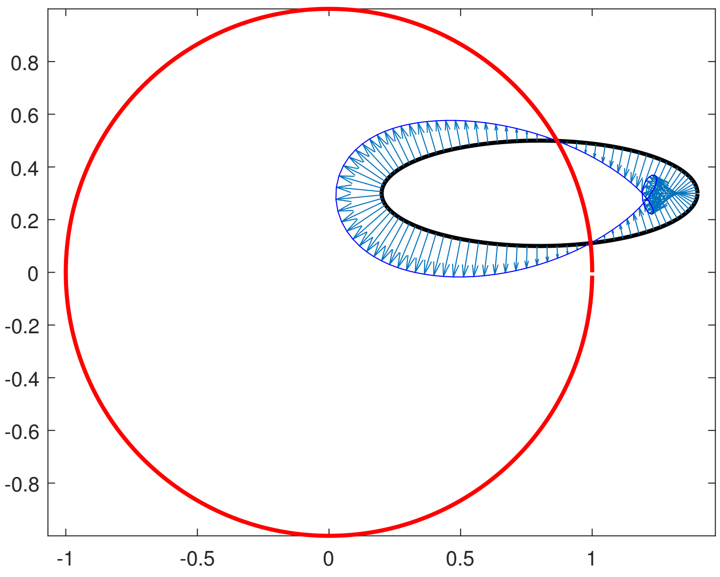
$$\Omega^{(0)} := \left\{ x : \frac{(x_1 - 0.8)^2}{0.6^2} + \frac{(x_2 - 0.3)^2}{0.2^2} \leqslant 1 \right\},$$

which intersects the optimum boundary at two points. The first iteration would look like this:



A complication arises in that the curve is no longer simple. Why does this happen? We can visualise the direction that points of the initial boundary are being transformed to using `quiver`:

```
quiver(real(g(t)), imag(g(t)), −0.2*real(fn(t)), −0.2*imag(fn(t)), 0)
```



In this example, points outside the zero level set that are further away from the optimum boundary have a greater value of f, so they are being transformed towards the level set with a greater scale factor than closer points. How can we choose $\partial\Omega^{(0)}$ to avoid this scenario?

4.1 Choosing the Initial Boundary

At the cost of some computational time, we can avoid this boundary-crossing behaviour by defining the initial boundary to be a small ball around a point at which f is minimised, i.e

$$\Omega^{(0)} := \{x : |x - x^*| \leqslant R\}, \quad x^* \in \arg\min_{x\in\mathbb{R}^2} f(x)$$

for some chosen radius $R > 0$. We can find such a point in MATLAB either with a descent method (as in Section 1), or using *Chebfun*'s `min2` command. The following code takes $R = 0.1$:

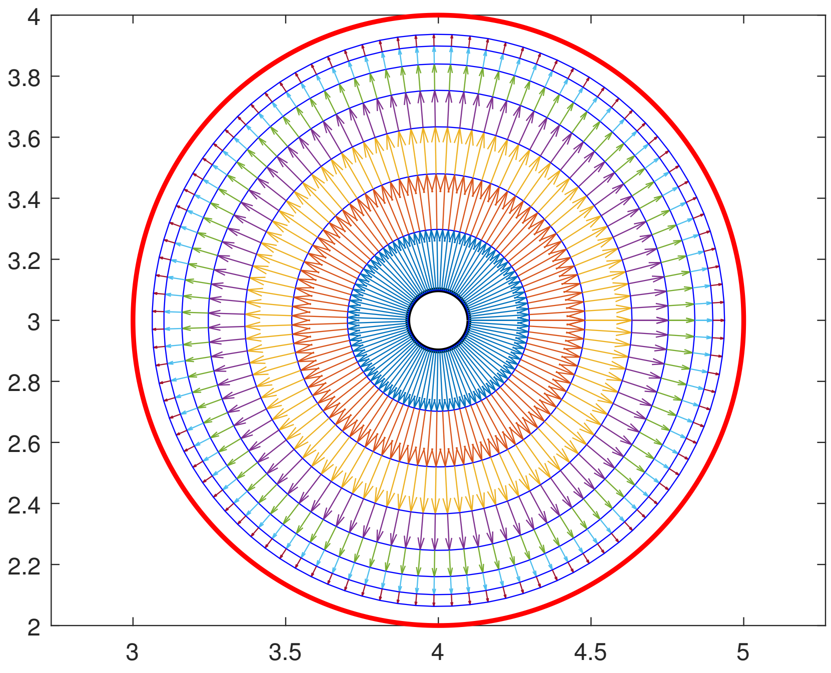
```
[Y,X] = min2(f);
x1 = X(1); x2 = X(2);
r = sqrt(x1.^2 + x2.^2);
the = atan2(x2,x1);
R = 0.1;
g_ = @(t) r*(cos(the) + 1i*sin(the)) + R*(cos(t) + 1i*sin(t));
```

Note that in the above, `r*(cos(the)+1i*sin(the))` is a scalar translating the parametrised ball `R*(cos(t)+1i*sin(t))`, and is not to be confused with the parametrisation itself. We test our new initial boundary technique in the following example.

Example 3.4 (New initial boundary): Let $f : \mathbb{R}^2 \to \mathbb{R}$ be a translated unit disk defined by

$$f(x) = (x_1 - 4)^2 + (x_2 - 3)^2 - 1.$$

The figure below plots the first 7 iterations of the shape optimisation method, with step size $s = 0.2$.

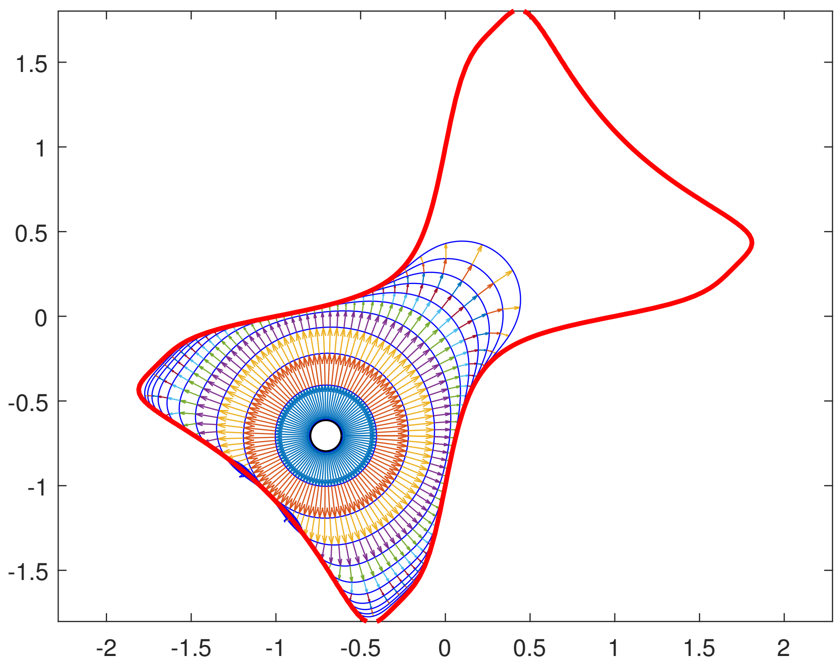


It is evident that the new method has detected the minimum point of f and translated the initial boundary accordingly. Using `pause` in the `for` loop allows us to check the speed of each iteration one at a time. In this case, the iterations are completed almost instantly.

4.2 The Bow-Tie Test Case

To test the success of our improved shape optimisation method, we will run it on a non-convex, bow-tie shaped level set function. The figure below shows 10 iterations with a step size $s = 0.1$ for

$$f(x) = (x_1 - x_2)^2 + (2x_1x_2 - 1)^4 - 5/2. \tag{6}$$

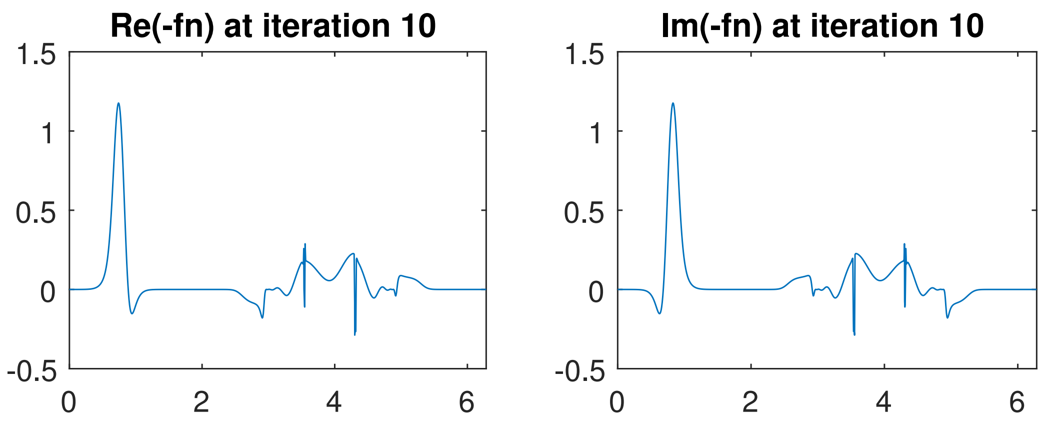


Using `pause`, we observe that our optimisation method slows down slightly with each iteration. The 10th iteration is incredibly slow, taking over 10 seconds to complete, and we get the message

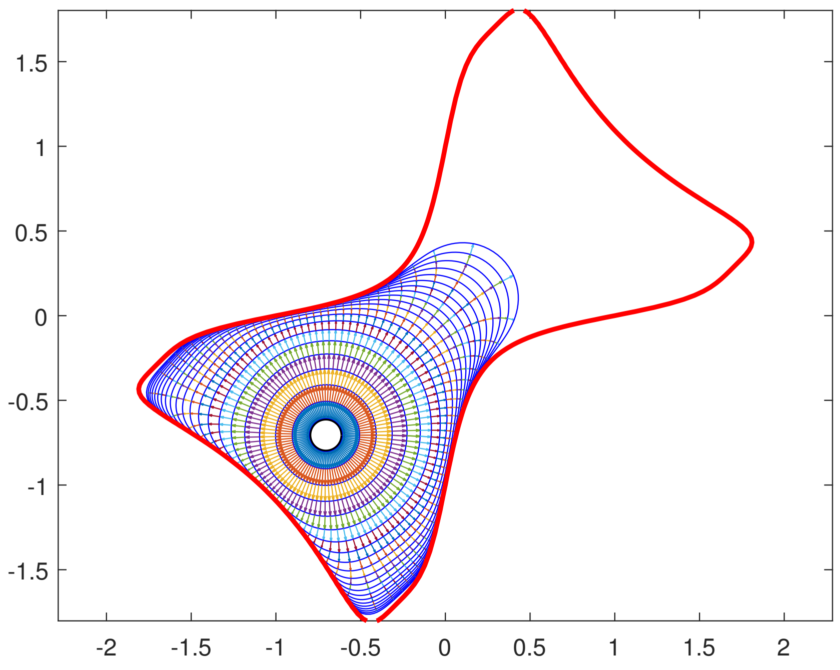
```
Warning: Composition with @(x)feval(op,feval(f,x)) failed to converge with 65536 points.
```

In an ideal optimisation method, we would expect the descent direction to vanish at parts of the boundary that have been optimised, and be approximately zero for intervals of t such that $g(t)$ is close to $\partial\Omega^*$. We would only expect spikes in $\mathrm{Re}(-fn)$ and $\mathrm{Im}(-fn)$ away from these intervals.

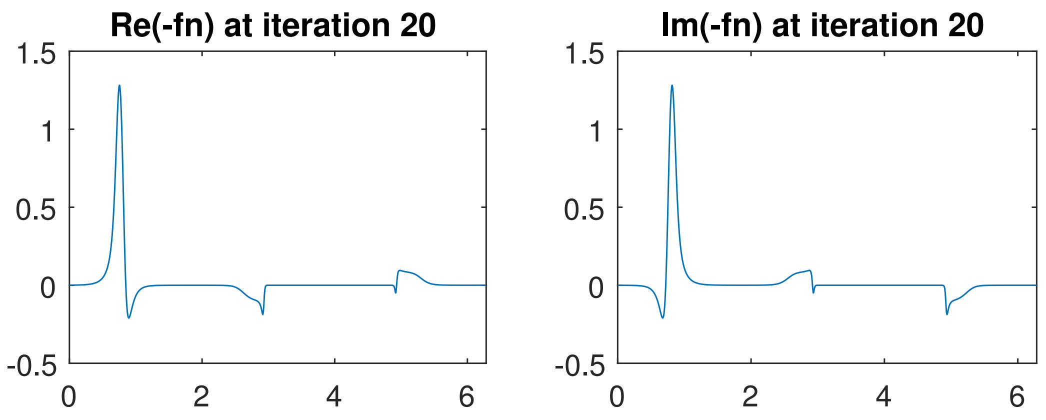
For the bow-tie, only the top end hasn't been optimised by the 10th iteration. The approximate interval of this area is $t \in [0, \frac{\pi}{2}]$, so we expect $-fn(t)$ to vanish for $t \notin [0, \frac{\pi}{2}]$. Sketching the real and imaginary parts of $-fn$ at the 10th iteration gives the following figures:



We observe that there are unexpected spikes in the interval $[\frac{\pi}{2}, 2\pi]$. Upon closer inspection, we find that the boundary is intersecting the zero level set at the bottom end of the bow-tie. Manually decreasing the step size prevents this:



The figure above shows 20 iterations for a halved step size $s = 0.05$. It means that the updated boundary stays entirely inside the level set, and the spikes we saw previously disappear:



The two intervals where fn remains non-zero in $t \notin [0, \frac{\pi}{2}]$ are the two bottom ends of the bow-tie that are not quite optimised, and the magnitudes of these spikes are decreasing with each iteration.

This is an improvement, but we are now performing double the amount of iterations and each one gets noticeably slower. How can we improve the speed of the algorithm?

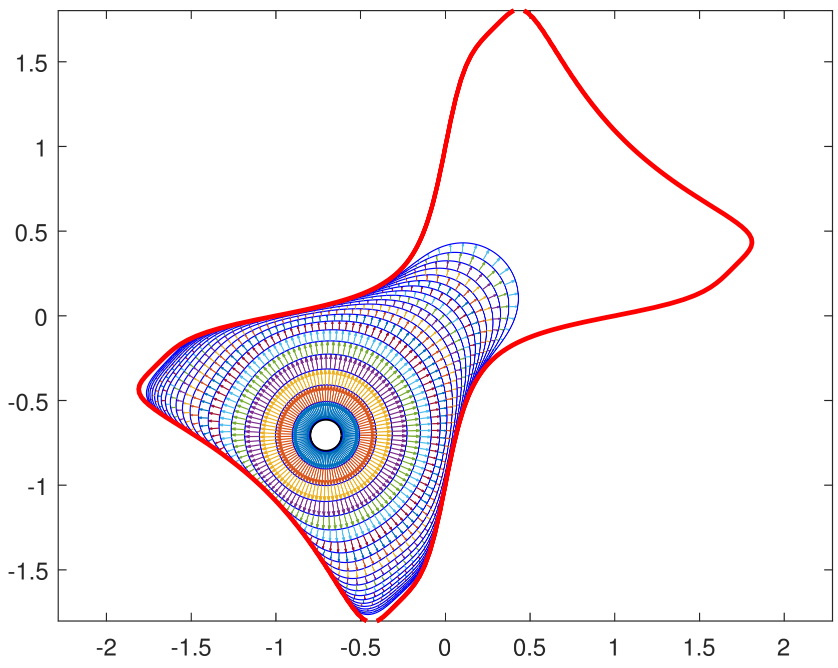
4.3 Reparametrising the Boundary

Notice that in the bow-tie figure above, the speed around the curve as t increases is lower at the bottom end and higher at the top end where the shape is not optimised - we can see this because the `quiver` arrows are spread out at the top, yet they are defined to be evenly spaced over $[0, 2\pi]$.

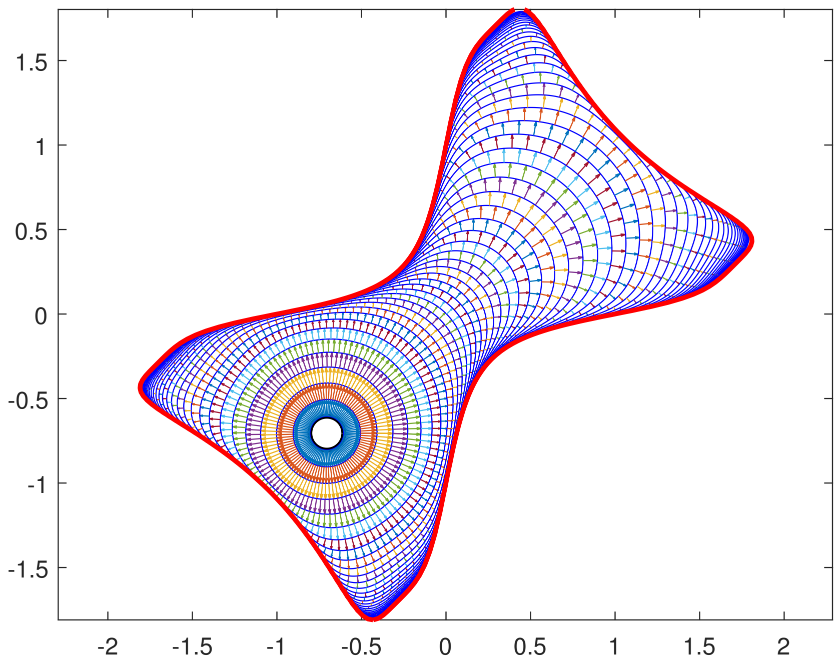
If we reparametrise the boundary to have a smaller, more uniform speed, we will increase the speed around areas of the boundary which have already been optimised, and we hypothesise that this will improve the efficiency our algorithm. Reparametrising in terms of arc length [7, Appendix E] achieves this; we can apply such a reparametrisation in MATLAB with the following code:

```
T = chebfun(@(t) t, [0 2*pi]); I = sum(abs(diff(g)),0,T); %Arc length at t=T
F = inv(I); scaling = T*I(2*pi)/2/pi; g2 = g(F(scaling)); %Reparametrised boundary
g = g2;
```

Putting this in the `for` loop, we can quickly check that this improves efficiency in the bow-tie test case. We can also see that the arrows are evenly spread around the curve:



The larger iterations are completed in a moderate time, and we can continue iterating until the entire shape is optimised. At the 50th iteration, just under a minute after the algorithm started, we retrieve the whole shape to extreme precision (real and imaginary parts of $-fn$ do not exceed 0.09 in magnitude).



The shape optimised Bow-Tie.

Conclusion

We began with a brief recap of unconstrained optimisation algorithms (namely Steepest Descent and Newton's Method), including exact and backtracking line search methods. We saw that Newton's Method converged in fewer iterations than Steepest Descent, but the cost of computing Δx_{nt} slowed the method down.

The notion of a shape derivative was then introduced and we expressed it as an integral around the shape boundary $\partial\Omega$, which was useful in the context of shape optimisation to find a search direction. To find a step size, we used our knowledge of unconstrained optimisation to create a similar backtracking algorithm for shapes. However, when implemented in MATLAB, we found that the time taken to compute $\mathcal{J}$ and $\mathrm{d}\mathcal{J}$ was far too long and rendered these algorithms impractical. A fixed step size was used thereafter.

We tested our shape optimisation method on different level set functions: firstly on the unit disk, then on a bow-tie shaped level set function (6). We found that complications occurred when the initial boundary intersected the optimum shape boundary, so we defined the initial boundary to be a small ball around a point that minimised f. Furthermore, we found that if our step size was too large - such that some iteration ends up intersecting the optimum boundary - our optimisation method slowed down (see Example 3.3 and the bow-tie test case).

Finally we found that reparametrising the boundary in terms of arc length made our optimisation method more efficient, despite the added time taken to compute the reparametrisation at each iteration.

Where does this report lead?

We could ask ourselves further questions about the efficiency of our shape optimisation method. Is there a better way to choose the initial boundary? For example, in the bow-tie test case there is symmetry at the origin, so we could have chosen to put the initial boundary there. Could we make a cost-effective backtracking method? What if our functional was subject to constraints? Often we find that the functional being solved depends on the solution of a PDE constraint [6]. How would we take our shape optimisation method into three dimensions?

There are many real-world applications of shape optimisation. For example, shape optimisation can be used to find the shape of an airplane wing which minimises drag, or even to find the shape of mechanical structures, such as cantilevers, which can resist a given stress whilst having a minimal mass or volume [6]. The hope is that this report can help to improve current shape optimisation methods that are used in practice.

References

[1] Stephen Boyd and Lieven Vandenberghe. *Convex Optimisation.* Cambridge University Press, 2004. `web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf`

[2] Knut Sydsæter and Peter Hammond. *Essential Mathematics for Economic Analysis.* Prentice Hall, 1995.

[3] Alberto Paganini. *Numerical Shape Optimization with Finite Elements.* Doctoral Thesis, ETH Zurich, 2016.

[4] Jan R Magnus and Heinz Neudecker. *Matrix Differential Calculus with Applications in Statistics and Econometrics.* Wiley, 1999.

[5] Tobin Driscoll, Nicholas Hale and Lloyd Trefethen. *Chebfun Guide.* `chebfun.org/docs/guide/chebfun_guide.pdf`

[6] *Shape Optimization.* Wikipedia. `en.wikipedia.org/wiki/Shape_optimization`

[7] Richard Palais and Robert Palais. *Differential Equations, Mechanics, and Computation.* The American Mathematical Society, 2009.