

Chapter 9

OPTIMIZATION AND THE COST FUNCTIONAL

9.1 Introduction

Chapter 1 introduced the problem of finding parameter values producing a flow solution that best matched a given set of flow data along a profile line. Subsequent chapters have developed the tools needed to set up parameters, define the problem, and determine the corresponding flow pattern.

Now that we can produce flow solutions, we want a way to assign a *score* that evaluates how well that solution satisfies our requirements. The measurement we will use will be defined in a *cost functional*. Once the cost functional is specified, our problem is ready for automatic treatment by an *optimization algorithm*, which determines how to go about choosing particular sets of parameters to test by evaluating the cost functional.

We have seen how the flow parameters β determine the particular problem to be solved, and how, at least locally, the resulting flow solution (u, v, p) can be regarded as a function of these parameters. We will see that the cost functional, initially defined in terms of the state variables, can also be regarded as a function of the underlying parameters.

The optimization algorithm will generally require not only the evaluation of the cost functional for a given set of parameters, but also the partial derivatives of the cost functional with respect to those parameters. We will investigate how to make this calculation, given that we don't have the necessary explicit information.

In anticipation of problems that will arise during computation, we discuss some modifications that can be made to the cost functional, involving *penalty functions*, in pursuit of *regularization*. Reasons for modifying the cost functional in this way include the existence of local minima or a functional whose contour levels are very "twisted" or badly scaled.

9.2 Local Minimizers in Computational Optimization

A *local minimizer* of a functional is a point which has a functional value that is less than or equal to the functional value of every other point in some sufficiently small neighborhood. A local minimizer may be contrasted with a *global minimizer*, whose functional value is less than or equal to the functional value of *all* other points. While a global minimizer is obviously more desirable than a local minimizer, most practical optimization algorithms are only able to locate local minimizers.

Some methods for checking whether a point is actually a true local minimizer are not suitable for a practical, finite computation: we might wish to have a perfectly true graph of the functional, in which case a single glance would tell us everything we need to know; we might imagine evaluating the functional at every point in a neighborhood of the candidate; we might, if we had an explicit formula for the functional in terms of the parameters, perform some simple analysis on the gradient.

The optimization code has no such luxuries. Its analysis of the problem is based entirely on sampling the cost functional and its gradient (which may itself be inaccurate!) at a finite

number of points. The Hessian matrix is also useful in this regard, and an optimization code may desire values of this matrix from the user, or, as in our case, attempt to approximate the matrix based on functional and gradient information. The following theorems suggest how this information may be used in the search for local minimizers (Dennis and Schnabel [10]):

Theorem 9.1 (The Gradient of a Local Minimizer) *Suppose that $\mathcal{J} : R^n \rightarrow R$ is continuously differentiable in an open neighborhood of β_0 . If β_0 is a local minimizer of $\mathcal{J}(\beta)$, the gradient vector $\nabla \mathcal{J}(\beta)$ must vanish at β_0 .*

Theorem 9.2 (The Hessian of a Local Minimizer) *Suppose that $\mathcal{J} : R^n \rightarrow R$ is twice continuously differentiable in an open convex neighborhood of β_0 . If β_0 is a local minimizer of $\mathcal{J}(\beta)$, and if the Hessian matrix $\nabla^2 \mathcal{J}(\beta)$ is Lipschitz continuous at β_0 , then the Hessian matrix must be positive semidefinite at β_0 .*

If we strengthen the condition on $\nabla^2 \mathcal{J}(\beta)$, we can turn these two necessary conditions on a minimizer into a sufficient condition for local minimization:

Theorem 9.3 (A Test for Local Minimization) *Suppose that $\mathcal{J} : R^n \rightarrow R$ is twice continuously differentiable in an open convex neighborhood of β_0 , that the gradient vector $\nabla \mathcal{J}(\beta)$ vanishes at β_0 , and that the Hessian matrix $\nabla^2 \mathcal{J}(\beta)$ is Lipschitz continuous and nonsingular at β_0 .*

Then β_0 is a local minimizer of \mathcal{J} if, and only if, $\nabla^2 \mathcal{J}(\beta_0)$ is positive definite.

These conditions are *local*; if we are allowed to assume the needed level of continuity, we may even assert that the conditions are *pointwise*. That is, to verify that a point is a local

minimizer, we generally need only check the value of the first two derivatives there. Hence, these conditions are computationally feasible to check.

The optimization code uses derivative information to seek to move in a “downhill” or “descent” direction, and assumes convergence when the partial derivatives are sufficiently close to zero. As the optimization proceeds, the gradient and the estimated Hessian of each new candidate are examined, and convergence to a (local) minimizer will be declared based primarily on that evidence.

As we have said, we really want a *global minimizer* from the optimization code. By construction, in most of our test problems we will know the solution beforehand. We denote the value of the target parameters by $(\lambda^T, \alpha^T, Re^T)$, and the corresponding state variables by (u^T, v^T, p^T) . The form of the cost functional then guarantees that at the target solution, the globally minimizing cost is 0. Yet we will see cases where the optimization code returns a local minimizer which is significantly far from the target solution, and with a higher value of the functional.

Supposing that there are local minimizers of the cost functional, aside from the global minimizer, it may actually be quite easy for the optimization code to return such a point. All that is required is that, during the course of the optimization, a point is computed which is close enough to the local minimizer. In that case, the gradient vector is likely to draw the optimization code even closer. A local optimizer generally lies in a “valley of attraction”, that is, a small depression in the graph of the functional. Most optimization codes are very reluctant to consider points with higher functional values than their most recently accepted point. Thus, once the optimization code has fallen into the valley of attraction of a local minimizer, it is unable to climb back out, and it will converge to the local minimizer contained in the valley.

9.3 The Optimization Algorithm

Our search for the bump shape, inflow conditions, and Reynolds number which produce the closest match to a given set of flow data has been rephrased to become the search for the set of parameters β which produce the lowest value of some functional $\mathcal{J}(\beta)$. Before we consider the details of the specification of the particular functionals that we will use, we wish to make some general remarks about optimization, and the particular optimization algorithm we will employ.

Although we are trying to minimize the cost functional $\mathcal{J}(\beta)$, a great deal of useful information may be gleaned from the gradient vector $\nabla\mathcal{J}(\beta)$. One reason for this is that the negative of the gradient vector points in the direction of steepest descent of $\mathcal{J}(\beta)$. Moreover, Theorem 9.1 tells us that a necessary condition for a minimizer is that the gradient function vanish there.

If we were able to express the dependence of \mathcal{J} on β explicitly, say, in a formula, then we might seek candidates for a minimizer by computing formulas for the partial derivatives of \mathcal{J} with respect to each component β_i , and trying to solve for those sets of parameters β where all the partial derivatives vanish:

$$\frac{\partial\mathcal{J}(\beta)}{\partial\beta_i} = 0. \tag{9.1}$$

However, for the discrete problem, we have no formula for \mathcal{J} , and our only understanding of its properties comes about by explicitly evaluating \mathcal{J} at a sequence of points. Therefore, the optimization is most naturally an iterative process. On the other hand, in this iterative process, we will still seek points where the gradient vanishes. The particular algorithm that we employ is a version of the *model/trust region* method.

The iterative process that we will carry out involves constructing a local model of the behavior

of \mathcal{J} , which we expect is valid over some small “trust region” centered at the most recently accepted point. We then predict the location of a point which lies within the trust region, and at which \mathcal{J} is expected to have a lower value. This prediction is based in part on the gradient direction, but uses other information as well. We then check the actual value of \mathcal{J} at the predicted point. If the value is lower, as expected, then we may accept this point as the next iterate, and construct a new trust region around it. Otherwise, we retain the old iterate, shrink the trust region and try again.

In this process, gradient information is extremely valuable; not only does it tell us *when* we have reached a candidate minimizer, but it also tells us *where* to look. This is because the negative of the gradient vector points in the direction of maximum decrease of the functional. Moreover, a gradient give us information about the behavior of \mathcal{J} in the entire neighborhood of a point. Many optimization algorithms require gradients of the cost function. We will not be able to supply the exact gradient, but we will be able to approximate it using discretized sensitivities or finite difference sensitivities.

At any time, the algorithm may decide to stop because it believes that the minimizer has been found. Such a decision will be based on the estimated value of the gradient, on the values of the functional at the previous points, and on the relative change in the position of the estimated location of the minimizer. The algorithm may also decide to stop because it suspects an error in the data, such as an inconsistency between the gradient data and the cost functional values. Otherwise, the algorithm will proceed to a new step, continuing to try to reduce the value of the functional.

This is as much as we need to know right now about the design and behavior of the optimization code. Further details about the algorithm are reported in Chapter 17.

9.4 Defining a Cost Functional

Our class of flow problems completely specified the shape of the region except for the bump, the boundary conditions except for the inflow, and the state equations except for the value of the Reynolds number. The quantities left unspecified, which we called the “flow parameters”, may then be freely varied to produce a variety of flow problems. We assume that any set of parameter values (λ, α, Re) represents a well-defined flow problem. For convenience, we will speak as though there exists uniquely a corresponding continuous flow solution (u, v, p) , although we know that the situation is actually more complicated. If we represent the parameters by β , we may emphasize the dependence of the flow solution on the parameters by writing, for instance $u(\beta)$ or $u(\lambda, \alpha, Re)$.

The desired behavior that we are seeking to optimize is for the flow solution to come as close as possible to some given flow data along a single vertical profile line in the flow region. Specifically, let us suppose that along the line $x = x_s$ we are given the following functions, or *target data*, $u^T(x_s, y)$, $v^T(x_s, y)$ and $p^T(x_s, y)$, for $0 \leq y \leq 3$. It is possible that these functions are utterly arbitrary, in which case we might expect poor approximation; it is possible that the profile data was generated as the flow solution of a discrete problem, but that this flow solution does not lie in the feasible space, so that we can, at best approximate its behavior along the profile line; finally, these functions may in fact be the profiles of an actual flow solution of a discrete problem from the feasible parameter space, generated by a set of parameters which may be designate as β^T . Only in this last case is it permissible to write $u^T(x, y) = u(x, y, \beta^T)$.

In the latter case, we might be tempted to assume that an alternate form of the optimization problem would be to recover the generating parameters β^T . We will see that this is *not* an equivalent problem. Many flows, corresponding to parameter values quite distinct

from β^T , might come close to the desired behavior, and be locally optimal solutions, while corresponding to a set of parameters quite distinct from the target set. On the other hand, simply by the assumed continuity of the flow solution $u(x, y, \beta)$ as a function of β , it is correct to expect that if an optimization produces an answer very close to β^T , that the flow profiles will be close to the target profiles, and the desired behavior nearly achieved.

Now to measure how well a given flow solution approximates the target data, we want to define a *cost functional*, which we may write as $J(u, v, p, \beta)$. The cost functional should compute, in some reasonable way, the closeness between the desired and achieved behaviors. Let us suppose that in our case, we are only interested in the discrepancy for the horizontal velocity component, between the target data and the computed flow solution. A natural choice of cost functional for the continuous problem would then be:

$$J(u, v, p, \beta) = \int_{x=x_s} (u(x, y) - u^T(x_s, y))^2 dy. \quad (9.2)$$

Since we're actually only going to be able to solve discrete flow problems, it is necessary for us to make the obvious transformations so that we can produce a corresponding cost functional for the discrete problem, which we write $J^h(u^h, v^h, p^h, \beta)$. Since the discrete flow solution actually produces functions defined over the whole region, the cost functional for the discrete problem will formally look identical to that for the continuous case:

$$J^h(u^h, v^h, p^h, \beta) = \int_{x=x_s} (u^h(x_s, y) - u^T(x_s, y))^2 dy. \quad (9.3)$$

While this is the basic form for the cost functional, note that we haven't specified the actual value x_s yet. We will discuss this matter shortly. We will also reserve the right to vary the form of this cost functional by adding new terms, in our case to control undesirable oscillations in the solution.

9.5 The Role of the Cost Functional in Optimization

The cost functional allows us to specify with a single number the closeness of any given flow profile to the desired target profile. That makes it simple to define the problem: we want the “best” flow solution, one for which there’s no flow solution with a lower cost functional. We also may have to settle for a “locally best” flow solution, for which there is no nearby flow solution with a lower cost functional.

Once the cost functional is specified, we can pose the *optimization problem for the parameterized continuous flow equations*:

Suppose that, for every choice of values of the parameters β there corresponds a continuous Navier Stokes flow problem with a solution $(u(\beta), v(\beta), p(\beta))$, which we will call a feasible flow. Suppose that a cost functional $J(u, v, p, \beta)$ is defined for flow functions (u, v, p) and parameters β . The **continuous optimization problem** seeks the parameter values β^* and the corresponding flow solution $(u(\beta^*), v(\beta^*), p(\beta^*))$ such that $J(u(\beta^*), v(\beta^*), p(\beta^*), \beta^*)$ is minimized over all feasible flows $(u(\beta), v(\beta), p(\beta))$.

In analogy, we also pose the *optimization problem for the discrete parameterized flow equations*:

Suppose that, for every choice of values of the parameters β there corresponds a discrete Navier Stokes flow problem, and that this problem has a flow solution $(u^h(\beta), v^h(\beta), p^h(\beta))$, which we will call a (discrete) feasible flow. Suppose that a cost functional $J^h(u^h, v^h, p^h, \beta)$ is defined for all flow functions (u^h, v^h, p^h) and parameters β . The **discrete optimization problem** seeks the parameter values β^* and the corresponding flow solution $(u^h(\beta^*), v^h(\beta^*), p^h(\beta^*))$ such that

$J^h(u^h(\beta^*), v^h(\beta^*), p^h(\beta^*), \beta)$ is minimized over all feasible flows $(u^h(\beta), v^h(\beta), p^h(\beta))$.

9.6 The Cost Functional as a Function of the Problem Parameters

We will find it advantageous to be able to express the optimization problem without explicit mention of the flow variables (u, v, p) . It should be clear that we can do this, at least formally. If we can assume that the parameters β uniquely determine the flow solution, or that this is true at least in a local sense, as discussed in Chapter 3, then a function which depends on u and β can be replaced by an equivalent function depending only on β .

In particular, let us suppose that, instead of arbitrary flow variables (u, v, p) , we are only going to evaluate the cost functional at flow variables that correspond to the argument β , so that we would write:

$$J(u(\beta), v(\beta), p(\beta), \beta) = \int_{x=x_s} (u(x, y, \beta) - u^T(x_s, y, \beta))^2 dy. \quad (9.4)$$

Since knowledge of β determines $u(x, y, \beta)$, we really only need to know β to evaluate J in this case. We can simply eliminate the mediation of the state variables, and regard the cost functional as a direct function of the parameter, arriving at what we will call the *constrained cost functional*:

$$\mathcal{J}(\beta) \equiv J(u(\beta), v(\beta), p(\beta), \beta) \quad (9.5)$$

$$= \int_{x=x_s} (u(x, y, \beta) - u^T(x_s, y))^2 dy. \quad (9.6)$$

Note the distinction between the unconstrained functional J and the constrained functional \mathcal{J} : the cost functional J is defined for arbitrary functions u, v, p , but \mathcal{J} requires the specific functions $u(\beta), v(\beta)$ and $p(\beta)$ that satisfy the Navier Stokes problem defined by the

parameters β . In a natural way, we define the constrained discrete cost functional $\mathcal{J}^h(\beta)$ in terms of $J^h(u^h, v^h, p^h, \beta)$.

9.7 Computing the Gradient of the Cost Functional

The optimization code will require the gradient of the cost functional. This computation would be straightforward if $\mathcal{J}(\beta)$ were given via an explicit formula. But we have no formula for the cost functional in terms of β , and we must devise a way of approximating the gradients. For any particular parameter component, we can write out the relationship between an entry of $\nabla \mathcal{J}$ and the derivatives of the original cost functional J :

$$\begin{aligned}
\frac{\partial \mathcal{J}(\beta)}{\partial \beta_i} &= \frac{DJ(u(\beta), v(\beta), p(\beta), \beta)}{D\beta_i} \\
&= \frac{\partial J(u(\beta), v(\beta), p(\beta), \beta)}{\partial u} u_{\beta_i} \\
&+ \frac{\partial J(u(\beta), v(\beta), p(\beta), \beta)}{\partial v} v_{\beta_i} \\
&+ \frac{\partial J(u(\beta), v(\beta), p(\beta), \beta)}{\partial p} p_{\beta_i} \\
&+ \frac{\partial J(u(\beta), v(\beta), p(\beta), \beta)}{\partial \beta_i},
\end{aligned} \tag{9.7}$$

while, for the discrete problem, we write the corresponding formula in terms of derivatives of J^h with respect to the individual finite element coefficients:

$$\begin{aligned}
\frac{\partial \mathcal{J}^h(\beta)}{\partial \beta_i} &= \frac{DJ^h(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{D\beta_i} \\
&= \sum_k \frac{\partial J^h(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{\partial u_k^h} (u_k^h)_{\beta_i} \\
&+ \sum_k \frac{\partial J^h(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{\partial v_k^h} (v_k^h)_{\beta_i} \\
&+ \sum_k \frac{\partial J^h(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{\partial p_k^h} (p_k^h)_{\beta_i} \\
&+ \frac{\partial J^h(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{\partial \beta_i}.
\end{aligned} \tag{9.8}$$

These formulas require the values of the continuous or discrete sensitivities. These are unobtainable for the continuous case, and difficult to calculate in the discrete case. Therefore, we consider ways of estimating the discrete cost gradient, using approximations to the discrete sensitivities.

If we have computed the discretized sensitivities, for instance, we may try the approximation:

$$\begin{aligned}
\frac{\partial \mathcal{J}^h(\beta)}{\partial \beta_i} &\approx \left(\frac{\partial \mathcal{J}(\beta)}{\partial \beta_i} \right)^h \equiv \sum_k \frac{\partial J^h(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{\partial u_k^h} (u_{k,\beta_i})^h \\
&+ \sum_k \frac{\partial J^h(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{\partial v_k^h} (v_{k,\beta_i})^h \\
&+ \sum_k \frac{\partial J^h(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{\partial p_k^h} (p_{k,\beta_i})^h \\
&+ \frac{\partial J^h(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{\partial \beta_i}.
\end{aligned} \tag{9.9}$$

Here, we have had to write awkward quantities such as

$$(u_{k,\beta_i})^h \tag{9.10}$$

to stand for the k -th finite element coefficient of the discretized velocity sensitivity with respect to β_i , in other words, the k -th coefficient of $(u_{\beta_i})^h$.

If we have computed the finite coefficient differences, then we can instead use the approximation:

$$\begin{aligned}
\frac{\partial \mathcal{J}^h(\beta)}{\partial \beta_i} &\approx \sum_k \frac{\partial J^h(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{\partial u_k^h} \frac{\Delta u_k^h}{\Delta \beta_i} \\
&+ \sum_k \frac{\partial J^h(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{\partial v_k^h} \frac{\Delta v_k^h}{\Delta \beta_i} \\
&+ \sum_k \frac{\partial J^h(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{\partial p_k^h} \frac{\Delta p_k^h}{\Delta \beta_i} \\
&+ \frac{\partial J^h(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{\partial \beta_i}.
\end{aligned} \tag{9.11}$$

Finally, we could choose to approximate the gradient directly, using the formula:

$$\frac{d\mathcal{J}^h(\beta)}{d\beta_i} \approx \frac{\Delta\mathcal{J}^h(\beta)}{\Delta\beta_i} \equiv \frac{\mathcal{J}^h(\beta + \Delta\beta_i) - \mathcal{J}^h(\beta)}{\Delta\beta_i}. \quad (9.12)$$

Assuming we use either the discretized sensitivities or finite coefficient differences, it remains to compute the partial derivatives of J with respect to u_k , v_k , and p_k . This is usually not a problem, since J is generally presented as a simple, explicit formula in terms of these variables. The partial derivative with respect to β_i should also be easy to compute, as long as there are no secondary effects on the cost functional due to changes in the region's geometry. That might happen, for instance, if the profile line were placed in a region which was affected by changes to β . As it happens, the profile line will always be placed in the fixed portion of the region, so we will not have to concern ourselves with such effects.

Note that, if we use the finite coefficient difference approach, the computation of component i of the gradient will require that we carry out a full flow solve to find $u^h(\beta + \Delta\beta_i)$, $v^h(\beta + \Delta\beta_i)$ and $p^h(\beta + \Delta\beta_i)$. Moreover, we must choose a “reasonable” increment β_i that is small enough for an accurate approximation of the partial derivative while being large enough to avoid roundoff problems. Finally, as we have already noted, at least a few nodes *must* move if we are using a shape parameter; perhaps most of the nodes move. If we actually wish to compute sensitivities at a *fixed* spatial location, then the finite coefficient differences computed at moving nodes must be adjusted as described earlier to subtract off changes in the flow solution that are merely due to spatial displacement of the nodes.

If we use the discretized sensitivity approach, then we will generally just have finished a Newton iteration for the flow solution, and so will have at hand exactly the matrix we need for the sensitivity system, already assembled and factored. Therefore, the only cost will be in the assembly of the right hand sides for each sensitivity, and the solution of the already factored linear system. We will see that, in general, this approach can be actually

cheap compared to the finite coefficient difference method. Instead of $NPar$ extra flow solutions, we simply have $NPar$ linear systems to solve. The main drawback to using the discretized sensitivities for geometric parameters is their inherent error. Once we have chosen a particular finite element mesh, there is an automatic discrepancy between the discretized and true sensitivities; the only sure cure for this is to refine the finite element mesh, a costly step.

9.8 The Formula for the Hessian

Although we will not do so here, we wish to note that, if we had approximated the *second order sensitivities* using discretized sensitivities or finite coefficient differences, the Hessian matrix could also be supplied to the optimization algorithm. We exhibit the appropriate formula for the continuous case, although for simplicity we suppress the terms involving differentiation with respect to the variables v and p :

$$\begin{aligned}
\frac{\partial^2 \mathcal{J}(\beta)}{\partial \beta_i \partial \beta_j} &= \frac{D^2 J(u(\beta), v(\beta), p(\beta), \beta)}{D\beta_i D\beta_j} \\
&= \frac{\partial^2 J(u(\beta), v(\beta), p(\beta), \beta)}{\partial u^2(\beta)} \frac{\partial u(\beta)}{\partial \beta_i} \frac{\partial u(\beta)}{\partial \beta_j} \\
&+ \frac{\partial^2 J(u(\beta), v(\beta), p(\beta), \beta)}{\partial u \partial \beta_j} \frac{\partial u(\beta)}{\partial \beta_i} \\
&+ \frac{\partial^2 J(u(\beta), v(\beta), p(\beta), \beta)}{\partial u \partial \beta_i} \frac{\partial u(\beta)}{\partial \beta_j} \\
&+ \frac{\partial J(u(\beta), v(\beta), p(\beta), \beta)}{\partial u} \frac{\partial^2 u(\beta)}{\partial \beta_i \partial \beta_j} \\
&+ \frac{\partial^2 J(u(\beta), v(\beta), p(\beta), \beta)}{\partial \beta_i \partial \beta_j}.
\end{aligned} \tag{9.13}$$

9.9 Optimization With Approximate Gradients

Whether we use the chain rule on discretized sensitivities or finite coefficient differences, or use finite cost functional differences directly, we will be producing an approximation of the cost gradient. To halve the approximation error in a first order finite difference quotient, we have only to halve the size of the perturbation, such as $\Delta\alpha$. A new estimate then costs us one more flow solve. But, if our discretized sensitivity error is of order $O(h)$, to halve the approximation error in a discretized sensitivity, we must halve h , which means we roughly quadruple the number of nodes and unknowns, making the work of factoring the Jacobian increase by a factor of more than 16. Thus there is a limit to how much we can reduce h simply to improve the gradient estimates.

The effect of using an inaccurate gradient depends greatly on which particular algorithm is being used. For a variation of the model trust region optimization method, Carter [8] showed that convergence to a minimizer would still be achieved as long as the approximate gradient $(\nabla\mathcal{J}(\beta))^h$ had a sufficiently positive projection on the correct gradient $\nabla\mathcal{J}^h(\beta)$. In such a case, the negative of the approximate gradient would still define a descent direction for the function to be minimized, though generally not a *steepest* descent direction. The condition given is equivalent to the requirement that there exist some fixed positive γ , so that for all parameter values β at which $(\nabla\mathcal{J}(\beta))^h$ does not vanish, it is true that:

$$0 < \gamma (\nabla\mathcal{J}(\beta))^h \cdot (\nabla\mathcal{J}(\beta))^h \leq (\nabla\mathcal{J}(\beta))^h \cdot \nabla\mathcal{J}^h(\beta). \quad (9.14)$$

While we don't have the exact gradients, we may carry out an approximate version of this check on the discretized sensitivity gradients, using finite difference gradient estimates.

9.10 Example: A Cost Functional Using Horizontal Velocities Only

As an example of a cost functional, let us consider a case where we are given the value of the function $u^T(x_s, y)$, with the profile line at $x_s = 3$:

$$J_2(u, v, p, \beta) = \int_{x_s=3} (u(x_s, y) - u^T(x_s, y))^2 dy. \quad (9.15)$$

which we may rewrite in terms of β as:

$$\mathcal{J}_2(\beta) = \int_{x_s=3} (u(x_s, y, \beta) - u^T(x_s, y))^2 dy. \quad (9.16)$$

Now, we would like to compute a partial derivative of the form $\frac{\partial J_2}{\partial u}$ but, as in the calculation of the *FP* operator in Chapter 3, we must use the method of variations in order that the computation makes sense. In other words, we use the fact that:

$$\frac{\partial J_2}{\partial u}(u_0, v_0, p_0, \beta_0) \tilde{u} = \lim_{\epsilon \rightarrow 0} \frac{J_2(u_0 + \epsilon \tilde{u}, v_0, p_0, \beta_0) - J_2(u_0, v_0, p_0, \beta_0)}{\epsilon} \quad (9.17)$$

to compute that

$$\begin{aligned} & \frac{\partial J_2}{\partial u}(u_0, v_0, p_0, \beta_0) \tilde{u} \\ &= \lim_{\epsilon \rightarrow 0} \frac{\int_{x_s=3} ((u + \epsilon \tilde{u})(x_s, y) - u^T(x_s, y))^2 dy - \int_{x_s=3} (u(x_s, y) - u^T(x_s, y))^2 dy}{\epsilon} \end{aligned} \quad (9.18)$$

$$= \lim_{\epsilon \rightarrow 0} \frac{\int_{x_s=3} ((u + \epsilon \tilde{u})(x_s, y) - u^T(x_s, y))^2 - (u(x_s, y) - u^T(x_s, y))^2 dy}{\epsilon} \quad (9.20)$$

$$= \lim_{\epsilon \rightarrow 0} \frac{\int_{x_s=3} 2\epsilon u(x_s, y) \tilde{u}(x_s, y) - 2\epsilon \tilde{u}(x_s, y) u^T(x_s, y) - \epsilon^2 \tilde{u}^2(x_s, y) dy}{\epsilon} \quad (9.21)$$

$$= 2 \int_{x_s=3} (u(x_s, y) - u^T(x_s, y)) \tilde{u} dy. \quad (9.22)$$

Now if we simply choose \tilde{u} to be $\frac{\partial u}{\partial \beta}$, we have

$$\frac{\partial \mathcal{J}_2(\beta)}{\partial \beta} = \frac{\partial J_2(u(\beta), v(\beta), p(\beta), \beta)}{\partial u} \frac{\partial u(\beta)}{\partial \beta} \quad (9.23)$$

$$= 2 \int_{x_s=3} (u(x_s, y, \beta) - u^T(x_s, y)) \frac{\partial u(\beta)}{\partial \beta} dy. \quad (9.24)$$

For the discrete problem, we can easily derive the corresponding formula:

$$\frac{\partial \mathcal{J}_2^h(\beta)}{\partial \beta} = \frac{\partial J_2(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{\partial u^h} \frac{\partial u^h}{\partial \beta} \quad (9.25)$$

$$= 2 \int_{x_s=3} (u^h(x_s, y, \beta) - u^T(x_s, y)) \frac{\partial u^h}{\partial \beta} dy \quad (9.26)$$

$$= 2 \int_{x_s=3} (u^h(x_s, y, \beta) - u^T(x_s, y)) u_\beta^h dy. \quad (9.27)$$

We can approximate this equation by using the discretized sensitivities, giving us the following computable approximation to the gradient:

$$\frac{\partial \mathcal{J}_2^h(\beta)}{\partial \beta} \approx \left(\frac{\partial \mathcal{J}_2(\beta)}{\partial \beta} \right)^h \quad (9.28)$$

$$\equiv \frac{\partial J_2(u^h(\beta), v^h(\beta), p^h(\beta), \beta)}{\partial u^h} (u_\beta)^h \quad (9.29)$$

$$= 2 \int_{x_s=3} (u^h(x_s, y, \beta) - u^T(x_s, y)) (u_\beta)^h dy. \quad (9.30)$$

When the discretized target flow u^T is actually a feasible flow, that is, $u^T(x, y) = u^h(x, y, \beta^T)$ for some set of parameters β^T , this formula implies that the approximate cost gradient computed using discretized sensitivities will always be identically zero at the global minimizer $u^h(x, y, \beta^T)$.

This is *not* because of some special property of discretized sensitivities, but rather because of the factor of $u^h(x_s, y, \beta) - u^T(x_s, y)$ in the cost gradient computation; an arbitrary approximation for u_β^h would satisfy the same condition. This means that while we should expect the approximated cost gradients to be zero at a feasible target, we should not take that as evidence of special accuracy!

Now we have available three approximations to derivative quantities like $\frac{\partial u^h}{\partial \beta}$, namely the discretized sensitivities and the unadjusted and adjusted finite coefficient differences. We

Table 9.1: Cost gradients via discretized sensitivities and finite differences.
The comparison is made at several sets of parameter values.

Gradient Component	Finite Coef Difference (FCD)	Adjusted FCD (AFCD)	Direct FD (DFD)	Discretized Sensitivity (DS)
$Re = 5.0$				
$\frac{\partial \mathcal{J}_2^h}{\partial \lambda}$	-0.86784	-0.86784	-0.86784	-0.86784
$\frac{\partial \mathcal{J}_2^h}{\partial \alpha}$	-0.031580	-0.031580	-0.031580	-0.034021
$\frac{\partial \mathcal{J}_2^h}{\partial Re}$	-0.00010525	-0.00010525	-0.00010525	-0.00010527
$Re = 7.5$				
$\frac{\partial \mathcal{J}_2^h}{\partial \lambda}$	-0.45423	-0.45423	-0.45423	-0.45423
$\frac{\partial \mathcal{J}_2^h}{\partial \alpha}$	-0.032849	-0.032849	-0.032849	-0.036069
$\frac{\partial \mathcal{J}_2^h}{\partial Re}$	-0.00017476	-0.00017476	-0.00017476	-0.00017476
$Re = 10.0$				
$\frac{\partial \mathcal{J}_2^h}{\partial \lambda}$	-0.334E-15	-0.334E-15	0.283E-06	-0.334E-15
$\frac{\partial \mathcal{J}_2^h}{\partial \alpha}$	0.608E-15	0.608E-15	0.170E-07	0.574E-15
$\frac{\partial \mathcal{J}_2^h}{\partial Re}$	0.151E-18	0.151E-18	0.356E-11	0.151E-18

may also approximate the cost gradient directly, without using an estimate for the sensitivities, using a direct finite difference approach. In Table 9.1, we compare the cost gradients we get via these methods. The problem being solved uses three parameters, one inflow, one bump, and Re , with a mesh of 240 elements and a mesh parameter $h = 0.5$. The target data was generated previously on the same mesh, setting $(\lambda^T, \alpha^T, Re^T)$ to the values $(0.5, 0.5, 10.0)$. The program generated flow solutions and cost gradients for the parameter values $(0.25, 0.25, 5.0)$, $(0.375, 0.375, 7.5)$, and $(0.5, 0.5, 10.0)$.

Note that, at $Re = 10$, where the cost gradients should be zero, it is the direct finite difference estimate that is most out of line. This error is to be expected, of course. We said that the direct finite difference estimate gave an answer that was equal to the true gradient, plus an error of order $\Delta\beta$. Since the true gradient is zero, we are seeing exactly the error term.

All four gradient estimates agree almost completely for the first and third components of the

gradient. However, there is a roughly 10% error in the estimates for the shape gradient when using discretized sensitivities. It is hard to judge this fact. We are using a crude mesh, and a finer one would improve the agreement, as we have seen in Chapter 8. Since the discretized sensitivities are arrived at by a “shortcut”, we must expect to pay a price for their ease of computation. But whether these inaccuracies in the gradient will actually hamper us in the optimization efforts is a question we will have to face shortly, in Chapter 11.

9.11 Regularization of Cost Functionals

We will shortly see a number of situations in which the optimization algorithm is unable to produce a satisfactory answer for us. This is not remarkable when we consider the constraints on the optimization code. It generally uses very local information, so that it cannot “see” that just beyond a rise in the functional lies a very substantial drop in its value. The optimization code will refuse to accept a new iterate which causes the functional value to increase. Hence, once an optimization code has entered a “valley” it generally cannot escape. It is very common for a functional to have multiple local minima, and many of these local minima can have functional values that are quite large relative to the value obtained at the global minimum.

Our first question then is, if the optimization code has stumbled into a local minimum which is unsatisfactory, but for which the “valley” is reasonably shallow, can we “escape” from somehow, and force the optimization code to try to find a better minimum?

A second, milder problem can also occur. Even though a function may have only one minimizer, it may be unusually flat, or be poorly scaled, or have level sets that are very twisted. In all these cases, it can be extremely inefficient to carry out the typical optimization algorithm, which uses local directional data, takes a large step in a descent direction, and

then penalizes itself severely if the step caused the functional to increase.

Finally, we may find that the minimizer uncovered by the optimization code has some undesirable property that we wish to avoid. For instance, if we are seeking the shape of a bump that will produce a given flow downstream, we may not realize that we want a smooth bump until the optimization code returns with a set of parameters that define an extremely oscillatory shape. The undesired oscillations can be controlled by adding a *penalty function*, which measures the amount of oscillation in the bump, and adds some small multiple of this amount to the cost function. A simple example might be

$$\mathcal{J}_3(\lambda, \alpha) \equiv \int_{x=x_s} (u(x, y) - u^T(x_s, y))^2 dy + \epsilon \int \left(\frac{\partial Bump(x, \alpha)}{\partial x} \right)^2 dx. \quad (9.31)$$

Using such a penalty function requires an *ad hoc* choice for the quantity ϵ , and also means that when the optimization code finds the solution that minimizes J_3 , there may be other flow solutions which have a smaller discrepancy integral, and hence are “better” in the old sense.