

Chapter 17

THE COMPUTATIONAL ALGORITHM

17.1 Introduction

In this chapter, we look in detail at the algorithm used to produce the results discussed in this thesis. We first give a logical outline of the order of operations in the algorithm. We next discuss the names and meaning of some of the user input quantities. We then describe the individual steps of the algorithm in terms of the variables that are used and altered at each step.

Finally, we discuss our choice of optimization software, including the algorithm employed, the input required from the user, and the types of stopping criteria.

17.2 Black Box Versus One-Shot Methods

The strategy we employ for our flow optimization is an example of the *black box* method. We may consider our overall algorithm to comprise two sub-algorithms: the flow solver, and the optimization code. The term “black box” is meant to suggest that the details of the flow solving are not used in any way by the optimization code. We may summarize our algorithm

as repeated executions of:

$$\text{Given parameters } \beta : \tag{17.1}$$

$$\text{Solve for state variables } F(u^h(\beta), v^h(\beta), p^h(\beta), \beta) = 0 \tag{17.2}$$

$$\text{Evaluate the cost } \mathcal{J}(\beta) = J(u^h(\beta), v^h(\beta), p^h(\beta), \beta) \tag{17.3}$$

$$\text{Accept } \beta, \text{ or terminate, or repeat.} \tag{17.4}$$

The function F is meant to represent the Navier Stokes system.

To understand that this is not the only possible approach, one might consider the following method, which seeks to solve the same problem, though using the cruder optimization technique of seeking a zero of the gradient of J .

$$\text{Given } (u_0^h, v_0^h, p_0^h, \beta_0) \tag{17.5}$$

$$\text{Find } (u^h, v^h, p^h, \beta) \text{ satisfying the pair of equations} \tag{17.6}$$

$$F(u^h, v^h, p^h, \beta) = 0 \tag{17.7}$$

$$\nabla J(u^h, v^h, p^h, \beta) = 0 \tag{17.8}$$

where J is the original cost functional that is written in terms of the flow functions as well as the parameters. Notice that the flow functions u , v and p used as arguments of J are *not* required beforehand to be actual flow solutions. This method would start with the arbitrary set of flow functions and parameters $(u_0^h, v_0^h, p_0^h, \beta_0)$, and try to solve $F = 0$ and $\nabla J = 0$ simultaneously. Thus, the optimization and flow portions of the calculation would be done *simultaneously*. Such an approach is called a *one-shot method*.

17.3 Black Box Flow Optimization Algorithm

Our algorithm carries out some initializations, and then repeatedly executes an optimization loop. It begins the loop with some set of parameter values, finds the corresponding flow

solution, evaluates the cost functional and the approximate cost gradient, and reports to the optimization code for further instructions. The simplified version of our algorithm that is described here assumes that our cost functional depends only on the horizontal velocity discrepancy, that the cost functional target data is computed in the usual way as sampled data of a flow generated by target parameters, and that discretized sensitivities are used to estimate the cost gradient.

Algorithm 17.1 (Black Box Flow Optimization)

Get user input.

Compute \mathbf{Utar} , the target horizontal velocities.

Initialize optimal parameter estimate \mathbf{Para} to zero.

Begin optimization loop:

Estimate flow solution $(\mathbf{U}, \mathbf{V}, \mathbf{P})$ for current parameters \mathbf{Para} .

Use Picard iteration to “improve” $(\mathbf{U}, \mathbf{V}, \mathbf{P})$ for use by Newton.

Use Newton iteration to produce correct flow solution $(\mathbf{U}, \mathbf{V}, \mathbf{P})$.

Compute discretized flow sensitivities \mathbf{dUdP} at \mathbf{Para} .

Compute $\mathbf{Cost} = \int (\mathbf{Utar}(\mathbf{XProf}, y) - \mathbf{U}(\mathbf{XProf}, y)) dy$

If finite difference sensitivities are desired, find nearby flow solutions.

Compute cost gradient vector \mathbf{dCdP} .

Report current values of \mathbf{Para} , \mathbf{Cost} and \mathbf{dCdP} to optimization code.

Optimization code makes one of the following recommendations:

Current parameters acceptable. Stop with success.

Optimization must be abandoned. Stop with failure.

New estimate of parameters in \mathbf{Para} . Repeat optimization loop

End optimization loop

In the succeeding sections, we discuss portions of this algorithm in detail.

17.4 User Input

The program accepts at runtime a file of user input. This allows the user to specify information about the following matters:

- the physical problem to be solved;
- the flow region discretization;
- the cost functional;
- starting parameter values;
- the Newton iteration;
- the cost gradient calculation;
- the optimization.

We will now list the exact information that may be specified by the user.

17.4.1 Physical Problem Input

The basic flow region is assumed to be a rectangle. The user gives the dimensions of this rectangle as follows:

XLngth The length of the flow region.

YLngth The height of the flow region.

The number of parameters used must be specified by the user:

NParB The number of bump parameters.

NParF The number of inflow parameters.

Note that there are a total of $\mathbf{NPar}=\mathbf{NParB}+\mathbf{NParF}+1$ parameters, because the Reynolds number Re is always considered a parameter, although it need not be varied in a particular problem.

17.4.2 Discretization Input

The flow region is covered by a grid of nodes. The columns of nodes are evenly spaced in the x direction. The nodes are then evenly spaced in the y direction, bearing in mind that the flow region may have a bump, and that the y coordinate of nodes over the bump must be adjusted each time the bump parameters change. Two parameters specify the density of nodes:

NX The number of x grid lines will be $2*\mathbf{NX}-1$.

NY The number of y grid lines will be $2*\mathbf{NY}-1$.

The mesh parameter h may be determined by the computations:

$$\mathbf{HX} = \mathbf{XLength}/(2 * (\mathbf{NX} - 1)) \quad (17.9)$$

$$\mathbf{HY} = \mathbf{YLength}/(2 * (\mathbf{NY} - 1)) \quad (17.10)$$

$$\mathbf{H} = \mathbf{Min}(\mathbf{HX}, \mathbf{HY}) \quad (17.11)$$

Of course, this calculation ignores the fact that, for various bump parameter values, the y mesh spacing will differ from \mathbf{HY} in the region above the bump.

17.4.3 Cost Functional Input

In our usual formulation, the cost functional is determined by picking a set of target parameters, generating the corresponding flow solution, and sampling the flow field along some specified vertical profile line. Then the integrals of the discrepancy between the components of the target flow and the current flow are computed. Certain penalty integrals may also be computed. The cost functional is then computed as a weighted sum of these integrals. The input that specifies this process includes:

ParTar	The target parameters.
WateB	Multiplier for the bump slope penalty integral.
WateP	Multiplier for p discrepancy integral.
WateU	Multiplier for u discrepancy integral.
WateV	Multiplier for v discrepancy integral.
XProf	The x coordinate of the profile line.

17.4.4 Initial Parameter Input

We generally prefer to initialize our parameters to zero. However, this choice is left up to the user. The user also specifies which parameters are allowed to vary during the optimization. Typically, for instance, the Reynolds number is held fixed.

Para1	The initial parameter values.
lopt	Indicates which parameters may be varied.

17.4.5 Simple and Newton Iteration Input

For each new set of parameters, an estimate of the flow solution is made. The true flow solution would satisfy $F(U, V, P) = 0$, where F represents the discretized Navier Stokes equations and the boundary conditions. Assuming our flow solution estimate does not satisfy this requirement, we apply a hybrid algorithm involving Picard iteration followed by Newton iteration to try to reduce the discrepancy. The user controls this scheme through the following variables:

- IJac** The Jacobian matrix will be updated every **IJac** steps.
- MaxNew** The maximum number of Newton steps per iteration.
- MaxSim** The maximum number of Picard iteration steps.
- TolNew** The Newton convergence tolerance.
- TolSim** The Picard convergence tolerance.

17.4.6 Cost Gradient Input

The cost gradient $\frac{\partial \mathcal{J}^h(\beta)}{\partial \beta_i}$ must be approximated for the optimization. The user controls how this is done:

- IGrad** 0, no gradient estimate is made
- 1, use the chain rule on discretized sensitivities.
- 2, use chain rule on finite coefficient differences.
- 3, use chain rule on adjusted finite coefficient difference differences.
- 4, use finite cost function differences.

17.4.7 Optimization Input

The user controls the optimization.

MaxStp The maximum number of optimization steps.

TolOpt The optimization convergence tolerance.

17.5 Algorithm Details

After reading the user input, the program must set up the data defining the cost functional. To do so, the program must produce the flow solution for the target parameters **ParTar**. It makes an initial guess of

$$\mathbf{UTar} = 0. \tag{17.12}$$

A Newton iteration then begins with this starting point, and proceeds until the convergence criterion is met. The resulting flow field is saved in its entirety for use in the cost functional.

The current estimate **ParNew** for the optimizing parameters is initialized to the user values in **Para1**. The loop expects “old” quantities evaluated at the previous solution, and we initialize them as

$$\mathbf{ParOld} \leftarrow \mathbf{ParNew} \tag{17.13}$$

$$\mathbf{UOld} \leftarrow 0 \tag{17.14}$$

$$\mathbf{dUdPOld} \leftarrow 0 \tag{17.15}$$

Inside of the optimization loop, our task is to compute the flow solution, and associated quantities, for the current set of parameters. We assume that we have the flow solution **UOld** corresponding to some previous parameter values **ParOld**, and some approximation

to the sensitivities of that solution, $\mathbf{dUdPold}$. Our initial estimate of the new value of \mathbf{U} is then:

$$\mathbf{U}(\mathbf{I}) \leftarrow \mathbf{Uold}(\mathbf{I}) + \sum_{\mathbf{J}=1, \mathbf{NPar}} \mathbf{dUdPold}(\mathbf{I}, \mathbf{J}) (\mathbf{ParNew}(\mathbf{J}) - \mathbf{ParOld}(\mathbf{J})). \quad (17.16)$$

We now may take several steps of Picard iteration. On each step, we use the current estimate of the solution \mathbf{U} to linearize the function $\mathbf{F}(\mathbf{U})=0$, replacing it by the linear system $\mathbf{H}(\mathbf{U}, \mathbf{UNew})=0$. We solve for \mathbf{UNew} , set \mathbf{U} to \mathbf{UNew} , and repeat.

The Newton iteration begins at the improved starting point \mathbf{U} returned by Picard iteration. The Newton iteration seeks an approximate solution of the set of equations:

$$\mathbf{F}(\mathbf{U}) = 0. \quad (17.17)$$

To do so, it requires the formation and factorization of the Jacobian matrix $\mathbf{FPrime}(\mathbf{U}, \mathbf{Para})$. A factored value of \mathbf{FPrime} from a previous step may be re-used, for efficiency, as specified by the user input quantity \mathbf{Jac} . The Newton iteration terminates successfully when the maximum norm of $\mathbf{F}(\mathbf{U})$ is less than \mathbf{TolNew} . If the Newton iteration does not terminate after \mathbf{MaxNew} steps, then a fatal error occurs, and the program halts.

We may now evaluate the cost functional at the new solution. To do so, we typically are required to evaluate the integral of $(\mathbf{U}(\mathbf{x}, \mathbf{y}) - \mathbf{UTar}(\mathbf{x}, \mathbf{y}))^{**2}$ along the line $\mathbf{x}=\mathbf{XProf}$. (In some cases, we may need to compute other terms as well). The integral is estimated using Gaussian quadrature, and saved as \mathbf{Cost} .

It is now necessary to estimate the cost functional gradient. If the discretized sensitivities are to be used, then for each varying parameter component \mathbf{J} , we set up $\mathbf{RHS}(\mathbf{J})$, the right hand side of the sensitivity system, and solve the set of equations:

$$\mathbf{FPrime} \cdot \mathbf{dUdP} = \mathbf{RHS} \quad (17.18)$$

for the J -th component of the sensitivities, \mathbf{dUdP} .

If a finite coefficient difference approach is being used, then for each varying parameter component \mathbf{J} , we temporarily set

$$\begin{aligned}\mathbf{Par2} &\leftarrow \mathbf{ParNew}, \text{ except that} \\ \mathbf{Par2}(\mathbf{J}) &\leftarrow \mathbf{ParNew}(\mathbf{J}) + \mathbf{Del}(\mathbf{J}) \\ \mathbf{U2} &\leftarrow \mathbf{U}\end{aligned}$$

and carry out a Newton iteration on $\mathbf{U2}$ to get a satisfactory flow solution. Then the finite difference sensitivities are computed as

$$\mathbf{dUdP}(\mathbf{I}, \mathbf{J}) \leftarrow \frac{(\mathbf{U2}(\mathbf{I}) - \mathbf{U}(\mathbf{I}))}{\mathbf{Del}(\mathbf{J})} \quad (17.19)$$

although, if shape parameters are involved, we may also want to add the adjustment term to account for movement of the nodes associated with the coefficients.

We may now use the approximated sensitivities, plus the partial derivatives of the cost functional with respect to the state variables, \mathbf{dCdU} , to estimate the partial derivatives of the cost with respect to parameter \mathbf{J} , using the chain rule:

$$\mathbf{dCdP}(\mathbf{J}) \leftarrow \sum_{I=1, N_w} \mathbf{dCdU}(\mathbf{I}) * \mathbf{dUdP}(\mathbf{I}, \mathbf{J}) \quad (17.20)$$

Alternatively, we may approximate the partial derivatives of the cost functional directly by finite differences. In that case, one at a time, we vary a single parameter, compute the cost functional, and form the appropriate finite difference quotient that estimates the derivative.

Our cost function \mathbf{Cost} and cost gradients \mathbf{dCdP} are now passed to the optimization code, which makes the determination of whether the optimization loop should be continued with a new estimate of the minimizer, or terminated because of convergence or errors.

17.6 Computational Optimization

Well-written software is available to treat numerical optimization problems of the type we have considered. The program we are using incorporates a minimization package written by Gay [12], published as ACM TOMS Algorithm 611. Attractive features of this package include:

- Its status as an ACM Algorithm;
- The user can choose to provide the gradient and Hessian, or have the Hessian, or both Hessian and gradient, approximated internally. This allows us to compare the use of optimization with no derivatives, or with approximate derivatives using sensitivities or finite differences. It also means we can later try to exploit second derivative approximations if we can make them;
- It may be called in “reverse communication mode”. This makes it very easy to work with functions that require extensive preprocessing before evaluation;
- The source code was publicly available through **netlib**[11];
- The program is easily portable to a variety of computers.

The program may be categorized as a *BFGS model/trust region method*.

The program requires that the user supply a starting point β_0 , a tolerance ϵ and an evaluation procedure for $J(\beta)$. If the user does not supply an evaluation procedure for the gradient of J , then this will be estimated using finite differences. The program also maintains an internal estimate of the Hessian matrix $H(\beta)$.

The point β_0 is taken as the initial estimate for the minimizer. The program will proceed in a series of steps, with each step being an attempt to replace the current estimate for the

minimizer by a better estimate, that is, by a point with a lower functional value.

Naturally, some times the program will find, instead, a point with a higher functional value. These points will be rejected, and the program will repeat its search based at the old point, using greater caution, to produce a new candidate. This insistence on progressive functional decrease means that the program cannot “climb up” out of a depression corresponding to a local minimum.

On each step of the iteration, the program uses recent functional values and exact or approximate information about the gradient and Hessian to construct a quadratic *model* of the functional’s behavior in a small neighborhood of the current estimate. This neighborhood, in which the program believes its model to be a good approximation, is called the *trust region*. It then analyzes the model, and estimates the direction and distance of a minimizer of this model. If the estimated minimizer lies outside the trust region, the program chooses instead the point within the trust region that is closest to the estimated minimizer.

Having determined its candidate for a better minimizer, the program then requires the user to evaluate the functional at this new point. The new functional value is used to update the model. If the functional value is higher than the lowest value seen so far, then the point is not accepted, the current trust region is reduced in size, the model is revised, and a new prediction is made. Otherwise, the estimated minimizer is accepted, a new trust region and model are constructed around the new point, where the new prediction is made.

The tolerance ϵ is used in order to determine when to halt the iterative optimization, which could otherwise continue forever, assuming perfect computational accuracy. The code stops when it finds that certain things are “small” compared to ϵ . Such criteria include:

- the gradient vector of the cost functional is small;
- the distance between the current point and the minimizer is believed to be small;

- the latest steps have produced only small changes in the estimated minimizer;
- the latest steps have produced only small changes in the value of the functional.

Normally, an optimization is fairly uneventful; the optimization software might proceed slowly, or make a series of poor choices for the minimizer, but it is very rare that it is unable to proceed. One case that will cause problems, and which the optimization code will notice, occurs if the user supplies incorrect gradient information. If the gradient is quite wrong, then the optimization code will find that for points in the direction of the negative gradient, the functional actually increases rather than decreases, and that this problem persists even for very small steps. In such a case, the optimization code will return with the message “False convergence”. In this case, the word “convergence” is only meant to convey the fact that no further progress can be made.