

Chapter 16

EFFICIENT SOLUTION OF THE OPTIMIZATION PROBLEM

16.1 Introduction

The optimization of a cost functional associated with a Navier Stokes problem is *expensive*; that is, the usual solution algorithm requires a considerable amount of arithmetic operations and computer time. It is important to investigate whether there are simple techniques for reducing the computational expense.

We choose a standard problem and estimate the amount of work that will be required to carry out an optimization using our original algorithm. We then consider several variations of this algorithm that strive to reduce the amount of work involved.

We also investigate how the computational expense will grow as the Reynolds number is increased, which makes the nonlinear terms more prominent, and the problem harder to solve.

16.2 A Standard Test Problem and Algorithm

To investigate methods for speeding up the optimization, we will find it useful to pick a standard problem, carry out the optimization procedure, and carefully record the computational effort required, paying particular attention to the assembly and factorization of the Jacobian matrix.

As a standard problem, we take a four parameter case, with one inflow parameter and three bump parameters. We fix the Re parameter at 1. Target data is generated using the parameter values $\lambda^T = 0.5$, $\alpha^T = (0.375, 0.5, 0.375)$.

For the discretized problem, we will use a mesh parameter $h = 0.125$, resulting in 960 elements, 2025 nodes, and 4190 unknowns. The Jacobian matrix is banded, with a bandwidth of 225 entries.

For the low Reynolds numbers we will be examining, it is usually not necessary to employ Picard iteration. Therefore, we will usually rely exclusively on Newton's method, using Picard iteration only as a fallback. We will use a Newton convergence tolerance of **TolNew**=1.0E-9.

The optimization will be carried out with an optimization tolerance of **TolOpt**=1.0E-9. We will start from the usual zero initial estimate, which has a cost functional value of 0.436.

To carry out the computation, let us begin with a simple-minded approach, which we will call *Algorithm 1*. Algorithm 1 will re-evaluate and refactor the Jacobian on every step of the Newton process. Once the main solution point is computed, finite coefficient differences will be made to get approximations of the sensitivities. The proper computation of these finite difference points will require yet another Newton process for each parameter component; we will update the Jacobian during these subsidiary Newton processes as well.

If we apply this algorithm to our problem, 34 optimization steps are required, after which

the optimization code declares satisfactory convergence to the parameters $\lambda = 0.499999$, $\alpha = (0.374365, 0.500065, 0.375045)$. At this point, the cost functional $\mathcal{J}_2^h(\lambda, \alpha) = 0.363E - 9$ and the estimated gradients are $\frac{\partial \mathcal{J}_2^h}{\partial \lambda} = 0.6E - 05$ and $\frac{\partial \mathcal{J}_2^h}{\partial \alpha} = (-0.2E - 06, 0.4E - 05, 0.6E - 05)$.

During this run we computed flow solutions at 169 separate sets of parameter values, which in turn required a total of 187 Newton iteration steps, as well as 187 Jacobian evaluations and factorizations. The optimization of our standard problem required 1112 seconds of CPU time on a DEC Alpha.

16.3 Expensive Computational Tasks

A timing analysis of the program reveals that the most expensive operations are:

- The evaluation of the finite element basis functions;
- The evaluation and factorization of the Jacobian;
- The solution of linear systems.

with the Jacobian factorization being the single most expensive item.

We have already attempted to economize on the basis function evaluations; we precompute the values at all quadrature points, and then simply look them up when needed. However, if there are geometric parameters, then for each new set of parameters these quantities must be recomputed, reducing our efficiency gain. No further improvement is likely here, since the number of evaluations on the finite element basis functions depends in turn on the number of evaluations of the Jacobian.

Our Jacobian is symmetrically banded. We apply the LAPACK banded factorization routine DGBTRF to compute its LU factors. The amount of work required for this operation is

Table 16.1: Matrix work required for various values of h .
 Work is measured in millions of floating operations.

h	MBand	Neqn	Factor (MegaOps)	Solve (MegaOps)
0.2500	66	1166	10	0.2
0.1666	93	2555	44	0.7
0.1250	120	4484	129	1.6
0.0833	174	9962	603	5.2
0.0625	228	17600	1829	12.0

roughly $O(2 * Neqn * Mband^2)$, where $Neqn$ is the number of equations represented by the matrix, and $Mband$ is the half bandwidth.

To solve a particular linear system, we use the corresponding LAPACK routine DGBTRS. The cost for a single linear solution from this routine is roughly $O(3 * Neqn * Mband)$.

The number of equations and the bandwidth depend, in turn, on the mesh parameter h . As the mesh parameter is halved, for instance, we expect roughly that $Mband$ will double, and $Neqn$ quadruple. This means that the computational work goes up by a factor of 16. If we halve h twice, it will take us roughly 256 times as long to factor the corresponding Jacobian. No other operation in the program will have this explosive growth with decreasing h . Therefore, as h decreases, it is vital that we be able to reduce the number of Jacobians we need. In some cases, reducing the number of Jacobians will increase the number of linear solutions required.

Table 16.1 lists the amount of work required for factorizations and solutions for a sequence of values of h . We can see, for instance, that in going from $h = 0.25$ to $h = 0.0625$, our estimate of a 256-fold increase in factorization work is reasonably close to the actual 182-fold increase. From this table, we can see that it takes quite a large number of linear solutions to equal the cost of one factorization.

Of course, there are many hidden costs that this table cannot show. For instance, we have not measured the costs of *evaluating* the Jacobian, which is roughly equal to the cost of evaluating the Newton residual. To make a more accurate assessment of, say, the relative cost of one Jacobian factorization versus several Newton steps, we would need to measure the evaluation work as well.

However, this table has given us a clear indication of the large amount of work required for Jacobian factorization, and its explosive growth as h decreases.

16.4 A Comparison of Algorithms at $Re = 1$

Since the evaluation and factorization of the Jacobian is our most costly computational task, we will now investigate whether we can reduce the overall cost of the computation by trying to reduce the amount of work devoted to the Jacobian.

The simplest change we can make to Algorithm 1 is to hold the Jacobian fixed when computing flow solutions required as part of a finite difference calculation. This is surely a reasonable economy; the last Jacobian calculated in the Newton iteration for a main solution point is likely to be quite close to the Jacobian at any of the finite difference points, whose parameters differ in a minuscule way from those of the main solution point. We will call this method *Algorithm 2*.

In fact, during a Newton iteration, the iterates are all likely to be close enough so that we could actually use a single Jacobian to compute several or all of the Newton steps. For our purposes, we will assume that at most 5 Newton steps in a row are computed with a fixed Jacobian, and that the Jacobian is *always* reevaluated and factored at the beginning of each Newton process. As before, we will hold the Jacobian fixed for Newton processes associated with finite difference computations. We call this method *Algorithm 3*.

Table 16.2: Work carried out for various algorithms, $Re = 1$.
 Pure Newton algorithm, tolerance **TolNew**=1.0E-9.

Algorithm	Opt Steps	Flow Solves	Newton Steps	Matrix Factors	CPU (sec)
1	34	169	187	187	1112
2	34	169	187	55	450
3	33	164	187	33	337
4	37	184	608	6	424
5	35	38	73	73	498
6	34	37	90	34	310
7	36	179	436	13	400

To see how far we can reduce the factorizations of the Jacobian, we also consider *Algorithm 4*, which *never* evaluates the Jacobian except on the very first Newton process, or if the Newton process has failed to converge. *Algorithm 7* is similar, but instead of waiting for the Newton process to fail, we evaluate the Jacobian and then reuse it for the entire Newton process, and for all Newton processes associated with “nearby” parameter sets.

Algorithm 5 is similar to Algorithm 2, except that we use discretized sensitivities instead of finite coefficient differences. *Algorithm 6* is the same as Algorithm 3, except that we use discretized sensitivities instead of finite differences.

We applied these algorithms to our standard problem, and display the results in Table 16.2.

What may we conclude from these results? At the moment, we wish primarily to consider questions of efficiency. If we consider the CPU measurement to be a reasonable measure of the overall work performed by the algorithm, then we may take our task to be the explanation of the timing results in terms of the underlying algorithms.

The first remarks then go to the longest run by far, our basic algorithm. We know that matrix factorization is the most expensive single operation in our algorithm; this correlates well with the very high number of factorizations carried out by Algorithm 1.

Surely the number of factorizations cannot be the only factor, else Algorithm 4 would have been our quickest, with only 5. The fact that Algorithms 3 and 6, with more matrix factorizations, are substantially faster, means that there are other sources of work that must also be controlled. Indeed, we can note that Algorithm 4 paid for its very low factorization rate with a correspondingly high number of Newton iterations. Each Jacobian evaluation, for instance, was preceded immediately by a Newton iteration that failed to converge after taking the maximum number of steps, accounting for a total of 200 steps. Since each Newton step requires the solution of a linear system, which is by no means a cheap process, we can attribute the mediocre performance of Algorithm 4 to a failure to control the number of linear system solves.

Algorithm 7, while similar to #4, manages to reduce the number of Newton steps by about 200, at a slight cost of 7 extra Jacobian evaluations. However, there is little improvement in speed for this problem.

Now let us consider Algorithm 6, which produced the lowest time to solve the problem. It is reasonable to attribute its speed to the combination of a low number of matrix factorizations and a very low number of Newton iterations. Algorithm 6 has limits on its stinginess: it reuses the Jacobian, but no more than 4 times. This seems to guarantee good Newton convergence behavior at a significantly lower cost in factorizations.

We note that the second best timing was for Algorithm 3, which actually had a high number of flow solves and Newton steps (because it used finite differences instead of sensitivities). Apparently, the fact that it was able to keep the number of factorizations almost as low as for Algorithm 6 was the decisive factor. The rule seems to be that factorizations are very important; linear solves are only important if their number gets “out of hand”.

So far, our results at $Re = 1$ have suggested that we need to be aware of an appropriate

Table 16.3: Work carried out for various algorithms, $Re = 100$.

Algorithm	Opt Steps	Flow Solves	Newton Steps	Matrix Factors	CPU (sec)
1	36	180	218	218	6189
2	38	190	228	80	2639
3	37	185	292	43	1813
4	38	190	1029	9	1912
5	36	40	105	105	3146
6	49	58	292	96	3298
7	39	195	702	11	1514

error tolerance for the Newton iteration and that we need to use the Newton system matrix wisely, neither updating it every step, nor reusing it to the point where the Newton process is severely hampered.

16.5 A Comparison of Algorithms at $Re = 100$

Our efforts at improving the efficiency of the algorithm do not show as much profit as we might like. But this may be that the flow problem is too easy to solve when the Reynolds number is low. As Re increases, the problem becomes more nonlinear, the Newton process requires more steps, and conservation of Jacobian evaluations will provide a bigger payoff. Let us recompute our results using the same set of algorithms, but for a Reynolds number of $Re = 100$, and a mesh spacing of $h = 0.083$. Table 16.3 shows the new timings.

Here we see a dramatic separation of the algorithms. Algorithm 1 is, as expected, the most expensive. But now we can rate Algorithms 2, 5, and 6 as mediocre, Algorithms 3 and 4 as good, and Algorithm 7 as superior.

We can find ready explanations for some of the results in the table. Our mesh parameter h had to decrease because the Reynolds number had increased. The overall cost for each algorithm is then multiplied by a factor of 4 to 10. The smaller mesh spacing means more

variables, and hence a bigger Jacobian. The cost of evaluating and factoring the Jacobian matrix becomes relatively more important, and hence, algorithms that conserve on this single cost will do better.

The fact that Algorithm 7 did better than Algorithm 4 emphasizes the importance of “preventive” Jacobian updating. It can be a bad idea to wait until the Jacobian is absolutely useless before updating it; a very poor Jacobian may be usable, but very ineffective. Algorithm 7’s technique of updating the Jacobian when the parameters have changed significantly seems to pay off. Good, fresh Jacobian information is needed repeatedly in the beginning of an optimization, but towards the end, when the iterates cluster around a single value, there is much less need to reevaluate.

Algorithm 3, which updates the Jacobian on each Newton process, but then holds it fixed, should be compared to Algorithm 2, which updates the Jacobian on every Newton step. Algorithm 3 cut down the number of Jacobian evaluations by roughly half, and was able to cut 800 seconds off its CPU time.

It is surprising that Algorithm 6 did so poorly. It was the best performer for $Re = 1$. However, it had a failure of the Newton process early in the computation, which cost it 60 wasted Newton steps, and it then seems to have had optimization difficulties, because of the use of discretized sensitivities, that made it compute 49 optimization steps, much above the range of 36 to 39 required by the other algorithms.

16.6 Appropriate Newton Tolerances

Since the Newton iterations are our main expense, it might seem that an easy way to cut down on work would be to loosen the Newton tolerance **TolNew** used for determining convergence. It is surely true that this can have the effect of shortening any one Newton

process. However loosening the Newton tolerance means increasing the inaccuracy that we will accept in our solution; this same inaccuracy then affects our calculations of the cost function, the sensitivities, and the cost gradients. Inaccurate cost data can greatly prolong the optimization process, particularly because descent directions will be inaccurate. And once the inaccuracies in the computed cost functional become large enough, the formerly smooth graph of the cost functional will behave as though it were covered by spurious local minima, bringing the optimization process to a confused and incorrect halt.

Normally, one would simply choose the Newton tolerance to be a “small” value. We have typically used values of the order of 1.0E-9 or 1.0E-10. However, working on higher Reynolds number problems, we considered using a tolerance of 1.0E-7. We were surprised at certain unexpected behaviors in the optimization process, and so we returned to a lower Reynolds number where we had more experience and did some comparisons.

As a model problem for this study, let us take a four parameter problem, with one inflow parameter λ and three bump parameters α , with target parameters $\lambda^T = 0.5$ and $\alpha^T = (0.375, 0.5, 0.375)$ and Re fixed at 1. A mesh parameter of $h = 0.125$ was used. Let us use adjusted finite coefficient differences as estimates of the discrete sensitivities used to compute the cost functional gradient. (Note that we don’t update the Jacobian for Newton steps carried out on finite difference computations.) Then, if we vary the Newton tolerance **TolNew**, leaving all other quantities fixed, we get the work estimates listed in Table 16.4.

The table indicates that an acceptable solution can be computed for **TolNew**=1.0E-9 or less. The most efficient calculation occurs at **TolNew**=1.0E-10, where, it seems the cost of accuracy in the Newton solution has been roughly balanced by the benefits of a rapid, accurate optimization. As the tolerance grows above this range, the quality of the optimization deteriorates very rapidly, because the data is too poor to work with. If instead, we decrease the tolerance, the work slowly but relentlessly rises, as we waste time computing

Table 16.4: Work carried out for various Newton tolerances, $Re = 1$. “PErr” is the L2 parameter error between the computed and global minimizers.

TolNew	Opt Steps	Flow Solves	Newton Steps	Matrix Factors	CPU (sec)	PErr
1.0E-05	63	314	314	69	470	0.44
1.0E-06	40	199	192	36	265	0.23
1.0E-07	31	154	152	32	221	0.26
1.0E-08	36	179	193	52	318	0.0004
1.0E-09	34	169	187	55	316	0.001
1.0E-10	33	164	183	55	305	0.001
1.0E-11	34	169	193	61	330	0.001
1.0E-12	34	169	197	65	346	0.001
1.0E-13	34	169	198	66	349	0.001
1.0E-14	34	169	301	70	425	0.001
1.0E-15	34	169	335	77	471	0.001

over-accurate solutions. For a tolerance of **TolNew**=1.0E-16, the entire optimization fails because the Newton iteration cannot converge to the given tolerance.

Note, however, what a very simple problem this must be. Up until **TolNew**=1.0E-14, the number of flow solutions is roughly equal to the number of Newton steps. In other words, a single Newton step is enough to correct the initial guess to the desired tolerance.

We repeat the calculations for a higher Reynolds value of 100, with the results shown in Table 16.5. Certain features of our previous results persist. In particular, we see again that the overall CPU time suddenly increases if the tolerances are too loose, while the CPU time slowly rises as we decrease the tolerances. Again, there seems to be a discontinuous behavior in the estimation of the parameters. Above a certain tolerance, the approximation is very poor. Below it, the approximation seems to be about the same, regardless of the tolerance used; only the work increases.

When comparing results at $Re = 1$ and $Re = 100$, we notice that the parameters for the global minimizer are approximated better for $Re = 100$, by about a factor of 10, and that

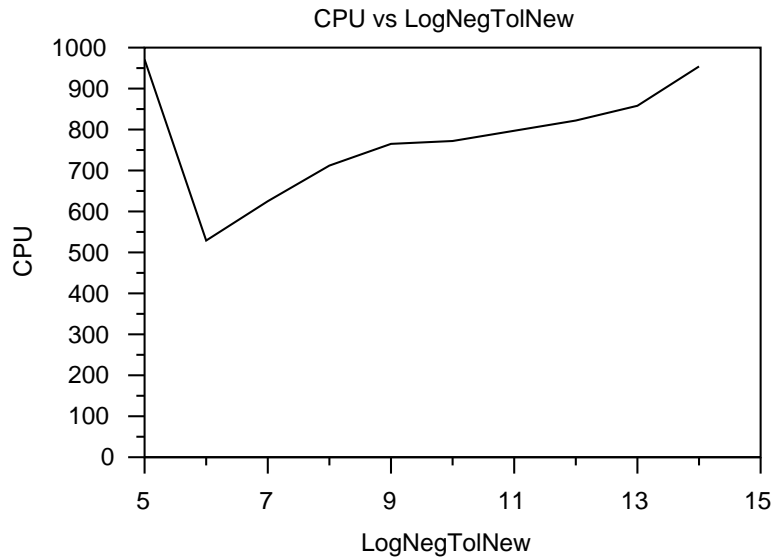


Figure 16.1: CPU time as a function of Newton tolerance, $Re = 1$.

this better approximation begins for a higher tolerance. On the other hand, the amount of work required to optimize the $Re = 100$ problem begins to rise at a much higher Newton tolerance, suggesting that the gradients are well approximated already. This suggests that while the higher Reynolds number flows are harder to solve *as flow problems*, they are easier to solve *as optimization problems*. In other words, in a high speed flow, small differences in the shape of the obstacle make a bigger difference in the downstream flow.

During a particular optimization, it may not at first be clear what an appropriate Newton tolerance should be. Fortunately, the signs of an inappropriate Newton tolerance can be easy to spot: an optimization that seems to drift aimlessly once the cost functional or gradient values have dropped small enough, or a situation where the Newton iteration repeatedly fails to reach the requested tolerance.

Table 16.5: Work carried out for various Newton tolerances, $Re = 100$.
 “PErr” is the L2 parameter error between the computed and global minimizers.

TolNew	Opt	Flow	Newton	Matrix	CPU	PErr
	Steps	Solves	Steps	Factors	(sec)	
1.0E-05	64	316	350	112	971	0.18
1.0E-06	38	186	193	59	529	0.44
1.0E-07	42	210	234	70	625	0.0002
1.0E-08	43	215	252	84	712	0.0002
1.0E-09	42	210	257	93	765	0.0005
1.0E-10	41	205	255	95	772	0.0001
1.0E-11	41	205	259	99	797	0.0001
1.0E-12	41	205	263	103	822	0.0001
1.0E-13	41	205	307	105	858	0.0001
1.0E-14	41	205	428	112	954	0.0002