Optimization of Automotive Valve Train Components With Implicit Filtering
by
Owen Eslinger
April 1999
Advisor: Dr. Tim Kelley
Joint Work: Tony Choi
Supported by NSF grant $\#DMS-9700569$
Computing activity was supported by an allocation from the North Carolina Supercomputing Center.

Chapter 1

1.1 The Problem

There is a class of problems whose goal is to find the optimal solution of a given function $f(x)$. These problems are called, appropriately, optimization problems. The solution to these problems may be either a minimum or a maximum of $f(x)$. A local minimizer of $f(x)$ is x^* with

$$f(x^*) = \min_{x \in D} f(x) \tag{1.1}$$

where D is the set of all x near x^*. We will refer to $f(x)$ as the objective function and D as the domain of $f(x)$. A local maximizer is defined similarly. However, if x^* is a local maximizer of $f(x)$, then there is a local minimum of $-f(x)$ at x^*. Therefore, we generalize all maximization problems and only look for minima. Our problem then is to find $x^* \in D$ such that (1.1) is satisfied.

If we do not limit the independent variable x, then $D = R^N$ and we say that (1.1) is an unconstrained problem. If D is a proper subset of R^N, we say that x is constrained and (1.1) becomes a constrained optimization problem. We will be looking at unconstrained optimization.

1.2 Notation

The conventions of [1] are used throughout this paper. All vectors will be column vectors. As before, x^* will be used to denote a local minimizer of f, and the variable x will be any potential solution in D. The first guess we have for the solution will be called x_0 and all subsequent guesses will give a sequence of iterates, which we will call $\{x_k\}_{k\geq 0}$. An error in x will be written $e = x^* - x$ and the error for the nth iterate will be $e_n = x^* - x_n$. The variable x is in R^N, so the ith component of x will be $(x)_i$. The partial derivative of f with respect to $(x)_i$ will be written $\partial f/\partial x_i$.

We will denote the gradient of f as ∇f and note that $\nabla f \in R^N$. Specifically,

$$\nabla f = (\partial f/\partial x_1, \partial f/\partial x_2, \ldots, \partial f/\partial x_N).$$

Continuing, we let $\nabla^2 f$ denote the Hessian of f, where

$$(\nabla^2 f)_{ij} = \partial^2 f/\partial x_i \partial x_j.$$

We will also use the Euclidean Norm

$$\|x\| = \sqrt{\sum_{i=1}^{N} (x)_i^2}$$

for vectors. The norm of a matrix B will be the matrix norm induced by the Euclidean norm

$$\|B\| = \max_{x\neq 0} \frac{\|Bx\|}{\|x\|}.$$

It is also necessary to define the following terms.

Definition 1 *A matrix A is positive semi-definite if $x^T Ax \geq 0$, $\forall x \in R^N$. A is positive definite if $x^T Ax > 0$, $\forall x \in R^N$, $x \neq 0$. If A has both positive and negative eigenvalues, we say A is indefinite. If A is symmetric and positive definite we say A is spd.*

1.3 Conditions for Optimality

In this section, the theorems are stated without proofs. For formal proofs we refer the reader to [1].

Let f be twice continually differentiable. Then we have the following theorems. The first gives necessary condtions for optimality, and the second gives sufficient condtions.

Theorem 1 *Let x^* be a local minimizer of f. Then $\nabla^2 f(x^*)$ is positive semidefinite and*

$$\nabla f(x^*) = 0.$$

Theorem 2 *Assume that* $\nabla f(x^*) = 0$ *and that* $\nabla^2 f(x^*)$ *is positive definite. Then* x^* *is a local minimizer of* f.

1.4 Termination Criteria

Our goal is to optimize f over D. We will do this by making our sequence of guesses $\{x_n\}$ approach the optimal point x^*. Unless we are lucky, we will not land exactly on the optimal point. Therefore, we will have to use some criterion to know when to stop updating the iteration. We can use Theorem 2 for this purpose. At an optimal point x^*, $\nabla f(x^*) = 0$. For x near x^*, $\|\nabla f(x)\|$ will likewise be close to zero. Thus, we can view $\|\nabla f(x)\|$ as an indication of our nearness to a solution. This allows us to establish a condition under which we will terminate the iteration and report an answer. If $\|\nabla f(x)\|$ is very small, then terminate. How small is small?

We could simply compare $\|\nabla f(x_c)\|$ for our current iteration to $\|\nabla f(x_0)\|$ for our original. If we only demand that there is a significant change relative to the original, then we can terminate when

$$\|\nabla f(x_c)\| < \tau_r \|\nabla f(x_0)\| \tag{1.2}$$

where $0 < \tau_r < 1$. In this manner we will have significantly reduced the gradient norm and we should be close to the optimal point. However, if our original point x_0 has a very small $\|\nabla f(x_0)\|$, then we may not be able to use floating point numbers to satisfy (1.2). A simple solution for this would be to have both a relative decrease, indicated by τ_r, and an absolute decrease, indicated by τ_a with $0 < \tau_a < 1$. Then our termination criterion becomes

$$\|\nabla f(x_c)\| \leq \tau_r \|\nabla f(x_0)\| + \tau_a. \tag{1.3}$$

We will use (1.3) as our termination criterion from this point on in the paper. At any point in which an algorithm tests for termination, that algorithm will use (1.3) unless it is otherwise indicated.

1.5 Descent

Now that we have established sufficient and necessary conditions for optimality, and a way to terminate our iteration, it is possible to once again address (1.1) and our problem.

If we have our objective function $f(x)$, then we know that $\nabla f(x)$ should give us a direction in which $f(x)$ is increasing. In fact, $\nabla f(x)$ should be the direction of greatest increase. Then naturally $-\nabla f(x)$ should give us the direction of greatest decrease. We call this direction the direction of steepest descent. In the next section we will introduce a minimization scheme called the Method of Steepest Descent that takes advantage of $-\nabla f(x)$ when updating the iteration. In general we can define any direction d to be a descent direction if moving in that direction reduces the function value. In fact we can say that d is a descent direction if

$$\left.\frac{df(x+td)}{dt}\right|_{t=0} = \nabla f(x)^T d < 0. \tag{1.4}$$

In order to define a minimization scheme, we need something tangible that indicates an improvement in our iteration. We are seeking to minimize the function, so we hope to decrease the function value. If x_c is our current iteration and x_+ is our next iteration, then we define simple decrease as

$$f(x_+) < f(x_c). \tag{1.5}$$

We will find that in general simple decrease alone is not enough for our purposes. Simple decrease alone allows us to take very small steps that satisfy (1.5), and to stagnate at some x that is not a minimum. Therefore, we must define a decrease condition that allows us to update the iteration in a manner which will not stagnate. We will call such a decrease a sufficient decrease for our scheme. If we have sufficient decrease, then we will update our iteration and continue our scheme. Each time we create a different minimization method, we will associate a sufficient decrease definition with it.

Now, we have developed necessary and sufficient conditions for optimality, which gave us a termination criterion. We have also defined simple decrease and we will define sufficient decrease for a method. With these ideas together, we can start to form a simple algorithm that will attempt to find x^* for a given f.

1.6 Method of Steepest Descent

We have stated that $-\nabla f(x)$ is a descent direction. Using this, we define an iteration update by

$$x_+ = x_c - \lambda \nabla f(x_c).$$

This technique is the Method of Steepest Descent.

Our step lengeth λ is defined as $\lambda = \beta^m$, where $0 < \beta < 1$ and m is a positive integer. Furthermore, we want m to be the smallest number that gives sufficient decrease in $f(x)$. For our steepest descent algorithm, we will make sufficient decrease mean

$$f(x_c - \lambda \nabla f(x_c)) - f(x_c) < -\alpha\lambda \|\nabla f(x_c)\|^2 \tag{1.6}$$

where $0 < \alpha < 1$. Typically we set $\alpha = 10^{-4}$. This update stategy is called the Amijo Rule. For futher development of our choice for sufficient decrease, we refer the reader to [1].

Using (1.6), we can create an algorithm using the iteration update defined above.

Algorithm 1 *Steep(f,$x = x_0$,$kmax$)*

1. *For $k = 1, \ldots, kmax$*

   (a.) Compute $f(x)$ and $d = -\nabla f(x)$. Check for termination using (1.3).

   (b.) Find smallest integer $m \geq 0$ such that $\lambda = \beta^m$ and (1.6) is satisfied.

   (c.) $x = x + \lambda d$

2. *If $k = kmax$ and termination failed then signal failure.*

1.7 Problems with Steepest Descent

The method above can be grouped under a category that contains all line search minimization strategies. In (1.4) we defined a general descent direction d, and so we will define a general line search method similar to Algorithm 1. The line search method will use the general descent direction d in place of $-\nabla f(x)$. We will alter our sufficient decrease condition to be

$$f(x_c + \lambda d) - f(x_c) < \alpha\lambda \nabla f(x_c)^T d. \tag{1.7}$$

It is possible that using this condition we will fail to get a sufficient decrease, as we have defined it. If that occurs, then we will simply terminate the iteration and signal failure. In order to do this, we must put an upper bound on m, $mMax$. If $m \geq mMax$ then we terminate and signal that the line search failed.

The technique we used in *Steep* will be the technique that we use for the generalized line search stategy. There are additional problems, though. To begin with, the reduction in the step size λ is very simple minded. Typically we set β equal to $1/10$, $1/2$, or something similar. However, if β is very large, then m will need to be very large in order to satisfy (1.7). If β is very small, then the sequence of iterates will not be efficient. But, there is a solution. By examining the previous step sizes, we can get bounds on the step size for the current iteration. We can also model

$$\xi(\lambda) = f(x_c + \lambda d)$$

with a polynomial. In combination with our established limits, our choice in λ will now be more efficient. For a description of possible polynomial models see [1]. We will not discuss it any further here, but step size control will be incorporated into the final algorithm. From this point on in the paper, we will just note that we have put thought into finding λ and we have found a suitable choice.

Our other problem lies in the way we will define and update our descent direction d. For the generalized line search stategy, the manner in which we define d from iteration to iteration is based upon the quadratic model of f,

$$m(x) = f(x_c) + \nabla f(x_c)^T(x - x_c) + \frac{1}{2}(x - x_c)^T H_c(x - x_c).$$

H_c is called the model Hessian. We use the underscore c to mark the fact that the model Hessian depends upon our current point x_c. By taking the gradient of this model and setting it equal to zero, we get

$$\nabla m(x) = \nabla f(x_c) + H_c(x - x_c) = 0,$$

and with $d = x - x_c$,

$$d = -H_c^{-1}\nabla f(x_c). \tag{1.8}$$

However, we are not guaranteed that d will be a descent direction according to (1.8) unless H_c is spd. This is the problem mentioned above. We need to somehow ensure the positive-definiteness of the model Hessian. To do this, we will use one of two methods, BFGS or SR1. These methods will be expounded in the next section.

1.8 Quadratic Model Hessian

We established the need for using a model Hessian in (1.8). After we calculate the first model Hessian though, we do not want to continue doing so for each new x_c. The cost is simply too large. Instead, we would prefer to use an update with a lower cost, while guaranteeing that the updated version will be spd.

The first of the two update methods is BFGS. For this method, we have a developed local theory which guarantees convergence for accurate initial approximations of x^* and $\nabla^2 f(x^*)$. For these initial conditions, convergence will be q-superlinear. The update for this method goes as follows. First, let us define $s = x_+ - x_c$ and $y = \nabla f(x_+) - \nabla f(x_c)$. Then the BFGS update is

$$H_+ = H_c + \frac{yy^T}{y^T s} - \frac{(H_c s)(H_c s)^T}{s^T H_c s}. \tag{1.9}$$

The other update method is SR1. We can define this method similarly, but we must keep in mind that SR1 does not have any local theory for convergence. Let y and s be defined as above. Then the SR1 update is

$$H_+ = H_c + \frac{(y - H_c s)(y - H_c s)^T}{(y - H_c s)^T s}. \tag{1.10}$$

With both of the update techniques, we can rewrite the Hessian update to update the inverse Hessian instead of the Hessian itself. In (1.8) we wrote d in terms of the inverse Hessian. Therefore, defining the updates in this way might seem natural. The strategy used will be important, and we will often refer to the line search according to what type of update is used.

1.9 General Line Search Method

Now that we have established some basic criteria for building a generalized line search, we can construct an algorithm based on those ideas. The following algorithm uses those ideas and the structure of Algorithm 1.

Before we begin, we must decide which Hessian Update Strategy we are using. With that accomplished we can proceed.

Algorithm 2 *General(f,$x_c = x_0$,$kmax$)*

1. *Compute* $f(x_c)$, $\nabla f(x_c)$, *and* H_c.
2. *For* $k = 1, \ldots, kmax$
   (a.) *Find a descent direction* $d = -H_c^{-1}\nabla f(x_c)$.
   (b.) *Set* $x_+ = x_c + \lambda d$, *where we find* λ *in an effecient manner and we ensure sufficient decrease in the function* f. *If we do not get sufficient decrease, terminate and signal failure.*
   (c.) *Compute* $f(x_+)$ *and* $\nabla f(x_+)$
   (d.) *Check for termination using (1.3).*
   (e.) *Find* H_+ *using* x_c, x_+, *and* H_c.
3. *If* $k = kmax$ *and termination failed then signal failure.*

Chapter 2

2.1 Noisy Problems

In Chapter 1 we attempted to lay the framework for some optimization problems. We did not concern ourselves with how $\nabla f(x)$ was calculated or how we originally found H_c. This chapter will deal with those issues in more detail.

To begin, let us take note of a problem that would be difficult to solve using the ideas that we have put forth thus far. If we wanted Algorithm 2 to solve

$$f(x) = x^2,$$

then we should converge to the correct answer in one iteration. It really does not matter what we use as our initial point because everything is well behaved. However this problem quickly becomes complicated if we introduce random pertubations into the function. Take $f(x)$, where $f(x)$ is shown in Figure 2.1. The difficulty here is easy to see. There is one global minimum, $x^* = 0$, and several local minima. If we began Algorithm 2 at any point that was not very close to the origin, we would quickly find ourselves at the bottom of one of the small wells, unable to get out. Our algorithm does not take this into consideration.

We need to rethink our results from before. Over the course of this chapter, that will be our main concern. First though, we must redefine the problem.

In the example above, we began with a smooth problem, and then we added random perturbations into it. This noise had a low amplitude and high frequency that came on top of the original function. That will be the general formula for our problem here. Let us start with a smooth function

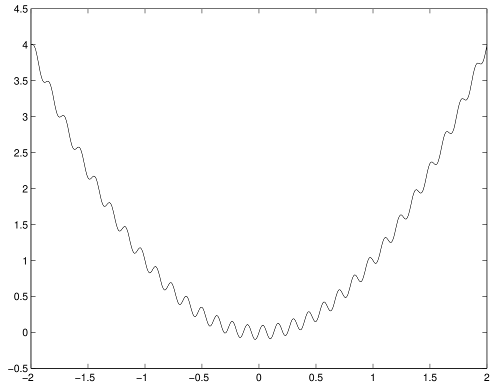

Figure 2.1: $f(x) = f_s(x) + \phi(x)$

$f_s(x)$. Then we can take a function of noise $\phi(x)$ and add the two together to get our objective function $f(x)$.

$$f(x) = f_s(x) + \phi(x) \tag{2.1}$$

The function in Figure (2.1) had a smooth function $f_s(x) = x^2$ and a noise pattern of $\phi(x) = (.1)\sin(15\pi x)$.

Ideally, when we optimize $f(x)$ we would hope to find the global minimum at $x^* = 0$. Therefore, hopefully, we would be optimizing the smooth part and forgetting about the noise. Unless we expressly know the pattern of noise though, we will not be able to separate the two. Over the next few sections we will address this issue.

2.2 Difference Gradients

To partially get around this problem we can take advantage of the way we handle difference gradients. Unless we know the derivative of the function explicitly, we will have to find some way to approximate it. In order to do this at x_c, we employ a difference gradient. Difference gradients find the derivative of a function at a point by exploiting the definition of a derivative. The definition states that the derivative is the change in the function value over a small change in x. This is almost explicitly how difference gradients are found.

When we compute a difference gradient, we have our choice of three different ways to go about it. The first is a centered difference gradient.

$$\nabla_h f(x) = \frac{f(x+h) - f(x-h)}{2h}. \tag{2.2}$$

If $x - h$ is not in the domain, D, of possible values for x, then we will have to use a forward difference gradient:

$$\nabla_h f(x) = \frac{f(x+h) - f(x)}{h}. \tag{2.3}$$

Likewise, if $x + h$ is not in the domain, then we will have to use a backward difference gradient:

$$\nabla_h f(x) = \frac{f(x-h) - f(x)}{-h}. \tag{2.4}$$

When we have more than one dimension, we will define the ith component of the gradient, $(\nabla_h f(x))_i$, with the following, which is similar to (2.2),

$$(\nabla_h f(x))_i = \frac{f(x + h * e_i) - f(x - h * e_i)}{2h}. \tag{2.5}$$

where e_j is the unit vector of the jth component. The forward and backward versions would be similar.

2.3 The Size of h

Now that we have defined a difference gradient, we must select a value for h that is appropriate. Of course, if h is large, then the gradient that we get out may be of little value. Also, h cannot be too small or the machine precision of the computer being used will weigh into the value of gradient. However, by selecting h correctly we can hopefully minimize the effect of the noise.

Let us use $f(x)$ from Figure (2.1) as an example. If we start with an arbitrary value of x, say $x_0 = -.95$, we will have a point that is somewhere on one of the small wells. By selecting $h = 1.0 * 10^{-6}$ we will not get that far. In fact, our gradient will be pointed in the positive x direction. If we move in the direction of $-\nabla f(x)$ we will be led into one of the smaller wells.

If instead, again starting with $x_0 = -.95$, we choose h to be larger, e.g. $h = .25$, then we will make it closer to the global minimum. Suppose we take a forward difference gradient. Since $x_0 + h = -.70$, our function value

$f(x_0 + h)$ will be smaller than $f(x_0)$. This in turn will point our difference gradient in the negative x direction and following $-\nabla f(x)$ we will move in the direction of the global minimum. Thus we will end up closer to $x^* = 0$ than we would with x_0 and $h = 1.0 * 10^{-6}$.

Again we are faced with a problem. Once we get close to $x^* = 0$ our value of h will be no good. If we try to take a difference gradient with $h = .25$ and $x_c = .05$, we will have difficulty. A forward difference will point the gradient in one direction, and a backwards difference gradient will point in the other. In this instance, we would prefer to have $h = 1.0 * 10^{-6}$. With this h, $-\nabla f(x)$ will point in the direction of x^* and we will move in the direction of the global minimum.

How do we make the decision of what h to pick? The example we just worked through shows us that we cannot just choose a constant h. Instead it would probably work better to start with large h values and then use smaller a h as the iteration progresses. We need a strategy for changing h that uses this type of thinking.

2.4 Strategy Involving h

A simple and sensible strategy for picking h would be to start with an h that is large. By large, we mean that our goal is to have an h such that a step $x_c + h$ will allow us to step over any noise present. For example, for our f in Figure 2.1 we had noise $\phi(x)$ with a period of $2/15$. When we picked $h = .25$, $h > 2/15$, we were able to see the downward trend in $f_s(x)$ around $x_0 = -.95$ through the presence of the noise. This problem was artificial in several ways. In general, a pattern of noise does not have a regular period that we can easily pick out, and the smooth part of the function may not be so easy to distinguish from the noise. But by using this thought process, we were at least able to start moving in the correct direction.

We hope this large value of h will get us close to the global minimum. If it has, we would like to reduce h and then use this new h to continue getting closer to the answer. However, how will we know when we should trigger a change in h? What is a good choice in the initial h and then the subsequent choices?

Let us begin with the last question. We want our first choice in h to be large enough to minimize the effect of the noise. In general, we want to have a sequence $\{h_k\}_{k=0}^{\infty}$ where $h_k \to 0$. We could simply choose $h_0 = 1$ and then

$$h_{k+1} = h_k * (.5). \tag{2.6}$$

As we have defined it here, this is an infinite sequence. At some point, the machine precision of the computer we are using will come into play. Therefore we generally stop (2.6) when $h_k < 1.0 * 10^{-8}$ or some other arbitrary value.

Now that we have a sequence $\{h_k\}$ we can answer our first questions. What will signal a change in h? We will simply use our default termination criteria. First we will set $h = h_k$ for some k. Using this value for h we proceed with the iteration until the iteration terminates. This occurs when the line search fails, too many iterations have been taken, or when (1.3) is signaled. Since we are using gradients calculated with differences, we ought to change (1.3) to reflect the change. Our new termination criterion will be

$$\|\nabla_h f(x)\| \leq \tau h \tag{2.7}$$

for some $\tau > 0$. Therefore we will signal the need for a new h if (2.7) is satisfied, too many iterations occur, or the line search fails.

Chapter 3

3.1 Implicit Filtering

Implicit Filtering, in a very simple form, is based on the steepest descent algorithm using difference gradients [1]. The core idea behind Implicit Filtering, though, is our gradual reduction in h as the iteration progresses. We could pair this with idea with a Quasi-Newton iteration, and we would still have implicit filtering. However, our treatment here deals only with implicit filtering as tied to the steepest descent algorithm.

The algorithms we have here for Implicit Filtering are indeed based on Algorithm 1. However, instead of using (1.3) for our termination criterion, we will use (2.7). Our sufficient decrease condition will be the same as (1.6), but we will use $\nabla_h f(x)$ in place of $\nabla f(x)$. Also note that whenever possible we will use centered difference gradients. Finally, we will use the sequence h_k as defined in (2.6).

Therefore we can easily build an Implicit Filtering Algorithm based on Algorithm 1. First we begin by taking the Steepest Descent algorithm and implementing the changes we have discussed.

Algorithm 3 *NewSteep*$(x, f, h, kmax, \tau)$

1. *For* $k = 1, \ldots, kmax$
   - (a.) *Compute* f *and* $d = -\nabla_h f$*; terminate if (2.7) is satified.*
   - (b.) *Find* λ *in some way that satisfies (1.6), the sufficient decrease condition. If line search fails, terminate.*
   - (c.) $x = x + \lambda d$

We have basically just added difference gradients to Algorithm 1 and we also found λ in a thoughtful manner. Now we can incorporate our changes in h with our Implicit Filtering Algorithm.

Algorithm 4 *Implicit1($x, f, kmax, \tau$)*

For $i = 0, \dots$

1. *Let $h = h_i$, and terminate if $h < 1.0 * 10^{-8}$.*
2. *NewSteep($x, f, h, kmax, \tau$)*

Written in this way, we see that we get a new value for h when $NewSteep$ terminates in some way. As we said before, this happens when we have taken too many iterations with the old h, when the line search fails, or when the termination criterion is met. Algorithm 4 will terminate on a whole, once h_k has reached a significantly small value.

We could just as easily have built Implicit1 based upon Algorithm 2. Doing that gives us the following algorithm. First we pick whether to do BFGS or SR1 updates.

Algorithm 5 *NewDescent($x, f, H, h, kmax, mMax, \tau$)*

1. *Compute $f(x_c)$, $\nabla_h f(x_c)$.*
2. *For $k = 1, \dots, kmax$*
   - *(a.) Find a descent direction $d = -H_c^{-1}\nabla_h f(x_c)$.*
   - *(b.) Set $x_+ = x_c + \lambda d$, where we find λ in an effecient manner and we ensure sufficient decrease in the function f. If we do not get sufficient decrease with $m < mMax$, terminate.*
   - *(c.) Compute $f(x_+)$ and $\nabla_h f(x_+)$*
   - *(d.) Check for termination using (2.7).*
   - *(e.) Find H_+ using x_c, x_+, and H_c.*

Again, we can incorporate our sequence h_k into the scheme.

Algorithm 6 *Implicit2($x, f, kmax, mMax, \tau$)*

1. *H=I*
2. *For $i = 0, \dots$*
   - *(a) Let $h = h_i$, and terminate if $h < 1.0 * 10^{-8}$.*
   - *(b) NewDescent($x, f, H, h, kmax, mMax, \tau$)*

3.2 IFFCO

$IFFCO$ is the implicit filtering algrithm that we implemented in this project. The idea behind $IFFCO$ is very close to Algorithm 6.

The full code for $IFFCO$ was written by Paul Gilmore and Tim Kelley. It was further modified by Tony Choi.

Chapter 4

Now that we have developed the basic ideas behind Implicit Filtering and introduced IFFCO, we can commence with a description of the project in which I was involved. I will often refer back to the ideas that we developed in the first couple of chapters.

4.1 The Automotive Valve Train Problem

We applied $IFFCO$ to a joint project between a group of automotive engineers and our group of applied mathematicians at North Carolina State University. The goal of the project was to model the valve train of a race car engine that ran at very high RPMs and to then take that model and redesign the cam shaft lobe.

The engineers in the project had developed a two dimensional model for one of the pistons inside the engine. The idea behind the model is straightforward. As the piston head moves up and down, the valve at the top of the piston opens and closes to allow the inflow of fuel before compression, and then the outflow of gas after ignition. At very high engine speeds, the valve is moving so quickly that it does not fully seat itself when it returns to a closed position. Instead, on contact with the valve seat, the valve bounces. This bounce allows gas to escape, and decreases the efficiency of the engine. We hoped that by reducing the valve bounce in the engine, we could improve the overall efficiency of the engine itself. In order to do this, we needed to do two things. The first was to model the engine, and the second was to then change the design of the cam lobe to see how the chages affected the performance of the engine. My participation in the project was solely in the first part of this process. In particular, I ran the code and provided information for Dr.

Kelley and Tony Choi.

The model used for the engine was a two dimensional model. The three dimensional model was shown be to equivalent to the two dimensional one for our purposes. There were were several different models for the different engines, a Chevrolet block, a Ford block, etc. With each model, there were generally more than ten variables. Some of these variables had physical limitations to their size, e.g. the constant of the spring used to return the valve to the valve seat. In addition, not all of the variables within the model could be directly measured or they may have be subject to change due to heat, or some other factor. Therefore, there are several parameters within the engine model that we must approximate using a least squares fit to the data that we did measure. In the end, we had a large, non-linear least squares problem. To find the solution, we used $IFFCO$.

The size of the problem alone begged the need for an efficient way to go about solving it. On a cluster of workstations with Intel chips, the simplest of the engine blocks could take on the order of days to weeks to solve. For this reason we used our implicit filtering algorithm, $IFFCO$, on the Cray T3E at the North Carolina Supercomputing Center. Solution times on the T3E were generally on the order of hours.

4.2 IFFCO and the Valve Train

In order to use $IFFCO$ on this problem, it was first necessary to link the model for the engine with $IFFCO$ itself. As I joined the project after it had begun, this part was done mainly by Tony Choi.

This is the point at which I became involved. $IFFCO$ was integrated into the engineer's code and we had started running the combined code on the T3E. As we saw in the first couple of chapters, there are a number of decisions that go into simply running the code. For instance, we needed to set the minimum value of the scale h, call it $hmin$. We also needed to decide on the maximum number of cutbacks in the line search, $mMax$, and if we intended on using the quadratic model Hessian, we needed to decide which Hessian Update strategy we were going to use, $BFGS$ or $SR1$.

Why are these types of decisions important? If we run the code and perform 200 function evaluations, the time elapsed will be on the order of an hour. When we set $hmin$ too low, we will not make any progress on the iteration and we will not get any closer to the answer. An example of $hmin$

set too low can be seen in the function in Figure 2.1. If we start with a large value for h and slowly reduce it, as we dictated in our implicit filtering algorithm, we will eventually be close to the minimum. We will be in one of the small wells near $x^* = 0$. Once we are in a small well, for example with $h = 1.0 * 10^{-6}$, it will not benefit us at all to reduce h further. Doing so would not improve the returned value, but it would increase the number of function evaluations taken over the course of the iteration. At one point, we will be fighting the noise in the funtion, and we will not win. Therefore, it was wise to put thought into the code we were running.

With a problem such as the automotive valve train, time becomes an important constraint. As we saw above, there are subtle things that we can do to the problem to improve performance that are not laid out in the theory we discussed in the first three chapters. Over the last year, I have been seeking such subtlties while I attempted to make $IFFCO$ run efficiently for this valve train problem.

4.3 Improving the Runs

In this section, I hope to give a general outline of the runs that I performed and the improvements that they made. Let us begin with the default value for a couple of the parameters and the outcome. I have graphed the valve bounce curve that resulted from the returned point from one run of $IFFCO$ in Figure 4.1. The default values for $IFFCO$ are in the following table, corresponding to Figure 4.1.

Filename	fscale	termTol	minh	fValue	symbol
D13_0729_1.log	1.0	1.00	$1.e-3$	$0.7101E-1$	x

The solid line is the experimental valve bounce curve. On this graph, the x-axis is the engine speed and the y-axis is the valve bounce. Our best results can be seen in Figure 4.4, which was the end result of my work.

As we can see from the run of the defaults, we were not successful in finding the variables in question. This was in part due to the parameters that we gave to $IFFCO$. In particular, our $minh$ was too large and so was $termTol$. The variable $fscale$ also became a factor in the iteration.

However, let us start with our discussion of $termTol$ and $minh$. We begin by setting $fscale = 7.5$. The variable $fscale$ is used to scale the function

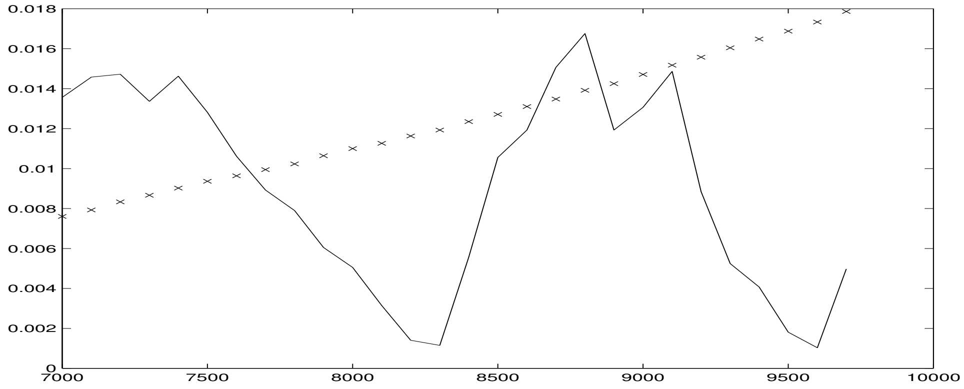

Figure 4.1: Default Values v. Experimental

values, and it should be set to the largest function value obtained over the course of the iteration. The largest function value we found was close to 7.5, and hence the setting. Now, with that done, we varied $termTol$ to see how this affected the iteration.

Name	fscale	termTol	minh	fValue	symbol
D13_0729_4.log	7.5	1.00	$1.e-3$	$1.269E-1$	x
D13_0730_5.log	7.5	.100	$1.e-3$	$0.2339E-1$	*
D13_0730_4.log	7.5	.010	$1.e-3$	$0.1603E-1$	+
D13_0730_6.log	7.5	.001	$1.e-3$	$0.1558E-1$	o

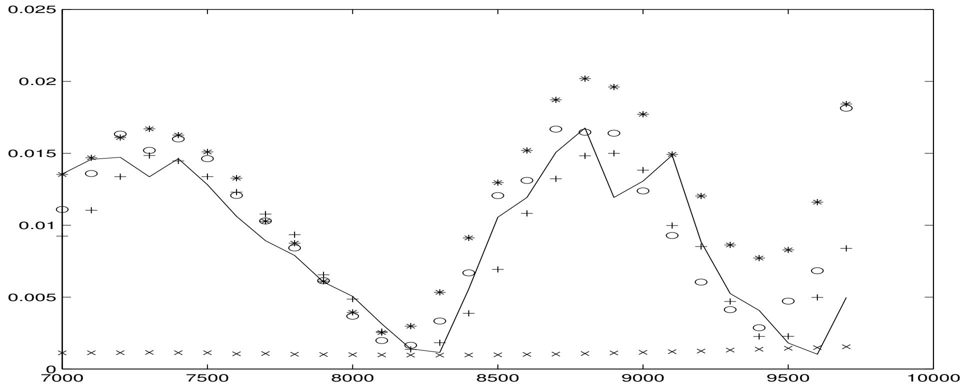

Figure 4.2: Variations in $termTol$ v. Experimental

As we can see in Figure 4.2 as the value of $termTol$ decreased, we were able to get a better and better fit to the experimental data. Within our implicit filtering code, $termTol$ is equivalent to τ in (2.7) which was part of the termination criteria. Therefore, this result makes sense, because with each reduction in $termTol$ we are asking $IFFCO$ to reduce $\|\nabla_h f(x)\|$ before changing h. Further reductions in $termTol$ would have traded a little bit of improvement for many more function evaluations, and so we decided that $termTol \leq .010$ would be an adequate setting for our purposes.

We did something similar with our setting of $minh$, which is the minimum value of the scale used in the iteration before the whole process terminates. The results of our varying $hmin$ were not as noticible as the above example. The reason for this is probably best understood if we refer back to our example in Figure 2.1. In that example, we saw how a reduction in h would allow us to initially jump over the noise and to then zero in on the minimum using a small h. However, we saw with that example that once we were in the correct well near the answer, we were no longer stepping over noise. Instead we were driving ourselves to the minimum. At that point, a reduction in h made little difference, as we were already close to the answer. We are seeing the same effect here with the valve train. Once $h = O(10^{-3})$ we have made it past all of the noise and further reductions would be of little value. Therefore, from this point out we used $hmin = 10^{-3}$.

Another noticible parameter that affected $IFFCO$ was the value of $fscale$. Again $fscale$ was the parameter that we used to scale the returned function values. Here again there is a great variety in the quality of answer that we get back.

Name	fscale	termTol	minh	fValue	symbol
D13_0730_2.log	1.0	.010	$1.e-3$	$0.5102E-1$	x
D13_0730_4.log	7.5	.010	$1.e-3$	$0.1603E-1$	+
D13_0731_4.log	10.	.010	$1.e-3$	$0.16022E-1$	*
D13_0804_1.log	50.	.010	$1.e-3$	$0.3446E-1$	o

As we mentioned before, the value should be set to 7.5, since this is close to the largest function the iteration reaches. However, if we look at the other values, then it seems interesting that changing the variable by a factor of ten has such an effect. This type of variation in answers was true no matter what the settings for $minh$ and $termTol$ were. Although I am not sure as to the

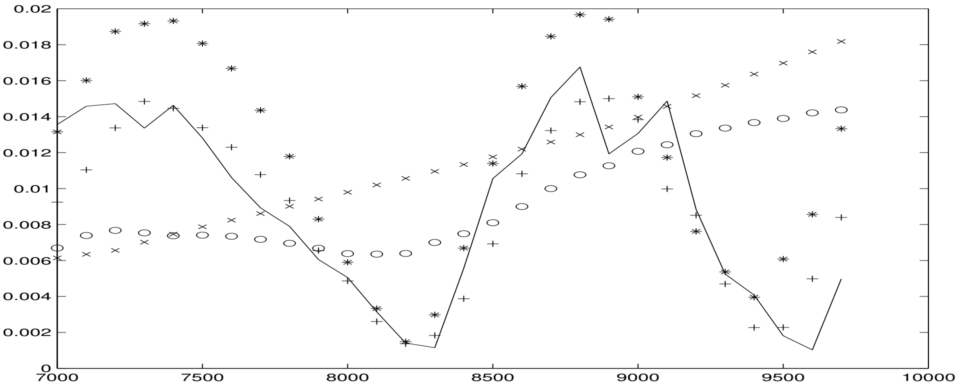

Figure 4.3: Varying $fscale$ v. Experimental

exact reason the effect was so pronounced, I simply noted that the scaling was important for the problem, and I moved on.

After doing the same type of analysis for several parameters in $IFFCO$, we found a general setting which we used from that point out. In terms of the variable that we have discussed, the settings were as follows.

Name	fscale	termTol	minh	fValue	symbol
D13_0730_6.log	7.5	.001	$1.e-3$	$0.1558E-1$	x

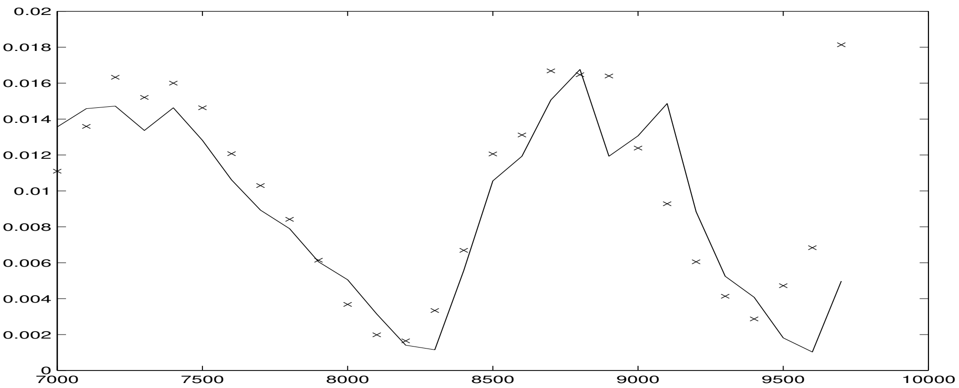

Figure 4.4: Best Results v. Experimental

4.4 Other Implicit Filtering Options

The other options that go along with $IFFCO$ include whether or not to do restarts, whether to use a BFGS or SR1 update for the approximate inverse Hessian B, whether or not to reinitiallize B, and also when to do that.

We start with restarts. A restart simply begins with the best point found in a run and restarts $IFFCO$. We saw in our example with Figure 2.1, if we get caught in one of the local wells, a restart could allow us to find the correct answer. As we can see from the following table, performing a restart with this problem was ineffective.

Name of Run		No. of Evals.	Obj. Func. Value	% of Original
D13_0731_2.log	Initial Point	1	$0.2635E+0$	100
	Before Restart	907	$0.5018E-01$	19.05
	After 1 Restart	1291	$0.4919E-01$	18.67

We obtained similar results with other runs involving restarts. If the extra .38% is important for the function value, a restart makes sense. However, in our case the added benefit was not worth the extra number of function evaluations.

Next I went on to comparing BFGS and SR1 updates of the approximate inverse Hessian. As we mentioned before with BFGS, we have local theory regarding convergence, which is not the case with SR1. We found though that BFGS did not converge as well as SR1. (I used SR1 for the runs in the last section.) With our tests comparing BFGS and SR1 we incorporated another option, when to reinitiallize B. This was done with a parameter we called $option(7)$. The settings of $option(7)$ can be summerized as follows.

option(7) = Quasi-Newton variation
= 0 Re-initialize B at each new scale.
= 1 Re-initialize B if active set changes.
= 2 Update Hessian even if active set changes (see (2.10) in paper)

So we ran the code using different settings for $option(7)$.

For SR1 we had found $option(7) = 1$ to work the best. In particular we obtained this table (using the settings found in the last section).

Name	Quasi	option(7)	Func evals	fValue	symbol
D13_1028_1.log	SR1	0	790	$0.8044E-02$	o
D13_1026_2.log	SR1	1	593	$0.8043E-02$	x
D13_1028_1.log	SR1	2	822	$0.8184E-02$	*

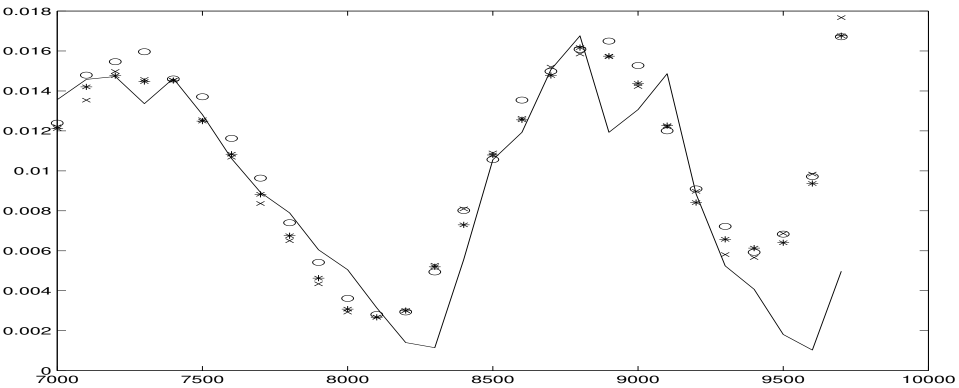

Figure 4.5: SR1 runs v. Experimental

In general BFGS gave answers that we not quite as good, but with fewer function evaluations. The number of function evaluations was generally about 10% fewer. With BFGS we had the following results.

Name	Quasi	option(7)	Func evals	fValue
D13_1029_2.log	BFGS	0	714	$0.8354E-02$
D13_1026_3.log	BFGS	1	436	$0.8397E-02$
D13_1026_1.log	BFGS	2	798	$0.7935E-02$

We judged the best runs as the following three.

Name	Quasi	option(7)	Func evals	fValue	symbol
D13_1026_2.log	SR1	1	593	$0.8043E-02$	o
D13_1026_3.log	BFGS	1	436	$0.8397E-02$	x
D13_1026_1.log	BFGS	2	798	$0.7935E-02$	*

In order to choose the "best solutions" we had to make decisions. I chose the two runs that gave a reasonable answer in the fewest number of function evaluations and also the run that gave the best answer.

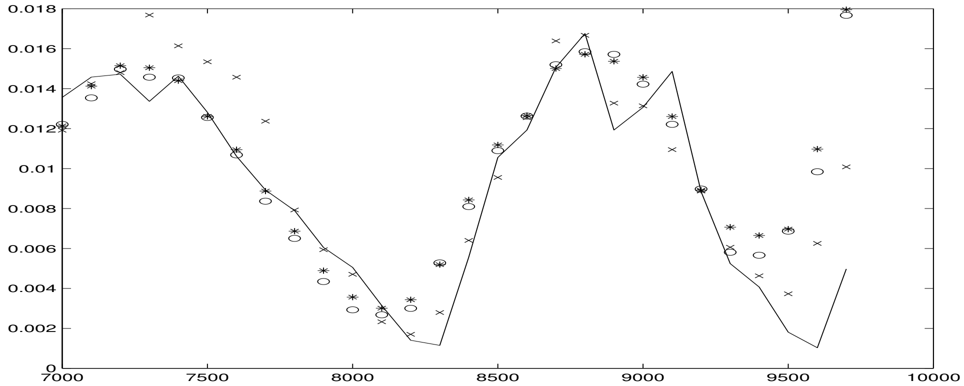

Figure 4.6: Best runs v. Experimental

After doing all of this it was important to ask what we expected in an answer. Was a good answer a fast one or one with the lowest possible function value? These questions ultimately would decide which of the above settings is the best for this problem. However, I was not the one that could make such decisions. Instead, my participation in this project provided the information that would allow those types of decisions to be made.

4.5 Other Problems

The race car problem eventually reached its conclusion, and we moved on to a new problem. Again we are applying $IFFCO$ to this problem, and we are again making decision about when to terminate and what settings we should use.

Bibliography

[1] C. T. Kelley, *Iterative Methods for Optimization,* SIAM, Philadelphia, 1999.

[2] C. T. Kelley, *Iterative Methods for Linear and Nonlinear Equations,* SIAM, Philadelphia, 1995.