Implicit Filtering for Constrained Optimization and Applications to Problems in the Natural Gas Pipeline Industry [1]

Alton Patrick
Department of Mathematics
Center for Research in Scientific Computation
North Carolina State University
Advisor: Tim Kelley

April 26, 2000

[1]Work supported by the National Science Foundation

Contents

Chapter 1

Introduction to Optimization

The goal of an optimization problem is to find a point x^* in the domain of the *objective function* f such that $f(x^*)$ is "optimal." "Optimal" can mean either a minimum or maximum of f. However, a maximum of f is a minimum of $-f$, so we can express every optimization problem as a search for a minimum of some f. Therefore, the optimal point x^* is given by

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

where D is the set of points near x^* and f is the objective function. If D is bounded in some way, we call (1.1) a constrained optimization problem. If D is $\mathcal{R}^n$, (1.1) is an unconstrained optimization problem.

1.1 Basic Solution Strategy

Let's consider a very simple optimization problem, with $f(x) = x^2$. We will call our first guess at a solution x_0. Subsequent (hopefully educated!) guesses at the solution will define a sequence of iterates, $\{x_k\}, k \geq 0$.

Clearly f has a minimum at $x^* = 0$. The solution to a serious optimization will rarely be so obvious. The initial iterate x_0 is usually a genuine guess, perhaps based upon a graph of f showing the approximate position of the solution, perhaps based upon nothing at all. Let's pretend we are ignorant about the location of the minimum and choose $x_0 = 2$.

Now we need a way to get from x_0 to x_1. We would like to move towards the minimum, so it would be good if $f(x_1) < f(x_0)$. The derivative of f at x_0, $f'(x_0)$, tells us which way to go from x_0 to reduce the objective function. If $f'(x_0)$ is negative, moving to the right will reduce the function value. If $f'(x_0)$ is positive, moving to the left will reduce the function value. In general, moving along the x-axis in the direction of $-f'(x_0)$ will reduce the objective function. $-f'(x_0)$ is therefore called a *descent direction*.

How far should we move in this descent direction? The simplest thing would be to set

$$x_1 = x_0 - f'(x_0) = 2 - 2 \cdot 2 = -2$$

This is unproductive, because $f(x_0) = f(x_1)$ and we are no better off than when we started. The solution is to multiply $f'(x_0)$ by a factor $\lambda \in (0, 1]$. λ is called the *step length*. Then x_1 is given by

$$x_1 = x_0 - \lambda f'(x_0)$$

We can try different values of λ, starting with $\lambda = 1$, until we find an x_1 that produces an acceptable value of $f(x_1)$. This process is called a *linesearch*.

Two questions immediately arise:

1. "What sequence of values should be used for λ?" and

2. "What is an 'acceptable' value for $f(x_1)$?"

Often, the first question is answered by taking

$$\lambda_m = \beta^m, m = 0, 1, 2, \ldots \tag{1.2}$$

where $0 < \beta < 1$. More sophisticated ways to choose a sequence of λ's will be discussed later.

Answering the second question requires that we define the concept of *sufficient decrease*. This means establishing a criteria to check that $f(x_1)$ is "sufficiently" smaller than $f(x_0)$. One such criteria is $f(x_1) < f(x_0)$. This is called *simple decrease*. Using the simple decrease condition, we would take as the step length the first λ_i such that

$$f(x_0 - \lambda_i f'(x_0)) < f(x_0)$$

Unfortunately, simple decrease is usually too simple. It can lead to taking very small steps that cause the iteration to stagnate at one point. We will use a more complicated idea of sufficient decrease called the Armijo rule. This requires that

$$f(x_0 - \lambda f'(x_0)) - f(x_0) < -\alpha\lambda|f'(x_0)|^2$$

where $0 < \alpha < 1$. Typically $\alpha = 10^{-4}$.

So far we have looked at a problem only in one dimension, to make the basic solution plan easier to follow. We want to be able to solve problems where $x \in \mathcal{R}^n$. Generalizing what we have done so far to n dimensions is fairly straightforward. The N-dimensional equivalent of the derivative is the gradient,

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_N} \right).$$

$-\nabla f$ gives a descent direction for f in N dimensions just as f' does in one dimension. The Armijo rule for sufficient decrease in N dimensions is

$$f(x_c - \lambda \nabla f(x_c)) - f(x_c) < -\alpha\lambda\|\nabla f(x_c)\|^2. \tag{1.3}$$

Using the gradient, we can generalize the above method for use in N dimensions. We will also introduce the notation x_c to represent the current point in the iteration and x_+ to represent the next point in the iteration. This yields the *Method of Steepest Descent*, given here as Algorithm 1. A few refinements that improve Steepest Descent will be introduced in the following sections.

Algorithm 1 Steepest Descent

Pick x_0.
Set $x_c = x_0$
while $f(x_c)$ is not the minimum of f **do**
  Find λ such that (1.3) is satisfied.
  Set $x_+ = x_c - \lambda \nabla f(x_c)$
  Set $x_c = x_+$.
end while

1.2 Termination Criteria

One issue Algorithm 1 dodges is how to know when $f(x_c)$ is the minimum of f. Two theorems will give us necessary and sufficient conditions to know that x_c is an optimal point. For proofs of these theorems, see [6].

Before giving those theorems, we must define some more terms. We let $\nabla^2 f(x)$ represent the Hessian of f, where

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

We also define the terms positive semi-definite, positive definite, and symmetric positive definite (spd).

Definition 1 *A matrix A is positive semi-definite if $x^T A x \geq 0, \forall x \in \mathcal{R}^N$. A is positive definite if $x^T A x > 0, \forall x \in \mathcal{R}^N, x \neq 0$. 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.*

Now we can present the two theorems which give the termination criteria for Algorithm 1.

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.*

From these two theorems, it is clear that we would like to stop when $\nabla f(x_c) = 0$. However, it is very unlikely that we will be lucky enough to land *exactly* on the minimum. Even if we do, it might take a very large number of iterations, many of which would only move x_c a very small distance. So we will simply stop when $\nabla f(x_c)$ is "close" to zero. We can use $\|\nabla f(x_c)\|$, the norm of $\nabla f(x_c)$, to measure how close to zero $\nabla f(x_c)$ is. One way to check that the gradient is "close" to zero is to see if $\|\nabla f(x_c)\|$ is significantly smaller than $\|\nabla f(x_0)\|$, i.e.

$$\|\nabla f(x_c)\| < \tau_r \|\nabla f(x_0)\|$$

where $0 < \tau_r < 1$. However, if $\|\nabla f(x_0)\|$ is very small, it may not be possible to satisfy this condition with floating point numbers. So we add an absolute tolerance, $0 < \tau_a < 1$:

$$\|\nabla f(x_c)\| < \tau_r \|\nabla f(x_0)\| + \tau_a. \tag{1.4}$$

This algorithm might fail to find a minimum or take a very long time to find a minimum. Therefore, it is useful to limt the number of iterations the algorithm will go through. From now on, our algorithms will incorporate a limit, $kmax$, on the number of iterations.

1.3 Refining the Descent Direction

We have been using $-\nabla f(x_c)$ as the descent direction. The linesearch method becomes more general if we replace $-\nabla f(x_c)$ with a general descent direction, d. With this change, the sufficient decrease condition, 1.3, becomes

$$f(x_c + \lambda d) - f(x_c) < \alpha\lambda\nabla f(x_c)^T d. \tag{1.5}$$

We will consider descent directions based on *quadratic models* of f. Just as quadratic models of functions in $\mathcal{R}^1$ require information about the second derivative, quadratic models of functions int $\mathcal{R}^N$ require information about the Hessian. We will use models of the form

$$m(x) = f(x_c) + \nabla f(x_c)^T(x - x_c) + \frac{1}{2}(x - x_c)^T H_c(x - x_c) \tag{1.6}$$

where H_c is an approximation to the Hessian of f called the *model Hessian*. H_c is used instead of the actual Hessian because 1.) the Hessian of f may not be known or may be too costly to calculate; and 2.) H_c is required to be spd, a requirement that the actual Hessian may not meet. The reason for this requirement is discussed later.

Minimizing m is, in general, easier than minimizing f because we have a simple expression for m. Setting the gradient of m equal to zero gives

$$\nabla m(x^\dagger) = \nabla f(x_c) + H_c(x^\dagger - x_c) = 0 \tag{1.7}$$

where $x^\dagger$ is the minimum of f. Note that $x^\dagger$ is only a minimum of f if H_c is positive definite (see Theorem 2). We want to move from x_c to $x^\dagger$ because the minimum of f is likely to be close to the minimum of m. So we set $d = (x^\dagger - x_c)$, or, using (1.7)

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

(1.8) may not give a descent direction if H_c is not spd [6]. To insure H_c is spd, we usually let H_0 be a clearly spd matrix (such as the identity) and then use one of two methods which guarantee symmetric positive definiteness to update H_c as the iteration progresses. If we let $s = x_+ - x_c$ and $y = \nabla f(x_+) - \nabla f(x_c)$, then the first update formula, called 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 second update formula, SR1, is

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

Both of these techniques can be adapted to update the inverse Hessian instead of the Hessian itself. This is more useful since we only need H_c^{-1}, not H_c, to find d. Optimization methods that update an approximation to the Hessian as the iteration progresses are called *Quasi-Newton methods*.

1.4 Refining the linesearch

We can make the linesearch more efficient. Given enough information about the objective function, we can model

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

with a polynomial. Then, instead of using (1.2) to define λ, we can choose λ to minimize $\xi(\lambda)$. Several possible polynomial models are discussed in [6]. Here, it is enough to say that we will be using some type of polynomial model to determine the step length from now on.

We also need to modify the linesearch by putting a limit on the number of times λ is reduced. Because of the new sufficient decrease criterion, (1.5), it is possible that the linesearch will not find a point that meets sufficient decrease. Therefore, if the linesearch does not find an acceptable point after $mMax$ reductions in λ, the linesearch will indicate failure. How that failure is dealt with will be explained later.

Combining all these refinements to Algorithm 1 leads to Algorithm 2.

Algorithm 2 Modified Steepest Descent

Pick x_0.
Set $x_c = x_0$ and $k = 0$
Calculate $f(x_c)$ and $\nabla f(x_c)$.
while $\|\nabla f(x_c)\| \geq \tau_r \|\nabla f(x_0)\| + \tau_a$ and $k \leq kmax$ **do**
  Calculate the descent direction, $d = -H_c^{-1} \nabla f(x_c)$.
  for $m = 0 \ldots mMax$ **do**
    Let $\lambda = \lambda_m$.
    if λ satisfies (1.5) **then**
      Exit FOR loop.
    else if $m = mMax$ **then**
      Signal failure.
    end if
  end for
  Set $x_+ = x_c - \lambda d$.
  Calculate $f(x_c)$ and $\nabla f(x_c)$.
  Update the model Hessian H_c using either the BFGS (1.9) or SR1 (1.10) update.
  Set $x_c = x_+$.
  Set $k = k + 1$.
end while

Chapter 2

Noisy Problems and IFFCO

So far, I have dealt only with "smooth" problems. The methods we have described can fail if applied to *noisy problems*, problems with many small perturbations and many local minima. More specifically, what we will call noisy objective functions are made up of a smooth function plus a noise function of much smaller magnitude. We represent such a function by

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

where $f_s(x)$ is the underlying smooth function, and $\phi(x)$ is the low-amplitude noise function. $\phi(x)$ is a distraction to the real problem, which is to find a minimum of $f_s(x)$. We would like to filter out the noise and leave just $f_s(x)$. This would be easy if we knew $\phi(x)$ explicitly, but we do not.

In this chapter I will describe how to adapt the methods we have already developed to work on such functions by effectively filtering out the noise. This strategy is called *implicit filtering*. At the end of this chapter I will describe IFFCO, a code for solving noisy optimization problems using Implicit Filtering.

2.1 Difference Gradients

So far, we have used $\nabla f(x)$ often but ignored where it came from. Gradient information for the functions we are interested in optimizing is not usually available. Many of the functions come from complex models involving differential equations. They can be very costly or impossible to differentiate.

We approximate $\nabla f(x)$ with *difference gradients*, represented by $\nabla_h f(x)$. There are three types of difference gradients. The ith component of the centered difference gradient is

$$(\nabla_h f(x))_i = \frac{f(x + he_i) - f(x - he_i)}{2h}$$

where $(\nabla_h f(x))_i$ is the ith component of $\nabla_h f(x)$ and e_i is the unit vector in the direction of the ith component of x. If $x - he_i$ is not in the domain of $f(x)$, D, a forward difference gradient may be used:

$$(\nabla_h f(x))_i = \frac{f(x + he_i) - f(x)}{h}.$$

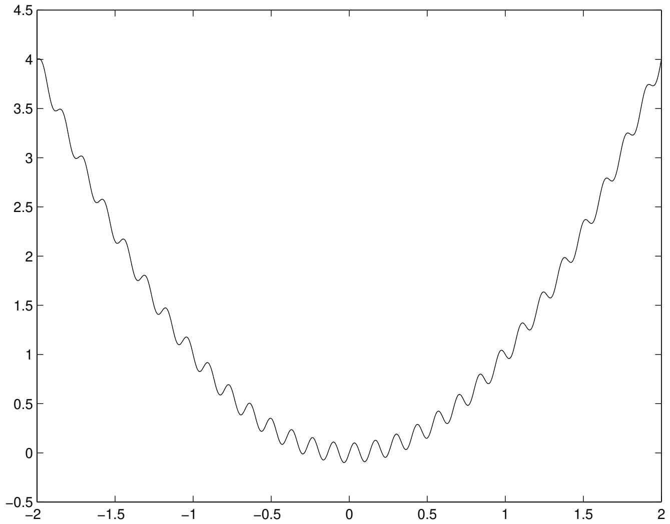

Figure 2.1: A noisy function

If $x + he_i$ is not in D, a backward difference gradient may be used:

$$(\nabla_h f(x))_i = \frac{f(x) - f(x - he_i)}{h}.$$

The centered difference gradient is the preferred method and is always used if possible.

2.2 The Size of h

Obviously the size of h is very important. As h becomes very small, the difference gradient approaches the actual gradient. If h is not small enough, the difference gradient will not give a very good estimate of the actual gradient of f.

Instead of being a problem, larger values of h are actually part of the solution to filtering low-amplitude noise out of noisy objective functions. Consider the following noisy function:

$$f(x) = x^2 + (.1)sin(15\pi x). \tag{2.2}$$

In this example, $f_s(x) = x^2$ and $\phi(x) = (.1)sin(15\pi x)$. Figure 2.1 is a graph of $f(x)$.

Because of the noise in this problem, accurate gradient information can lead our optimization algorithm the wrong way. If the initial iterate is on the wrong side of one of the small oscillations, the descent direction could very easily point away from the global minimum at $x^* = 0$. Even if the descent direction is the right one, the algorithm will likely become trapped in one of the local minima before it reaches $x^* = 0$.

However, if we use a difference gradient with a relatively large h, we can ignore the low-level noise. By using a grosser approximation to the gradient, we effectively filter out the relatively small effect of $\phi(x)$. Take, for example, $x_0 = -.95$ and $h = .25$. The centered difference gradient $\nabla_h f(x_0)$ is -2.1, which makes the descent direction positive and fairly large. This means the linesearch should take a good step towards the global minimum at $x = 0$.

Obviously, we will not get a very good solution if we leave $h = .25$ in subsequent iterations. It has served its purpose in letting the iteration move towards the global minimum, but once we get in the neighborhood of x^*, the large h will only cause problems. We would like to have an increasingly accurate gradient as the iteration progresses and (hopefully) gets closer and closer to the global minimum.

The solution is to let h go to zero as the iteration progresses. We choose a sequence of h_k, for instance,

$$h_0 = .5, h_{k+1} = \frac{h_k}{2}.$$

Implicit Filtering is an optimization algorithm for noisy functions based on the idea of filtering out noise by changing h as the iteration progresses. It is very similar to Algorithm 2 with difference gradients. h is reduced, i.e. h_k is replaced by h_{k+1}, when one of three things happens:

1. The linesearch fails.
2. More than $kmax$ iterations are taken.
3. The new termination criterion, based on the size of h, is met. This criterion is:

   $$\|\nabla_h f(x)\| \leq \tau h$$

   for some $\tau > 0$.

Algorithm 3 summarizes Implicit Filtering.

2.3 IFFCO

IFFCO (Implicit Filtering for Constrained Optimization) is the implicit filtering algorithm used for the work described in the rest of this paper. It is based on Algorithm 3 and implemented in Fortran. IFFCO was originally written by Paul Gilmore and Tim Kelley. It was later modified by Tony Choi and Owen Eslinger.

Algorithm 3 Implicit Filtering

```
Pick $x_0$.
Set $x_c = x_0$, $k = 0$, and $h_0 = .5$.
Calculate $f(x_c)$ and $\nabla_h f(x_c)$.
while $h \geq hMin$ do
  while $\|\nabla_h f(x_c)\| > \tau h$ and $k \leq kmax$ do
    Calculate the descent direction, $d = -H_c^{-1}\nabla_h f(x_c)$.
    for $m = 0 \ldots mMax$ do
      Let $\lambda = \lambda_m$.
      if $\lambda$ satisfies (1.5) then
        Exit for loop.
      else if $m = mMax$ then
        Signal failure; reduce $h$ and go to the next iteration of the outer while loop.
      end if
    end for
    Set $x_+ = x_c - \lambda d$.
    Calculate $f(x_c)$ and $\nabla_h f(x_c)$.
    Update the model Hessian $H_c$ using either the BFGS (1.9) or SR1 (1.10) update.
    Set $x_c = x_+$.
    Set $k = k + 1$.
  end while
  Set $h = \frac{h}{2}$.
end while
```

Chapter 3

Application of IFFCO to Optimization of Natural Gas Pipelines

The natural gas used in our homes and businesses is usually transported through a highly developed system of cross-country pipelines [7]. A schematic diagram of a simplified gas transmission network is shown in Figure 3.1. At points in the network, part of the gas (usually estimated at three to five percent [2]) is burned to drive engines at compressor stations. These compressors pressurize the gas in order to drive it through the pipeline. The cost of the gas used to power the compressors accounts for 25 to 50% of a typical gas transmission company's operating budget [7]. At 1998 U.S. prices, the cost of the gas burned in this fashion was about two billion dollars per year [2]!

It is not surprising, then, that a common optimization problem in the industry is, given the supply and delivery flows, to minimize the amount of gas burned by the compressors. I applied IFFCO to three problems of this type. Figures 3.2, 3.3, and 3.4 show the problems, which will be referred to, respectively, as Problems A, B, and C. The vertical axis is the amount of gas (in thousand cubic feet per day) used for fuel in a compressor station. The horizontal axes are two flow variables. The flow variables may be inlet or outlet flows or a Kirchoff Law representation of possible flow splits between alternative paths.

Evaluating the objective function for a combination of the flow variables involves optimizing pressure settings throughout the network to achieve those flows. This is accomplished by solving a large combinatoric problem using non-sequential dynamic programming [2]. For some flow settings, this problem does not have a solution and the objective function will fail to return a value.

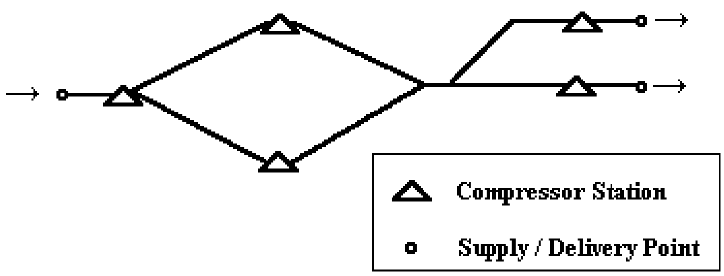

Figure 3.1: Schematic of a Simple Gas Transmission Network [7]

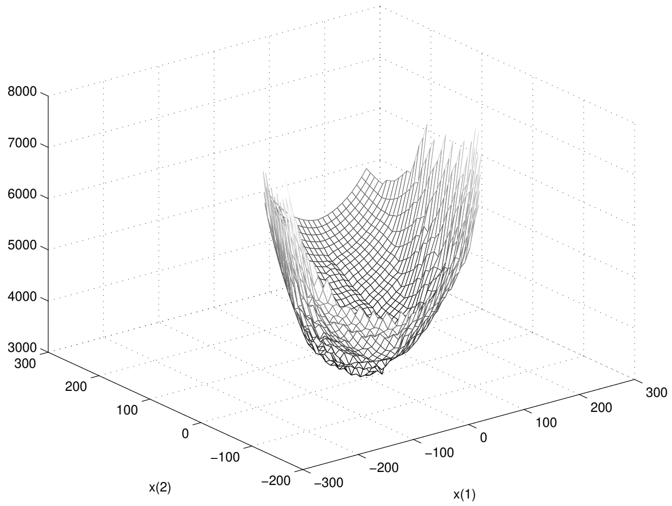

Figure 3.2: Problem A

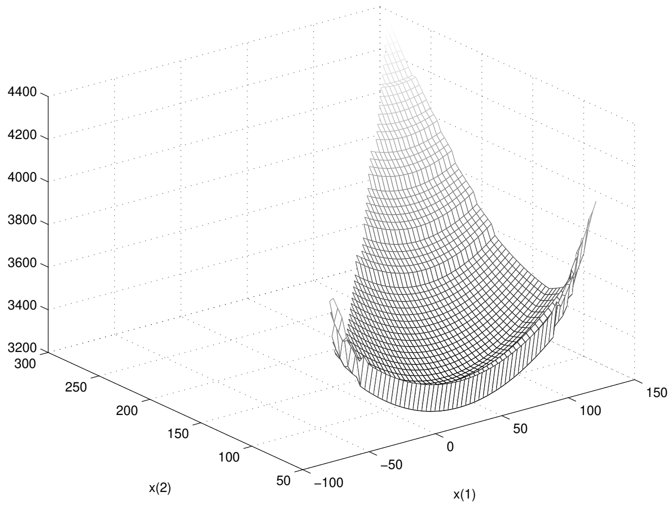

Figure 3.3: Problem B

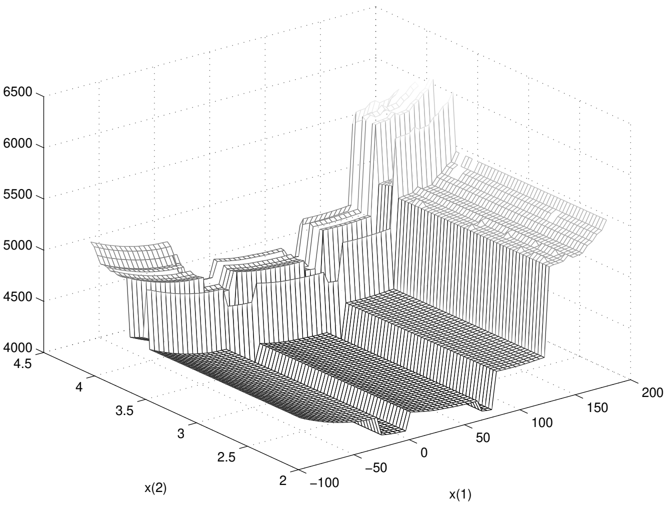

Figure 3.4: Problem C

This is the case in Figures 3.2, 3.3, and 3.4 in areas over which no surface is plotted. This situation is discussed in more detail in Chapter 4.

The discrete variables embedded in these problems give rise to some of the discontinuities in the landscapes [4]. The realities of the compressor stations themselves lead to more. The compressor stations consist of from one to more than twenty compressors. Sometimes the compressors are of different types installed over the years. Even when they are of identical manufacture, they rarely operate with the same characteristics, due to uneven wear and tear over the years [1]. The compressors can be turned on and off in different combinations.

The discontinuities of Problems A (Figure 3.2) and B (Figure 3.3) are probably due to individual compressors in stations turning on and off and the discontinuous controls for certain types of engines. The discontinuities of Problem C (Figure 3.4) are likely caused by the fact that shutting down and restarting a station can cost as much as running it for several hours.

In studying these problems, I worked with linear interpolations of data provided by Richard G. Carter, Lead Research Scientist at Stoner Associates, Inc., in Houston, Texas. I worked with interpolated data rather than actually evaluating the function to protect proprietary data and to make it easy to distribute the problems to the wider community. It also greatly reduced the amount of time required to evaluate the objective function.

Chapter 4

Hidden Constrants

The problems I studied contained hidden constraints, and this posed a problem for IFFCO. In this chapter, I describe hidden constraints, how they appear in these problems, and what problems they present for IFFCO. Then I present two ways of dealing with hidden constraints.

4.1 Definition

Recall that the goal of a constrained optimization problem is to minimize a function f in a domain $D \subset R^N$. IFFCO finds local minimizers for such problems. A local minimizer is $x^* \in D$ such that

$$f(x^*) \leq f(x) \text{ for all } x \in D \text{ near } x^*$$

The constraints on the domain D can be given as simple bound constraints, non-linear constraints, or hidden constraints. *Simple bound constraints* give a rectangular area in R^N. The general form of simple bound constraints is

$$a_i \leq x_i \leq b_i, i = 1, 2, \ldots, N$$

Non-linear constraints bound the values of at least one of the components of x by a non-linear equation. An example of a non-linear constraint on $(x)_2$ in R^2 is

$$0 \leq (x)_1 \leq 1$$

$$(x)_1^2 \leq (x)_2 \leq 1$$

The constraints on D are not always known. In this case, they are called *hidden constraints*. Hidden constraints may be simple bound constraints or non-linear constraints, or they may not conform to any rule which is easy to write down. The only thing to do in a domain with hidden constraints is to try evaluating the objective function f at a point x. If f returns a value at x, x is in the domain and is called a *feasible point*. Otherwise, x is not in the domain and is termed *infeasible*.

Importantly, hidden constraints can be discontinuous. This means it is not safe, in general, to assume f is defined everywhere between two feasible points. For example, if $f : R^1 \to R^1$ is defined at $x = 2$ and $x = 3$, it is not necessarily the case that f is defined for $x \in [2, 3]$.

Discovering x to be feasible only gives limited information. It tells us that there is a (possibly very small) neighborhood of feasible points around x, but there is no way to know how large that neighborhood is.

The problems I studied contain hidden constraints. Recall that the domain of optimization consists of values for two flow variables. Evaluating the objective function involves optimizing pressure settings throughout the system to achieve those flows. For some flow settings, this cannot be done, and the objective function fails to return a value. The combinations of flow settings for which the objective function does not return are the infeasible regions in these problems. The nature of the machinery involved makes the infeasible regions disconnected and difficult to predict. However, in the three problems I studied the feasible region was generally surrounded by a large infeasible region, and sometimes contained other, much smaller, infeasible regions. The infeasible regions are visible in Figures 3.2, 3.3, and 3.4 as areas over which no surface is plotted.

4.2 The problem with hidden constraints

Hidden constraints pose a problem for IFFCO's method of taking difference gradients. It may happen that one or more of the points in the difference gradient stencil lie in an infeasible region. If so, IFFCO obviously cannot use the value of the objective function at that point in calculating the gradient. We would still like to use the information IFFCO has, i.e. the values of the objective function at feasible points on the stencil, to calculate something like a difference gradient which will let IFFCO avoid the infeasible region and hopefully move towards the minimum. When there are no infeasible points on the stencil, IFFCO should act normally.

4.3 Method 1

One simple solution, which I will call Method 1, is to set the objective function in infeasible regions to an artificial value greater than any expected function value. This will cause the difference gradient to point *into* the infeasible region, and the linesearch direction to point *away* from the infeasible region (remember that the line search direction is, roughly speaking, the opposite of the gradient).

Method 1 is easy to implement, and it accomplishes the goal of keeping IFFCO away from the infeasible regions. Unfortunately, using an artificially high value can make the gradient *too* large. The linesearch will start far away from the current iterate, possibly even in another infeasible region! Even if the linesearch starts in a feasible region, it could take many function evaluations to find a point which meets the sufficient decrease requirement, especially if the current iterate is near the minimum.

Figure 4.1 shows how Method 1 might go to an undesirable point in problem B. The graph shows the domain of the objective function. The gray region is feasible; the white region is infeasible. IFFCO's initial point is the point marked A, and point M is the function's minimum. To calculate the difference gradient, IFFCO samples the objective function at points 1, 2, and 3. Because the function is infeasible at points 1 and 3, it is assigned a value of 10^6 at these points. Table 4.1 lists the function values Method 1 gives at the four points on the stencil.

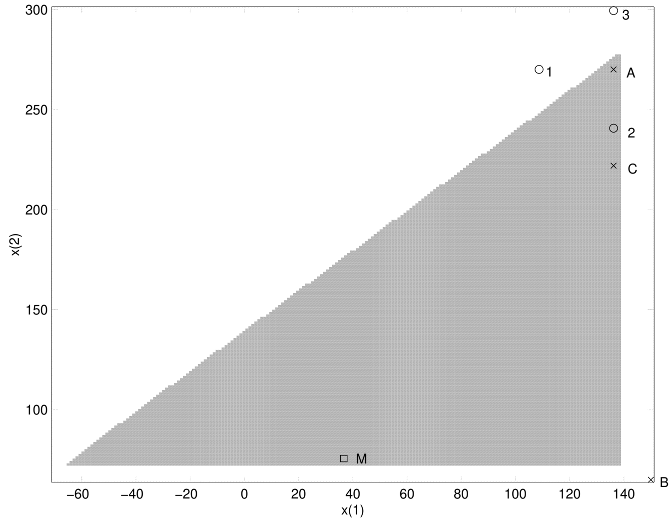

Figure 4.1: Behavior of Methods 1 and 2

The difference gradient IFFCO calculates using these function values points up and to the left, and its norm is $8.91e + 6$. IFFCO's linesearch direction, which is the opposite of the gradient, points down and to the right. Because the gradient is so large, IFFCO's linesearch direction is the projection of the opposite of the gradient onto the domain boundary. The first point the line search tries is point B, in the lower right corner of the domain. B is away from the minimum and in an infeasible region; clearly it is not a desirable point. In this case, the line search's sufficient decrease condition prevents IFFCO from accepting B as the new point, since the reported function value at B (10^6) is much larger than at A. However, IFFCO must still waste several function evaluations in the line search to discover its mistake.

Table 4.1: Function values for Method 1 in Figure 4.1

Point	Function Value
A	$4.356e + 3$
1	$1.0e + 6$
2	$4.130e + 3$
3	$1.0e + 6$
B	$1.0e + 6$

4.4 Method 2

A better way to deal with infeasible points is to assign them the largest function value at any feasible point on the stencil. I will call this Method 2. This method also gives IFFCO a push away from the infeasible region, but now the push is proportional to the size of the function near the current iterate.

Using Method 2, the example in Figure 4.1 has more desirable behavior. The largest value at a feasible point on the stencil occurs at point A, so this value is used for the infeasible points 1 and 3. Table 4.2 lists the function values Method 2 gives at the four points on the stencil.

Using these values, IFFCO calculates a much smaller difference gradient than with Method 1. The horizontal component of the gradient is zero, since the perceived difference in function values between A and 1 is zero. This means the linesearch direction is straight down. Since the norm of the gradient is much smaller ($1.50e + 3$ compared to $8.91e + 6$ for Method 1), the linesearch step is shorter, and the linesearch moves to point C. This point has several advantages over B, the point Method 1 found. Since the function at C is smaller than at A, the linesearch only takes one function evaluation. Point C is in the feasible region, so on the next step IFFCO can use three feasible points in calculating the difference gradient, instead of the two it had to work with at the initial iterate. Also, C is in the direction of the minimum, instead of off to the right of the domain.

Table 4.2: Function values for Method 2 in Figure 4.1

Point	Function Value
A	$4.356e + 3$
1	$4.356e + 6$
2	$4.130e + 3$
3	$4.356e + 3$
C	$3.981e + 3$

4.5 Comparison of Methods 1 and 2

Tables 4.3, 4.4, and 4.5 compare the results of using Methods 1 and 2 on Problems A, B, and C. The tables show the initial iterate (x_0), number of function evaluations ("Functions"), and the minimum IFFCO found ("Minimum") for each problem. In each table, the first initial iterate is the center of the domain for that problem. The other initial iterates are near the edge of the feasible region, where the problem with hidden constraints is most likely to occur.

Method 2 is not superior in every case. In Problem A especially, Method 2 sometimes took significantly more function evaluations to find the same minimum as Method 1 (although in example 2, it interestingly found a *better* minimum. In this case the extra function evaluations may be justified). In other cases it is only faster, if at all, by one or two function evaluations. In still other cases, it is clearly superior: on Problem B, Method 2 converged about 12.5% faster in example 1 and about 25% faster on example 3.

It is somewhat unfair to compare the two methods in this way. Methods 1 and 2 follow different iteration histories when they encounter infeasible points. That can significantly effect how long IFFCO takes to find a minimum and what minimum it finds, just as choosing different initial iterates can sometimes drastically change the number of function evaluations or the minimum found. Method 2 protects IFFCO from a possibly costly error, and if it does not always converge *faster* than Method 1, it at least does not converge much slower. Therefore we chose to use Method 2.

Table 4.3: Comparison of Methods 1 and 2 in Problem A

	Method 1		Method 2	
x_0	Functions	Minimum	Functions	Minimum
1. (0, 27.5)	55	3271	55	3271
2. (-150, 150)	72	3271	92	3242
3. (150, -50)	71	3269	77	3270
4. (100, 200)	79	3235	85	3235

Table 4.4: Comparison of Methods 1 and 2 in Problem B

	Method 1		Method 2	
x_0	Functions	Minimum	Functions	Minimum
1. (40, 182.5)	72	3208	63	3208
2. (100, 160)	62	3208	62	3208
3. (140, 75)	69	3208	52	3208

Table 4.5: Comparison of Methods 1 and 2 in Problem C

	Method 1		Method 2	
x_0	Functions	Minimum	Functions	Minimum
1. (43.5, 3.06)	97	4041	97	4041
2. (160, 3)	63	4063	62	4063
3. (160, 3.8)	101	4063	101	4063

Chapter 5

Computational Experiments

In this chapter I describe the computational results we achieved by running IFFCO on Problems A, B, and C.

5.1 IFFCO Parameters

IFFCO has a number of parameters which must be "tuned" for use with each problem. IFFCO will generally perform "ok" with the default values for these parameters. However, IFFCO's performance, measured in terms of the number of function evaluations and the how small the minimum it finds is, can be improved by changing the parameters. Since these problems were very simple, I was able to run IFFCO multiple times on each problem to determine which parameter settings worked best. The best parameters for each problem are not necessarily the same. Since the goal was to find parameters that work in general for this class of problems, I used the parameters that seemed to give the best performance, on average, on all three problems.

Since the choice of an initial iterate can greatly affect IFFCO's result, I set x_0 equal to the center of the bounding box in each problem in order to be able to compare the results across problems. Table 5.1 lists the "best" parameters I found for IFFCO on these problems. The IFFCO runs described in the next section all used these parameters. For a full explanation of what each parameter means, see [5].

A couple of examples will illustrate the type of impact the choice of these parameters can have, as well as the way I made choices about what parameters to use.

First, consider selecting $minh$, the smallest value of h used. Too large a value can cut off the algorithm before it gets very close to the minimum; too small a value can cause the algorithm to waste function evaluations for very small improvements in the objective function value. I tried $minh$ equal to 10^{-3}, 10^{-4}, and 10^{-6}. Table 5.2 shows the results of this experiment for each problem.

Reducing $minh$ to 10^{-4} significantly reduced the minimum found in Problems A and B, at the cost of more function evaluations. Further decreasing $minh$ substantially increased the number of function evalations but did not make the minimum much better. I decided to set $minh = 10^{-4}$.

Now consider the task of selecting an appropriate value for $maxcuts$, the number of reductions in the step length before the linesearch signals failure. I experimented with $maxcuts$ equal to 0, 1, 2, and 3. The results are tabulated in Table 5.3

Table 5.1: "Best" parameter values for use with IFFCO

Parameter	Value
fscale	10^4
minh	2^{-13}
maxh	$.5$
Max iterations at each h	3
Max function evaluations)	1000
restarts	0
termtol	1
maxcuts	1
Quasi-Newton Update Type	SR1
Minimum Strategy	take min as current point at new step
Quasi-Newton Update Strategy	re-initialize B if active set changes

Table 5.2: Experimentation with $minh$

	Problem A		Problem B		Problem C	
$minh$	Function Evaluations	f	Function Evaluations	f	Function Evaluations	f
10^{-3}	59	3270.75	47	3208.3	94	4042.07
10^{-4}	114	3269.9	74	3208.3	146	4041.19
10^{-5}	156	3269.8	121	3208.3	172	4041.18

One can see that increasing $maxcuts$ beyond 1 resulted in little or no improvement in the minimum found and increased the number of function evaluations. $maxcuts = 1$ actually saved function evaluations over $maxcuts = 0$, however, so I chose to set $maxcuts = 1$.

Decisions on the other parameters were made in a similar fashion. It is possible that other parameters would give slightly better results, or require slightly fewer functions, but it is unlikely that the difference would be significant.

5.2 Results Using IFFCO

On Problem A, IFFCO found a minimum $f = 3228.90$ after 147 function evaluations. The convergence of IFFCO for this problem is plotted in Figure 5.2.

On Problem B, IFFCO found a minimum $f = 3208.30$ in 66 function evaluations. The convergence of IFFCO for this problem is plotted in Figure 5.2.

On Problem C, IFFCO found a minimum $f = 4068.09$ in 69 function evaluations using the parameters from Table 5.1. The convergence of IFFCO for this problem is plotted in Figure 5.2.

A feature of IFFCO which has not been mentioned before is the ability to perform *restarts*. This means taking the final point IFFCO finds and using it as the initial iterate in a completely new run of IFFCO. On Problem C, using one restart found a better minimum. However, restarts are not generally useful; on Problems A and B a restart had no effect.

Table 5.3: Experimentation with $maxcuts$

	Problem A		Problem B		Problem C	
$maxcuts$	Function Evaluations	f	Function Evaluations	f	Function Evaluations	f
5	114	3269.9	74	3208.3	146	4041.2
3	108	3269.9	74	3208.3	137	4041.2
2	104	3269.9	74	3208.3	131	4041.2
1	100	3269.9	74	3208.3	125	4041.2
0	108	3269.9	74	3208.3	137	4041.2

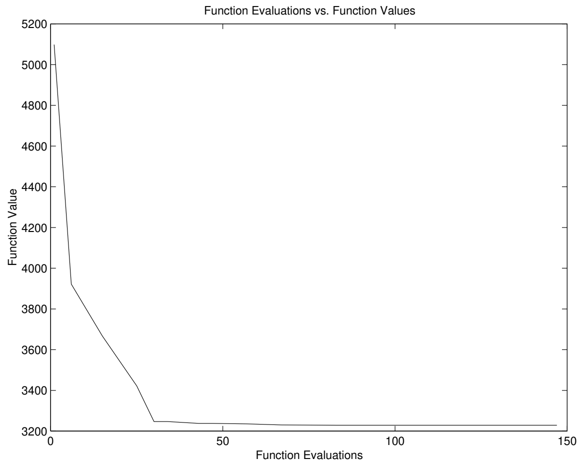

Figure 5.1: Convergence of IFFCO in Problem A

5.3 A Hybrid Approach

Better results on these problems can be obtained at lower cost in terms of function evaluations using a hybrid approach, which is described in [3]. This approach gives another optimization code, DIRECT, a fixed budget of functions. The best point DIRECT finds is passed to IFFCO as a the initial point. DIRECT is a good algorithm for global search on non-smooth function. IFFCO has better local convergence properties than DIRECT. Since the initial point DIRECT gives IFFCO is close to the minimum, IFFCO can converge to the minimum faster than DIRECT alone. The combination of the two is more efficient that either alone.

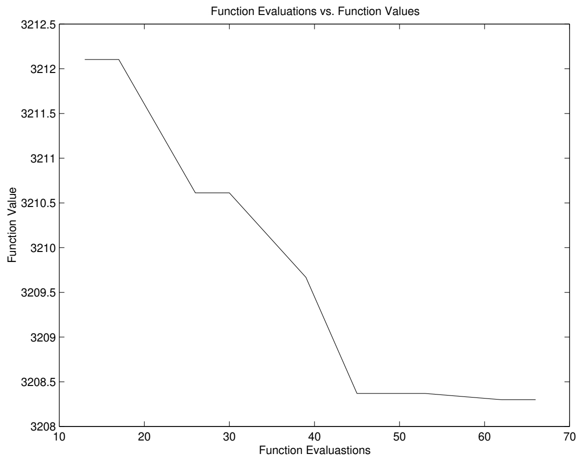

Figure 5.2: Convergence of IFFCO in Problem B

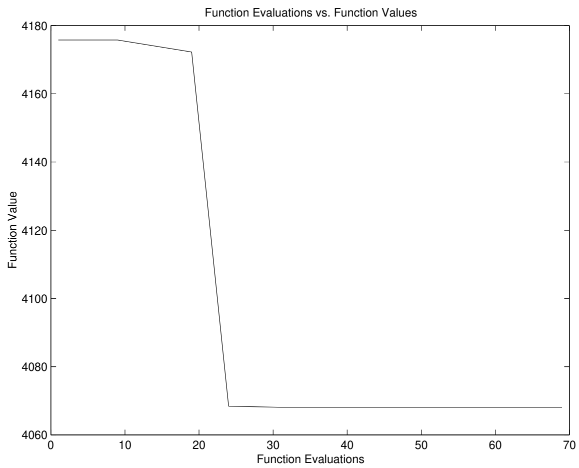

Figure 5.3: Convergence of IFFCO in Problem C

Chapter 6

Parallel Computing

Parallel computation can offer significant speed advantages for some algorithms. Section 6.1 will explain, in a simplistic way, some of the issues of parallel computing. Section 6.2 will explain how IFFCO can be parallelized.

6.1 Overview

Most personal computers run programs in serial. This means instructions are executed sequentially, one after another. In a serial computer, only one program can run at a time. Modern operating systems create the illusion that many programs are running at once by switching rapidly from one to another.

Parallel computers execute many instructions at once by running programs on many processors. A parallel computer may be running a different program on each processor, or each processor may be running a different part of the same program. It might seem that, given a parallel machine with four processors, one could divide a program into four parts, run each part on a separate processor, and finish in one-fourth the time required for a serial implementation. In practice, things rarely work out quite so well.

To see why, consider multiplying two vectors, u and v, of length four. This requires four multiplications and three additions. On a serial computer, each operation would be done sequentially, and all together they would take seven cycles (assuming multiplication and addition take one cycle each). On a parallel machine with four processors, all the multiplications can be done at once, each on a separate processor, in one cycle. Next, one processor can compute $u_1v_1 + u_2v_2$ while a second processor computes $u_3v_3 + u_4v_4$. This also takes one cycle. Finally, one of the processors can do the final addition. All together, the parallel implementation requires three cycles, for a speedup of $7/3$. This is significant, but it is not one-fourth the time the serial version requires.

This example demonstrates several important concepts of parallel computing. First, not every part of a problem is fit to be parallelized. Sometimes, one operation depends on the result of another, as the third addition in the vector multiplication example depends on the results of the first two additions. In this case, those two parts of the problem cannot be executed in parallel with each other. When some part of an algorithm consists of operations which can be done at the same time, we say it exhibits *parallelism*.

Second, throwing more processors at a problem does not always reduce the time required to solve it. If we had eight processors to run the vector multiplication on, we could do it no faster than 3 cycles, since there is no work for the four additional processors. The problem simply cannot be divided further. The number of processors a problem can be divided (or scaled) onto determines the problem's *scalability*.

Third, there is the issue of *load balancing*. We assumed that every operation in the vector multiplication would take the same amount of time. If this is not the case, then some processors will end up waiting, with nothing to do, while another processor finishes a longer task. Load balancing strategies aim to distribute (roughly) the same amount of work to each processor so that idle processor time is kept to a minimum.

Finally, we must consider the time required for communication between processors. The parallel computing environments IFFCO has run on are distributed memory machines, like most of the largest parallel computing machines. This means each processor has its own private store of data. If one processor needs something another processor knows, the two processors have to communicate. This takes time. So it is not true that the parallel implementation of the vector multiplication would take only three cycles. There is additional overhead for communication. Communication is one of the biggest culprits in slowing down parallel computation.

There are two main standards for communication in distributed memory parallel computing environments. PVM (Parallel Virtual Machine) was the most widely used standard until a few years ago. Now, MPI (Message Passing Interface), first introduced in 1994, has largely superseded PVM. The parallel version of IFFCO uses PVM for inter-processor communications. A port of IFFCO to MPI sometime in the future is likely.

This section has presented a gloomy view of parallel computing, outlining the problems that can come up. However, parallel computing techniques can significantly reduce run times if applied properly to algorithms that are good candidates for parallelization. IFFCO is such an algorithm.

6.2 Parallelism In IFFCO

IFFCO's basic algorithm gives us a natural way to implement parallelism. Algorithm 3 gives the serial algorithm for IFFCO. Two operations in algorithm 3 involve evaluating the objective function f multiple times:

- Calculate the difference gradient $\nabla_h f(x)$
- Perform a linesearch in the direction d

Importantly, none of these function evaluations are dependent on the others. In other words, finding the first function value needed to form $\nabla_h f(x)$ has nothing to do with finding the second function value, and finding the first value in the linesearch has nothing to do with finding the second. IFFCO can take advantage of this on a parallel machine by sending each function evaluation to a different processor. In the parallel implementation of IFFCO, one processor serves as the *master*, sending instructions to the other processors to evaluate f at certain points and interpreting the results. The other processors are *slaves*; their sole job is to evaluate f at the point given them by the master processor.

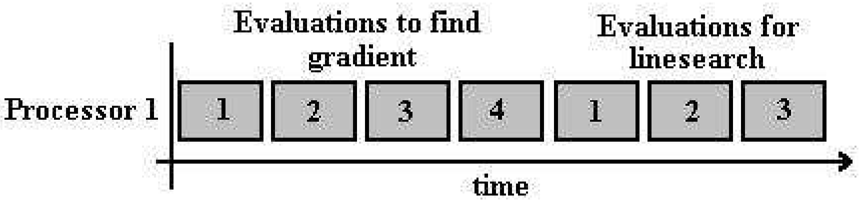

Figure 6.1: Serial IFFCO Example

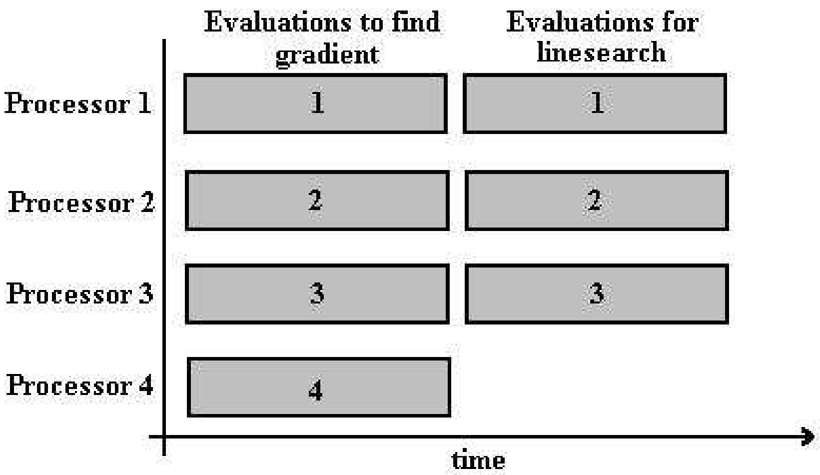

Figure 6.2: Parallel IFFCO Example

6.2.1 Comparison of Serial and Parallel IFFCO

If the domain of f is in $\mathcal{R}^n$, calculating $\nabla_h f(x)$ requires from n to $2n$ function evaluations. Depending on how quickly the sufficient decrease condition is met (if it is met at all), the linesearch can take up to $m = mMax$ function evaluations. So f may be evaluated as many as $2n + m$ times in one iteration of IFFCO.

Depending on the application, evaluating f can take minutes or even hours. In fact, function evaluations often take so long compared to the rest of the work IFFCO does that the total time for one iteration of IFFCO is approximately the same as the time spent doing function evaluations. Consider a specific example where $n = 2$, $m = 3$, and a function evaluation takes about one minute. In a serial implementation of IFFCO, the function evaluations must occur one after another (see Figure 6.1), taking up to $(2 \cdot 2 + 3) \cdot 1$ minute $= 7$ minutes for one iteration of IFFCO.

Now assume $2n = 4$ processors are available to IFFCO in a parallel machine. Now the function evaluations can be performed as in Figure 6.2. All four function evaluations for the gradient are done at once, on four separate processors, and then combined on one processor to form $\nabla_h f(x)$. This takes roughly one minute. The maximum of three function evaluations for the linesearch can also be performed simultaneously, on three separate processors, which again takes about one minute. So the total time for one iteration of parallel IFFCO is two minutes, compared to seven minutes in the serial case.

6.2.2 Scalability of IFFCO

Notice that at most $max(2n, m)$ processors can be used at a time in this parallel implementation of IFFCO. Allocating more than $max(2n, m)$ processors to IFFCO will not speed up its performance; the extra processors will remain idle.

In practice, we usually run IFFCO on $k + 1$ processors, where k evenly divides n. This means the function evaluations for the difference gradient (remember that there are between n and $2n$ of them) can be fairly evenly distributed between the processors. The extra processor is the master processor.

More processors can be utilized and performance further increased if the calculation of f itself has some sort of parallelism. In this case, each slave processor controls other slave processors which each do part of the work of evaluating f.

6.2.3 Load Balancing in IFFCO

The preceding discussion has oversimplified several points for the sake of explanation. One of them is the assumption that each function evaluation takes the same amount of time. In the real world, the time it takes to evaluate $f(x)$ can vary greatly depending on x. For instance, x could be a vector of parameters, such as damping constants, passed to an ODE solver to simulate a system involving springs. Depending on the damping constants, the system could take a very short or very long time to solve.

Further, we generally don't have the luxury of one processor per evaluation of f. This, plus the difference in time between evaluations of f, leads to load balancing problems in IFFCO.

As a specific example, take the task of calculating $\nabla_h f(x)$ on two processors using four function values: $f(x_1)$, $f(x_2)$, $f(x_3)$, and $f(x_4)$. We might first think of dividing the work like this:

Processor 1	Processor 2
$f(x_1)$	$f(x_3)$
$f(x_2)$	$f(x_4)$

This strategy is simple, but naively assumes that all four function evaluations take the same amount of time. Suppose, instead, that $f(x_1)$ takes three times as long as the others to evaluate. The situation is depicted in Figure 6.3. In this scenario, Processor 2 could have evaluated $f(x_3)$, $f(x_4)$, *and* $f(x_2)$ in the time it took Processor 1 to evaluate just $f(x_1)$. Furthermore, Processor 2 is idle half the time. This is a pessimistic case, but it can arise and IFFCO's strategy for distributing n function evaluations to p processors is designed to guard against it. This strategy is given as Algorithm 4.

Algorithm 4 Distribute

```
Distribute the first n points to processors 1 ... p
while There are points left do
  Give the next point to the first processor that finishes its assignment
end while
```

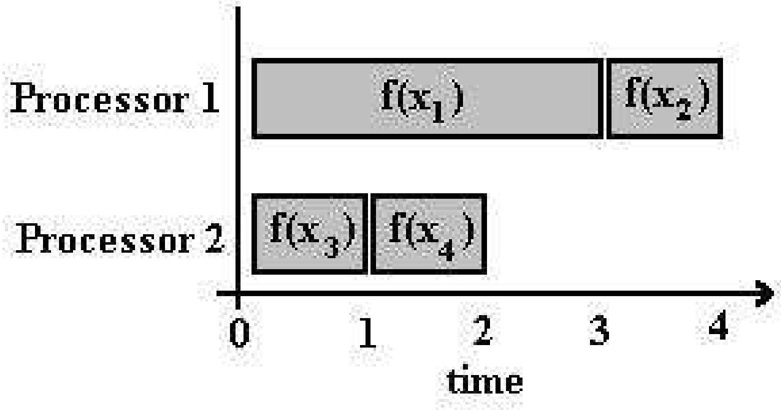

Figure 6.3: Bad Load Balancing

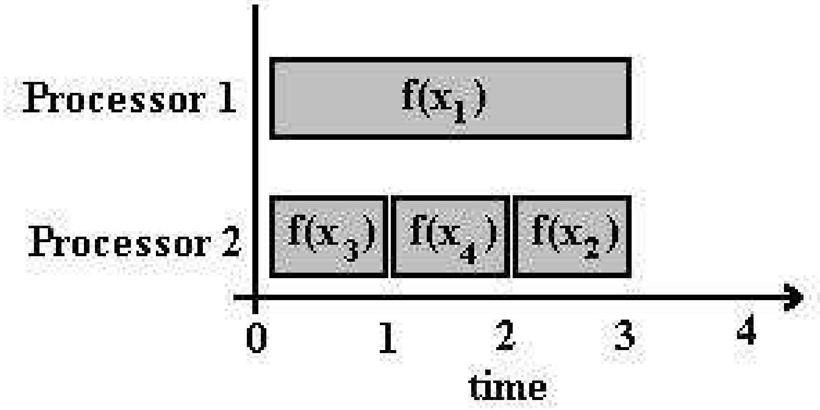

Figure 6.4: Good Load Balancing

Algorithm 4 changes the situation in Figure 6.3 to that in Figure 6.4. The time required is one minute less, and Processor 2 is no longer idle half the time. Of course, the timings won't normally work out so neatly.

In the case that all the function evaluations take about the same time, this strategy does what our intuition suggested at the beginning: it distributes (approximately) n/p function evaluations to each processor.

Bibliography

[1] R.G. Carter. Compressor station optimization: computational accuracy and speed, 1996.

[2] R.G. Carter. Pipeline optimization: dynamic programming after 30 years. Proceedings of the Pipeline Simulation Interest Group, Paper number PSIG-9308, 1998.

[3] R.G. Carter, J.M. Gablonsky, A. Patrick, C.T. Kelley, and O.J. Eslinger. Algorithms for noisy problems in gas transmission pipeline optimization. *Not sure*, 2000.

[4] R.G. Carter, D.W. Schroeder, and T.D. Harbick. Some causes and effect of discontinuities in modeling and optimizing gas trnasmission networks. Proceedings of the Pipeline Simulation Interest Group, Paper number PSIG-9308, 1994.

[5] Tony Choi, Paul Gilmore, Owen J. Eslinger, C.T. Kelley, Alton Patrick, and Jorg Gablonsky. Iffco: Implicit filtering for constrained optimization, version 2. Technical Report CRSC-TR99-23, North Carolina State University, Center for Research in Scientific Computation, 1999.

[6] C.T. Kelley. *Iterative Methods for Optimization*. Society for Industrial and Applied Mathematics, Philadelphia, 1999.

[7] C.A. Luongo, B.J. Gilmour, and D.W. Schroeder. Optimization in natural gas transmission networks: a tool to improve operational efficiency. Presented at the 3rd SIAM Conference on Optimization, 1989.