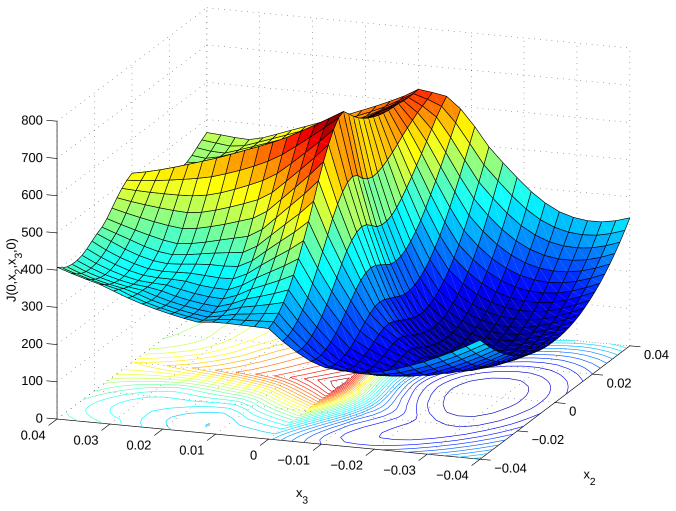

User's Guide to IFFCO

T. D. Choi
O. J. Eslinger
P. A. Gilmore
C. T. Kelley
H. A. Patrick

Version of May 18, 2001

Contents

Preface

Implicit filtering is a projected quasi-Newton method for bound constrained optimization problems. The gradients are computed with a finite difference and the difference gradient varies as the optimization progresses.

IFFCO is a FORTRAN implementation of the implicit filtering method. This book is a complete reference to Version 2 of IFFCO, covering installation, testing, and use of the serial, PVM, and MPI implementations.

Chapter 1
Implicit Fitering

1.1 Sampling Methods for Optimization

Chapter 2

Introduction to IFFCO

IFFCO (Implicit Filtering for Constrained Optimization) is an algorithm for optimizing functions with multiple minima. IFFCO is designed to solve problems subject to simple box constraints. The mathematical description of these problems is:

$$\min_{x \in Q} \hat{f} : R^n \to R \text{ where } Q = \{x \in R^n \mid l^i \leq x^i \leq u^i \ , i = 1, \ldots, n \ \} \ , \tag{2.1}$$

where l^i and u^i are the lower and upper bounds respectively on the ith variable. The set Q, defined by the constraints on the variables, is called the hyper-box.

IFFCO was designed to minimize functions of the form:

$$\hat{f}(x) = f(x) + \phi(x) \ . \tag{2.2}$$

In (2.2) $f(x)$ is a smooth function with a simple form. For example, $f(x)$ could be a convex quadratic. $\phi(x)$ is a low-amplitude high-frequency perturbation, which we refer to as noise in this document. In this context, low amplitude means

$$\max_{x \in Q} |\phi(x)| \ll \max_{x \in Q} |f(x)| \ . \tag{2.3}$$

$\phi(x)$ need not be continuous. IFFCO is particularly effective on problems where the amplitude of $\phi(x)$ decays near local minima of $f(x)$. It is not necessary to be able to calculate $f(x)$ and $\phi(x)$. It is only necessary that $\hat{f}(x)$ behaves like a function that satisfies Equation 2.2.

Implicit filtering has been successfully applied to problems in semiconductor design [17–20], high-field magnets [4, 11, 16], automotive engineering [5–7], and geosciences [1, 2, 10, 15]. The algorithm used in IFFCO is analyzed in [9, 14]. IFFCO and algorithms like IFFCO are applied to problems far more complex that those that satisfy the hypotheses of the theoretical results.

IFFCO is a variation on the gradient projection method described in [3], that uses a sequence of finite difference steps (scales) to approximate the gradient. A brief outline of the algorithm used in IFFCO is given below.

Algorithm 2.1 Simple IFFCO algorithm

```
Pick initial $x$ and $h$; find $f(x)$ and the Difference Gradient $\nabla_h f(x)$.
Initialize the model Hessian $B$ to the identity
while $h$ and $\nabla_h f(x)$ satisfy conditions do
  Use $\nabla_h f(x)$ and $B$ to calculate a descent direction $d$. This step is a quasi-Newton step.
  Perform a line search in the direction $d$, and signal success if some criteria are met.
  if line search was successful then
    Accept new point and project into the box $Q$.
  else
    $h \leftarrow h/2$
  end if
  Calculate the Difference Gradient $\nabla_h f(x)$.
  Update $B$ with either a rank-one SR1 update, or a rank-two BFGS update.
end while
```

A variation of Algorithm 2.1 is to restart Algorithm 2.1 using the last point obtained in the iterative process as the next initial point until a point is obtained that satisfies the termination criteria at every scale. A point that satisfies the termination criteria at every scale is called a minimum at all scales.

2.1 Serial and Parallel Implementations

Both serial and parallel versions of IFFCO are included in the distribution package. Both are implemented in ANSI standard Fortran 77. There are two separate parallel versions, one using PVM 3.4 calls and one using MPI 1.1 standard calls. Both of these parallel versions operate in exactly the same way, and we often refer to them collectively as "parallel IFFCO" or "the parallel version." Users with long function evaluations (at least several seconds in duration) will see the greatest benefit from using the parallel version.

The serial version is essentially a straight-forward implementation of Algorithm 2.1. The parallel version seeks to improve the performance of this algorithm by evaluating the objective function in parallel at two stages: during the difference gradient calculation and during the line search.

Externally, using the parallel version is no different from using the serial version except for the extra overhead of setting up the parallel environment. The calling sequence and output are exactly the same. In most cases, the minimum found by both versions will be the same or about the same. The number of function evaluations required for the parallel version of IFFCO will usually exceed slightly that required for the serial version, but since the added function evaluations are carried out in parallel little or no extra wall-clock time is used. This differences is explained in Section 5.2.

Because of the way IFFCO is parallelized it can use at most $max(2n + 1, m)$

processors, where n is the dimension of the objective function and m is the maximum number of cutbacks performed in the line search. Using more processors will not improve the performance of IFFCO.

2.2 Supported Platforms

The serial version of IFFCO should compile and run on any system with a UNIX-like operating system and a Fortran 77 compiler. In addition, the parallel version requires PVM 3.4 or an implementation of MPI which supports the MPI 1.1 standard (e.g. LAM/MPI 6.3, MPICH 1.2.1, and almost every other recent version of MPI).

2.3 Availability

The latest version of IFFCO can be downloaded from the IFFCO web page:
`http://www4.ncsu.edu/~ ctk/iffco.html`

All three versions of IFFCO (serial, PVM, and MPI) are distributed in the same package.

2.4 Documentation

This manual is the main source of information about using IFFCO. The IFFCO distribution includes a "readme" file containing most of the information in Chapter 3. In addition, each subroutine of IFFCO is documented by comments in the source code.

`Tim: add something about IFFCO papers`

2.5 Contact for Questions and Problems

The primary contact for IFFCO is:

C.T. Kelley
Department of Mathematics
Center for Research in Scientific Computation
North Carolina State University
Raleigh, NC 27695-8205
`Tim_Kelley@ncsu.edu`

Electronic mail is the optimal way to contact Kelley.

Chapter 3

Getting Started with IFFCO

This chapter explains how to install IFFCO and what is included in the installation package. Checking that IFFCO is installed correctly by running it on the included test problems is also discussed. The last section of this chapter is a "quick start" guide of sorts, which describes how to use IFFCO on your own objective function in the most straight-forward manner.

3.1 Unpacking IFFCO

To unpack IFFCO, copy the distribution file `IFFCO.tar.gz` (see Section 2.3 for information on obtaining IFFCO) to the directory above where you want to install IFFCO. Then issue the commands:

```
> gunzip IFFCO.tar.gz
> tar xf IFFCO.tar
```

This will create the directory `iffco/` and place all IFFCO files there.

IFFCO's directory structure is quite simple; it is diagrammed in Table 3.1. The main directory, `iffco/`, contains a readme file; the makefile; a script for creating the serial, PVM, and MPI versions of IFFCO from the combined source file; and batch files for submitting parallel IFFCO jobs under the IBM Parallel Operating Environment. Executable files and the output files IFFCO creates are also placed here.

Table 3.1. *IFFCO's directory structure*

`iffco/`		Main directory
	`src/`	Source code files
	`testdata/`	Test problem configuration files

The subdirectory `src/` contains all IFFCO source code files. Table 3.2 describes the contents of each source file. The separate source code files for the serial, PVM, and MPI versions of IFFCO require a bit of explanation. Since the three versions contain much common code, we work with a master file, `iffco.F`, during our development and use the Fortran preprocessor on it to generate the three separate files for distribution.

If you want to alter IFFCO's source code and plan to use only one version, you can make whatever changes you want solely in the source code file for that version. If you want to use more than one version, you might want to modify `iffco.F` and then create the separate files from that. The script `iffco/makevers` is provided for creating `ser_iffco.f`, `mpi_iffco.f`, and `pvm_iffco.f` from `iffco.F`.

Table 3.2. *Files in iffco/src.*

`iffco.F`	The master file; contains all three versions of IFFCO, separated by preprocessor directives.
`ser_iffco.f`	Serial version of IFFCO.
`pvm_iffco.f`	PVM version of IFFCO.
`mpi_iffco.f`	MPI version of IFFCO.
`ser_main.f`	Driver program for the serial version of the test problems.
`par_main.f`	Driver program for the parallel versions of the test problems.
`testfunc.f`	Function evaluation subroutines for the test problems.
`ser_sample_main.f`	Sample driver program for serial IFFCO.
`par_sample_main.f`	Sample driver program for parallel version of IFFCO.
`sample_func.f`	Sample function evaluation subroutine.
`vtf.f`	LINPACK subroutines used by IFFCO. These subroutines are included in the IFFCO distribution to avoid requiring that LINPACK be installed and to avoid the unnecessary complication of linking with LINPACK.

The other subdirectory of `iffco/`, `testdata/`, contains configuration files for each of the test problems.

3.2 Configuring the Makefile

There are four platform-specific variables in `iffco/Makefile`. To compile IFFCO, these must be set properly. Table 3.3 lists the variables and their meanings. All four are defined near the top of the makefile, and example settings (in comments) are provided for several platforms.

Table 3.3. *Variables in iffco/Makefile.*

F77	The command to use for compiling Fortran 77 code.
INCLUDE_DIR	The location of the PVM header file (fpvm3.h) or the MPI header file (mpif.h).
LIB_DIR	The location of PVM or MPI libraries.
LIBS	The names of the PVM or MPI libraries to link with.

3.2.1 Serial

To compile the serial version, you need only set the variable `F77` to the command used to compile Fortran 77 programs on your system. The default is `f77`. The other three variables can be left at their defaults.

3.2.2 PVM

If PVM is configured in the standard way on your system, the following settings should allow you to compile the PVM version of IFFCO:

```
F77 = f77
INCLUDE_DIR = $(PVM_ROOT)/include
LIB_DIR = $(PVM_ROOT)/lib/$(PVM_ARCH)
LIBS = -lfpvm3 -lpvm3 -lgpvm3
```

Note that the environment variable `PVM_ROOT` must be set to the path of the main PVM directory and the environment variable `PVM_ARCH` must be set to the architecture type for your system. These variables are usually set when PVM is installed. If these variables are not set properly, or if the settings above do not work for your system, contact your system administrator for help.

3.2.3 MPI

Since there is so much variation among MPI packages, it is impossible to give one set of settings that will work on most systems. If you want to use the MPI version of IFFCO, check the makefile to see if settings for your MPI package are already there. If they are not, or if the specified settings do not work properly, contact your system administrator for help.

Sometimes, MPI packages include a compiler or compiler script which automatically includes the MPI header file and links in the appropriate libraries. If this is the case, you can set `F77` to that command and leave the other three variables at their default values. For instance, LAM/MPI 6.3 provides `hf77` for compiling Fortran 77 code using MPI (the analogous command under MPICH 1.2.1 is `mpif77`). Therefore, the following settings should work on a system using LAM/MPI 6.3:

```
F77 = hf77
INCLUDE_DIR = .
LIB_DIR = .
LIBS =
```

3.3 Running the Test Program

To help you verify that you have correctly installed IFFCO, the distribution comes with a suite of twelve test problems. The test problem subroutines used were written by Jörg Gablonsky. Three of the functions are simple constant, linear, and quadratic functions in two dimensions. Seven are functions originally given by Dixon and Szegö [8]. These problems have been widely used to compare global optimization algorithms [8, 12, 13]. The remaining two come from Yao [21].

Test programs have been included to run IFFCO on all the test problems and compare the results your installation of IFFCO returns to control results. The test programs inspect 1.) the number of function evaluations IFFCO used to find a minimum and 2.) the value of the objective function at the minimum found. These two pieces of data must match the control results with a relative error no greater than 0.01 for the results to be considered "correct."

The control results were generated on an IBM SP/2 under AIX Version 4 Release 3. The serial version was compiled with `xlf`. The parallel versions were compiled with the script `mpxlf`. The PVM version used AIX4SP2 header files and libraries in PVM 3.4; the MPI version was compiled using MPI header files and libraries in the IBM Parallel Environment for AIX, Version 2 Release 4. Both parallel versions were run on two processors using IBM's Parallel Operating Environment.

3.3.1 Compiling the test program

If you have already set the makefile variables as described in Section 3.2, compile the appropriate test program by typing

```
> make ser_test
```

for the serial version,

```
> make pvm_test
```

for the PVM version, or

```
> make mpi_test
```

for the MPI version.

3.3.2 Executing the test program

The process for running the test program varies depending on which version of IFFCO you want to use.

Serial

To run the serial version of the test program, simply type the command

```
> ser_test
```

from the `iffco/` directory.

PVM

Since we ran the PVM test program on two processors to generate the control results, you should also run the test program on two processors to allow a valid comparison between the results on your system and the control results. Changing the number of processors may affect the number of function evaluations required. It should not significantly affect the minimum found.

On most systems, you will run the PVM version of IFFCO from the PVM console. To do this, you need to first set your working directory and executable path to the directory in which you installed IFFCO. You can do this by adding the following line at the top of your hostfile:

```
* wd=$IFFCO ep=$IFFCO:$PVM_ROOT/bin/$PVM_ARCH
```

where `$IFFCO` is the full path name of the directory where you installed IFFCO, `$PVM_ROOT` is the directory where PVM is installed on your system, and `$PVM_ARCH` is the PVM architecture type of your system.

Then, start PVM with that hostfile:

```
> pvm hostfile
```

and run `pvm_test` from the console:

```
pvm> spawn -2 -> pvm_test
```

If you use a batch system instead of the PVM console to run parallel programs, submit the program `pvm_test` through that system. A batchfile for use with `llsubmit` in the IBM Parallel Operating Environment, `iffco/pvm_test.cmd`, is included in the IFFCO distribution. If you do use this batchfile, note that it specifies *three* processors. This is because under the Parallel Operating Environment, one processor is dedicated to managing PVM. Only the other two actually run the test program. You can submit the test program to run by typing:

```
   > llsubmit pvm_test.cmd
```

MPI

Since we ran the MPI test program on two processors to generate the control results, you should also run the test program on two processors to allow a valid comparison between the results on your system and the control results. Changing the number of processors may affect the number of function evaluations. It should not affect the minimum found.

On most other systems, the command for running the test program under MPI on two processors is:

```
 mpirun -np 2 mpi_test
```

or something similar.

If you use a batch system to run parallel programs, submit the program `mpi_test` through that system. A batchfile for use with `llsubmit` in the IBM Parallel Operating Environment, `iffco/mpi_test.cmd`, is included in the IFFCO distribution. You can submit the test program to run by typing:

```
   > llsubmit mpi_test.cmd
```

3.3.3 Interpreting the test program output

For each problem, the test program will print to standard output a brief diagnostic message. If the results of running IFFCO on problem n are close enough (have a relative error no greater than 0.01) to the control results, the test program will print

```
Problem n:  [Problem name] worked
```

If they are not close enough, the test program will print

```
Problem n:  [Problem name] failed
```

If IFFCO solves all test problems correctly, it is running properly on your system. If one or more test problems fail, all is not lost. Look at the file `iffco/logfile`, where the test program's results and the control results are tabulated for each problem. Inspect the output for the problems that failed. If the test and control values for minimum (Min) and number of function evaluations (Fevals) are fairly close, chances are IFFCO is working properly and the results simply aren't within the test program's tolerance for "correctness." These kinds of disagreements can be caused by differences between the floating point arithmetic on your system and the system used to calculate the control results.

3.3.4 Modifying the test problem parameters

Modifying the test problem parameters should normally not be necessary, but it may be interesting to experiment with them to learn the effect each parameter has on the algorithm. The parameters IFFCO uses for each test problem are stored in configuration files in the directory `iffco/testdata`. There is one configuration file per test problem. The format of the configuration files is one parameter per line, like so:

```
2                   Dimension
0.5                 Starting point
0.5
1.0                 Upper bounds
1.0
0.0                 Lower bounds
0.0
1.0                 Scaling
2.4414062500D-04    minh
0.50D0              maxh
.
.
.
```

As you can see, when a vector is required in the input (as is the case for the starting point, upper bounds, and lower bounds), each component of the vector is listed on a separate line.

The names of the configuration files are all numbers. Table 3.4 describes the mapping from problem name to file name.

Table 3.4. *Test problem configuration file names*

Test Problem Name	File Name
Constant	10
Linear	11
Quadratic	12
Branin	13
Shekel-5	14
Shekel-7	15
Shekel-10	16
Hartman-3	17
Hartman-6	18
Goldprice	19
Sixhump	20
Shubert	21

3.4 Using IFFCO the Fast and Easy Way

This section describes how to apply IFFCO to your own objective function in the most straightforward manner possible: using the serial version and the default parameters. This section is meant as a sort of "quick start" guide. Chapter 4 gives more detail on the topics covered briefly here.

We have included some source code files to serve as a template for using IFFCO. `src/sample_func.f` contains a sample function evaluation subroutine. `src/ser_sample_main.f` and `src/par_sample_main.f` are driver programs for the serial and parallel versions, respectively, which call IFFCO and report its results. We will now explain how to modify these files for your own purposes. Only the serial version is discussed here. The process for the parallel version is similar, but uses `src/par_sample_main.f` instead of `src/ser_sample_main.f`.

The first step is packaging your objective function in a way that IFFCO can use. This means a Fortran 77 subroutine with a particular calling sequence. The calling sequence IFFCO expects is:

```
func(n, x, f, flag, idata, ilen, ddata, dlen, cdata, clen)
```

where `n` is the problem dimension, `x` is the point to evaluate the function at, and `f` is the parameter for returning the value of the objective function at `x`. The other parameters will be explained later. The last six parameters to the function call are for passing data other than the problem dimension and the point to evaluate. These will not be necessary for most users and are not discussed here; they are covered in Section 4.1.14.

`src/sample_func.f` contains the skeleton of an IFFCO-ready objective function. You may either place your objective function code directly in this file, or have this subroutine call another subroutine which evaluates you objective function. Note that if you call an external subroutine, you must also modify the makefile to compile and link in the external source code.

Before returning, your function evaluation subroutine should set `f` to the value of the objective function at `x`. It should also set `flag` to 0 if the function evaluated normally or 1 if the function failed to evaluate. IFFCO uses this parameter to detect hidden constraints.

Here is an example of an objective function subroutine for use with IFFCO:

```
      subroutine myfunc(n, x, f, flag, idata, ilen, ddata, dlen,
     +    cdata, clen)

      implicit none
C     Parameters
      integer n, flag, ilen, dlen, clen
      double precision x(n), f
      integer idata(ilen)
      double precision ddata(dlen)
      character cdata(clen)

C     A convex quadratic with low-amplitude, high-frequency noise:
      f = x(1)*x(1) + x(2)*x(2) + (.1)*sin(15*3.14159*x(1)*x(2))

C     Since this function will always return a value, flag is always 0.
      flag = 0

      end
```

Once the code for your objective function is prepared, you need to write a program to call IFFCO. `src/ser_sample_main.f` provides an example of such a driver program.

The driver program first defines variables for each of IFFCO's parameters. One of these is the dimension of the problem:
`parameter(dim = 2)`
Make sure `dim` is the same as the dimension of your objective function.

Next, the driver program sets each IFFCO parameter. Eventually you will probably want to tune the IFFCO parameters for your problem (see Section 4.1 for help with that), but the parameters used in `src/ser_sample_main.f`, which are IFFCO's defaults, should work passably well in many cases. There is one change you will have to make, however. Locate the section of code which sets the vectors `u` and `l`, which looks like this:

```
     do 10 i = 1,dim
       u(i) = 1.0
       l(i) = 0.0
  10 continue
```

These are the upper and lower bounds in each coordinate direction of the hyper-box IFFCO searches for a minimum. You should set these to something reasonable for your problem.

After setting up the parameter values, the driver program calls IFFCO. Once IFFCO returns, the driver program prints out the minimum IFFCO found. The code for calling IFFCO and reporting the result looks like this:

```
C     Start IFFCO
      call iffco(myfunc,x,u,l,fscale,minh,maxh,dim,maxit,
     +     restart,writ,termtol,f,maxcuts,option,
     +     idata, ilen, ddata, dlen, cdata, clen)

C     Print brief results of the run.
      write(6,*) 'IFFCO found this minimum:'
      write(6,*) 'f = ', f
      do 20 i = 1,n
        write(6,*) 'x(', i, ') = ', x(i)
 20   continue
      write(6,*) 'IFFCO used ', nevalsIF, ' function evaluations.'
```

You should not need to modify any of this code.

Now that the objective function and driver program are set up, you can compile your program by typing:

```
> make ser_sample
```

from the `iffco/` directory. To run the program, type:

```
> ser_sample
```

Chapter 4

General Use of IFFCO

The previous chapter briefly described the use of IFFCO in the simplest case. This chapter will take a more general approach. Each parameter in IFFCO's calling sequence is described, as well as IFFCO's return values and output. Most of the information in this section applies to both the serial and parallel versions of IFFCO. Chapter 5 covers issues specific to the parallel version.

4.1 Calling Sequence

The calling sequence for IFFCO is

```
iffco(func, x, u, l, fscale, minh, maxh, n, maxit,
     + restart, writ, termtol, f, maxcuts, option,
     + idata, ilen, ddata, dlen, cdata, clen)
```

The following sections describe each of these parameters in depth and give suggestions for choosing appropriate values.

4.1.1 Objective function `func`

`func` is a user-supplied Fortran subroutine for evaluating the objective function, $\hat{f}(x)$. IFFCO calls your subroutine like this:

```
   call func(n, x, f, flag, idata, ilen, ddata, dlen, cdata, clen)
```

Therefore, you should force your subroutine to have the calling sequence above by either rewriting your code or writing a wrapper subroutine that calls the actual function evaluation subroutine. (Alternatively, you may change the call to `func` in IFFCO, which is located in the subroutine `funcIF`. However, this is not recommended.) You should also ensure `func` sets the two return values correctly, as explained below.

In the calling sequence above, `n` is the dimension of the domain of $\hat{f}$, `x` is the point to evaluate $\hat{f}$ at, `f` is a return value for $\hat{f}(x)$, and `flag` is another return value.

The six remaining parameters are for passing extra data to the function evaluation code. This way, if your function requires input other than `n` and `x`, it can receive that input and still conform to IFFCO's calling pattern. See Section 4.1.14 for details on using these parameters. Table 4.1 summarizes the types and meanings of each of the parameters to `func`.

Table 4.1. *`func` parameters*

Name	Type	Meaning
`n`	integer	Dimension of problem
`x`	double precision vector, length `n`	Point
`f`	double precision	Return value: $\hat{f}(x)$
`flag`	integer	Return value: feasibility flag
`idata`	integer vector, length `ilen`	Extra integer data
`ilen`	integer	Length of `idata`
`ddata`	double precision vector, length `dlen`	Extra double precision data
`dlen`	integer	Length of `ddata`
`cdata`	character vector, length `clen`	Extra character data
`clen`	integer	Length of `cdata`

The second return value, `flag`, tells IFFCO whether or not the value returned in `f` is a legitimate function value. Normally, `func` should set `flag` to 0. `flag` should be set to 1 if the objective function did not evaluate normally or is not defined at `x`. This could happen, for instance, if the objective function is a model of some physical process and `x` indicates some combination of input parameters that is physically impossible. A point `x` for which $\hat{f}(x)$ has no value is called *infeasible*. If an objective function has one or more regions of infeasible points, it is said to have *hidden constraints*, constraints that can only be discovered by attempting to evaluate the objective function. `flag` helps IFFCO detect and deal with hidden constraints.

If `x` is infeasible (i.e. `flag` = 1), IFFCO will do one of two things. If in the line search, IFFCO will use $\hat{f}(x)$ =`fscale`. Since `fscale` is expected to be greater than or equal to $max_{x \in Q}\left(|\hat{f}(x)|\right)$, the line search should never accept such a function value. If in the difference gradient evaluation, IFFCO will use

$$\hat{f}(x) = f^* + \epsilon |f^*| \tag{4.1}$$

where

$$f^* = max\left[\hat{f}(x_c) | x_c \text{is feasible and on the stencil}\right] \tag{4.2}$$

and $\epsilon = 10^{-6}$. This guarantees that the search direction IFFCO uses will be away from the infeasible region, but not overwhelmingly so.

4.1.2 Initial iterate `x`

The initial iterate, `x`, is a double precision vector of length `n`. If `option(3)` is set to 0, IFFCO will use `x` as the starting point to search for a minimum. If `option(3)` is set to 1, IFFCO will ignore `x` and use the center of the search region as the starting point.

Before IFFCO returns, it sets `x` to the location of the minimum found.

4.1.3 Bounds `u, l`

Together, `u` and `l` define the hyper-box Q, which is the part of the domain of $\hat{f}$ where IFFCO searches for a minimum — the search region. Both `u` and `l` are double precision vectors of length `n`. `u` contains the upper bounds of Q for each coordinate direction; `l` contains the lower bounds. Q should fit as tightly around the region of the minimum as possible; this will reduce the number of scales IFFCO has to run through to find a solution.

Internally, IFFCO maps Q to the unit hyper-cube (i.e. $[0,1]^n$). IFFCO performs all calculations on points within the unit hyper-cube. The final solution is mapped back to the original hyper-box before being returned to the user.

4.1.4 Function scaling `fscale`

`fscale` is a double precision scalar used to scale the function. `fscale` should be an approximation to the maximum magnitude the objective function obtains in the search region. The default value for `fscale` is 1. If `fscale` is set to zero, the default is used.

Unfortunately, $max_{x \in Q}\left(|\hat{f}(x)|\right)$ is not usually known. Hence, users may have to experiment with values of `fscale` to determine a value appropriate for their problem. Choosing `fscale` too large often causes IFFCO to find answers that are unacceptable to the user while reporting convergence at many scales without taking a step at these scales. If `fscale` is too small, the gradients calculated may become too large. Choosing `fscale` too small often results in the line search being unable to find a suitable new point for many point-scale pairs. For more details concerning problems with `fscale` see Chapter 6.

4.1.5 Difference increment limits `minh, maxh`

`minh` is a double precision scalar. It is a lower bound for the last and smallest finite difference step (scale) IFFCO uses to calculate the finite difference gradient. When the scale shrinks below `minh`, the algorithm either terminates or restarts. `minh` must be no greater than `maxh`.

`maxh` is a double precision scalar. It is the first and largest scale IFFCO uses to calculate the finite difference gradient. `maxh` must be no greater than 0.5. If `minh` = `maxh`, IFFCO will use exactly one scale — `maxh`.

`minh` and `maxh` are *scaled* finite difference steps. That is, if `maxh` = 0.5, then the first difference step IFFCO uses will be $0.5(u(i) - l(i))$ for each dimension i.

The default value of `maxh` is 0.5, and this is a good starting point for most problems. Determining a suitable value for `minh` is more complicated problem. Recall that IFFCO was designed to minimize functions of the form:

$$\hat{f}(x) = f(x) + \phi(x) \tag{4.3}$$

where $f(x)$ is a smooth function with a simple form and $\phi(x)$ is a low-amplitude high-frequency perturbation (noise). Choosing `minh` involves the curvature of $f(x)$, the amplitude of $\phi(x)$, the amount of accuracy needed by the user, and the cost per function evaluation.

If `minh` is too large, IFFCO may not obtain an acceptable answer. However, if `minh` is too small, some of the scales used may be so small that gradients calculated using them are dominated by changes in $\phi(x)$. Choosing `minh` too small often manifests itself by the line search being unable to find a suitable new point for some of the smaller scales

If the amplitude of $\phi(x)$ is approximately constant, then optimally `minh` should be a low estimate of $O(\max_{x\in Q}(|\phi(x)/fscale|^{1/3})$ (where Q is the search region). If $\phi(x)$ decays near minima of $f(x)$, `minh` may be smaller. For instance, if it is known that $\hat{f}(x)$ is smooth near minima of $f(x)$, `minh` may be on the order of the cube root of machine epsilon. Unfortunately, the behavior of $\phi(x)$ is usually not well known, so users may have to experiment with `minh` to find a suitable value for their problems.

4.1.6 Function dimension `n`

`n` is the dimension of the domain of the objective function or, put another way, the number of independent variables in the objective function. If `n` is greater than 24, the user will have to change the value of the parameter `mx` in IFFCO's source code (see Section 4.2).

4.1.7 Iteration limits `maxit`

`maxit` is an integer vector of length two.

`maxit(1)` specifies the maximum number of iterations IFFCO is allowed to take at a given scale. The default value is 100, which will be used if `maxit(1)` is set to 0.

`maxit(2)` is a soft limit on the number of function evaluations IFFCO is allowed to use over the course of the entire algorithm. The default is $100n^2$, which will be used if `maxit(2)` is set to 0. IFFCO will not stop immediately if it exceeds `maxit(2)` in the middle of an iteration. Rather, it will finish the current iteration and then stop. This means that the actual number of function evaluations used will sometimes exceed `maxit(2)`.

4.1.8 Number of restarts `restart`

`restart` is an integer specifying the number of restarts to perform.

Restarts should be done only if IFFCO is returning answers that are not as low as the user expects. Restarts are done to help ensure that the final answer obtained is a minimum at all scales.

The default is not to do restarts.

4.1.9 Level of output `writ`

`writ` is an integer scalar that controls the type and amount of output IFFCO generates. IFFCO writes most of its output to the file `iffco.out`. IFFCO can also mirror the same output to standard output (usually the screen). For file output only, `writ` can range from 0 to 4, with `writ`=0 indicating no output and higher values indicating more output. Adding 10 to `writ` (i.e. `writ` = 10 to 14) causes the same output to be copied to standard output (usually the screen).

The levels of output are described briefly in Table 4.2. For more on what the levels of output mean, see Section 4.4.1.

The value of `writ` has no effect on IFFCO's other output file, `points.out`.

Table 4.2. *Settings of `writ` and corresponding level of output*

`writ`	**Type of Output**
0, 10	No output
1, 11	Standard information (scaled) for each iteration
2, 12	Standard information plus current iterate (scaled) and number of function evaluations used so far
3, 13	Standard information (unscaled)
4, 14	Standard information plus current iterate (unscaled) and number of function evaluations used so far

4.1.10 Termination criterion at a scale `termtol`

`termtol` is a double precision scalar used for determining convergence of the algorithm at a given point for a given scale. The default value of `termtol` is 1.0 and will be used if `termtol` is set to 0.

The algorithm is considered to have converged at a point x when

$$||x - P(x - d(x))|| \leq h \cdot termtol \tag{4.4}$$

where $P(x - d(x))$ is the projection of the steepest descent step onto the search region and h is the current difference increment, or scale. When the algorithm converges at one scale, the scale is reduced by a factor of two.

The appropriate value for `termtol` is problem-dependent. In many computations, using the default value for `termtol` has worked well. However, this is not the case for all problems. If `termtol` is too large, the algorithm will report convergence at many scales without finding a new point at these scales, and return an unacceptable answer. If `termtol` is too small, the algorithm may not reduce scales fast enough. In this case the line search will be unable to find acceptable new points for many scales, thus costing the user many unnecessary function evaluations.

4.1.11 Minimum function value `f`

`f` is a double precision variable used for returning the minimum value of the objective function IFFCO finds. IFFCO ignores the initial value of `f`.

4.1.12 Line search limit `maxcuts`

`maxcuts` is an integer that specifies the maximum number of cutbacks IFFCO may use in the line search. The default value for `maxcuts` is 3 and it will be used if `maxcuts` is set to 0.

4.1.13 Options `option`

`option` is an integer vector of length seven. Its components control different aspects of how IFFCO works.

`option(1)` determines the type of quasi-Newton update IFFCO uses. If `option(1)` is set to

- 0, no quasi-Newton step is performed.
- 1, the SR1 update is used.
- 2, the BFGS update is used.

The default is `option(3)`=1.

`option(2)` determines how IFFCO uses the location of the current minimum at each stage. As IFFCO progresses, it keeps track of the lowest function value it has seen (in the line search or gradient calculation) and the location of that value, x_m (x_m will not always be one of the iterates). If `option(2)` is set to

- 0, x_m is not used.
- 1 or greater, x_m is taken as the current point at a restart.
- 2 or greater, x_m is taken as the current point at each new scale.
- 3, x_m is taken as the current point after each line search.

The default is `option(2)`=2.

`option(3)` determines how IFFCO uses the initial iterate, `x`. If `option(3)` is

- 0, IFFCO uses `x` as the initial iterate.
- 1, IFFCO ignores `x` and uses the center of the search region as the initial iterate.

The default is `option(3)`=1.

`option(4)` determines when IFFCO re-initializes the quasi-Newton matrix, which is the approximate inverse Hessian (if BFGS is being used) or the approximate Hessian (if SR1 is being used). The quasi-Newton matrix, B, is initialized to the identity matrix, I. If `option(4)` is

- 0, B is re-initialized at each new scale.
- 1, B is re-initialized whenever the active set changes.
- 2, B is re-initialized only if positivity is lost.

The default is `option(4)`=1.

`option(5)` and `option(6)` set the location of IFFCO's main output file. If `option(5)`=0, IFFCO creates `iffco.out` in the working directory and uses it for output. If `option(5)`=1, `option(6)` should be the unit number of an open output file. IFFCO will use that file for output. The default is `option(5)`=0.

`option(7)` is used only in the parallel version. If `option(7)`=0, the master processor is used for function evaluations in the difference gradient calculation; if `option(7)`=1, it is not. This option can help with load balancing; see Section 5.1 for more information. The default is `option(7)`=0.

4.1.14 Extra data for the function evaluation

The last six parameters to IFFCO allow you to pass extra data to your objective function subroutine, `func`. Three vectors are provided for doing this: `idata` for integers, `ddata` for double precision data, and `cdata` for characters. The other three parameters, `ilen`, `dlen`, and `clen` give the sizes of `idata`, `ddata`, and `cdata`, respectively.

Every time IFFCO evaluates the objective function, it passes these six parameters to `func`. They can be used when a function requires input other than `n` and the point to evaluate. If no extra data is needed, leave the vectors empty and set their sizes to 0.

IFFCO does not modify any of these vectors or their sizes. However, you may modify them in `func`. If this happens in the serial version, the modified data will be passed to `func` the next time it is called. This may be a useful feature in some cases. In the parallel version, however, we do not recommend you modify the "extra data" parameters. Since each processor maintains its own copy of the vectors and their sizes, changes on one processor would be propagated to subsequent function evaluations *only* on that processor, not on the others. This may cause unwanted results.

4.2 Other Parameters

Besides the parameters in IFFCO's calling sequence, there are a few other variables in the IFFCO source code that may have to be changed in rare cases. Most of these specify the sizes of temporary arrays. Because Fortran does not allow dynamic

sizing of local arrays, the sizes of some arrays must be specified in advance. This is done in Fortran parameter statements, which have the form
`(parameter varname = m)`
where `m` is a constant expression containing no variables.

Some array sizes depend on the problem dimension, `n`. IFFCO uses `mx` in place of `n` when calculating these array sizes. By default, `mx` is 24. If your problem dimension is greater than 24, you will have to edit the IFFCO source code file and change the value of `mx` in the following subroutines: `iffco`, `initIF`, `statsIF`, `funcIF`, `ser_gradIF`, `par_gradIF`, `minIF`, `maxIF`, `ser_linesearchIF`, `par_linesearchIF`, `evaltolIF`, `quasiIF`, `stepIF`, `restartIF`, `takeminIF`, `pointsIF`, `mastersendIF`, and `slaverecvIF`.

The array `fhist` stores the value of $\hat{f}(x_c)$ for every iteration. The size of `fhist` is given by `mx2`, which is 1000 by default. If you expect more iterations, you should change `mx2` where it is defined in the main subroutine, `iffco`.

The last of these array sizes is used only in the parallel version. `maxprocs` is the maximum number of processors available to IFFCO. Since IFFCO can use at most max(2*`n`, `maxcuts`)+1 processors at once, `maxprocs` is set to 2*`mx`+1 by default. This should be enough unless you allow more than 2*`mx`+1 cutbacks in the line search. In that case, you need to change the value of `maxprocs` in the subroutines `iffco`, `par_gradIF`, and `par_linesearchIF`.

The final variable you may have to adjust in the IFFCO source code is the global variable `mepsIF`. `mepsIF` is meant to be an approximation to machine epsilon. It is used, for instance, as the tolerance for comparing two floating point numbers. That is, $a = b$ if and only if $|a - b| < mepsIF$. By default, `mepsIF`=10^-12. `mepsIF` is set in the main subroutine, `iffco`.

4.3 Return Values

IFFCO returns two pieces of data through the calling sequence. In addition, you can garner more information by accessing the global variables IFFCO declares.

The location of the minimum found, x^*, and the value of the objective function there, $\hat{f}(x^*)$, are returned through the calling sequence in `x` and `f`, respectively.

IFFCO also defines several global variables that might be interesting. To access them, you need to make the following variable declarations in the program you use for calling IFFCO:

```
integer nevalsIF
double precision fminIF, xminIF(mx), fmaxIF
common /globalIF/nevalsIF
common /globalIF2/fminIF, xminIF, fmaxIF
```

`nevalsIF` is the number of function evaluations IFFCO used to find a minimum. `fminIF` is the smallest objective function value IFFCO saw at any point — in the line search or the difference gradient computation — and `xminIF` is the point x_m where $\hat{f}(x_m)$ =`fminIF`. (The value of `mx` in the declaration of `xminIF` should be the same as the value of `mx` used in IFFCO.) `fminIF` may not be the same as `f`, the minimum function value IFFCO returns. This is because depending on the value of option(2), x_m may never be adopted as the current iterate, even if $\hat{f}(x_m) < \hat{f}(x_c)$ (where x^c is the current iterate). fmaxIF is the greatest objective function value IFFCO encountered.

4.4 Output

IFFCO generates two output files, iffco.out and points.out. These files are both placed in the calling program's working directory. iffco.out traces the progress of the algorithm and contains various error messages. The amount of information output to this file can be controlled by one of the parameters to IFFCO, writ. The information in iffco.out can also be mirrored to standard output or put in a different file. points.out reports the function value and coordinates of every objective function evaluation IFFCO performs.

4.4.1 iffco.out

IFFCO's main output file is iffco.out. The output in iffco.out is divided into three sections: parameter reporting, algorithm progress, and function value history.

The amount and type of output in iffco.out are controlled by one of the parameters to IFFCO, writ. If writ is 0, no output is produced except for messages reporting errors that cause IFFCO to exit. Setting writ to 1, 2, 3, or 4 determines the type of information given in the algorithm progress section and has no effect on the other two sections. Adding 10 to the value of writ (i.e. writ = 10, 11, 12, 13, or 14) causes whatever is written to iffco.out to be written to standard output.

In some cases you may want to send the output that normally goes to iffco.out to another file. You can do this by setting option(5) to 1 and putting the unit number of the file to use for output in option(6). The unit number specified in option(6) must be a valid unit number to a file that is already open.

The first section of iffco.out, the parameter reporting section, reports the values of all parameters passed to IFFCO (except for idata, ddata, and cdata, which may be very long vectors). Next, it reports how many processors IFFCO is running on. Finally, it reports the number of invalid parameters detected. IFFCO checks to see if the parameters it is given meet the expected conditions. For instance, u(i) must be less than l(i). If IFFCO finds any invalid parameters, it will output a brief message describing the problem(s) and quit. These error messages are always sent to standard output, even if writ is less than 10. As long as writ is not equal to 0 (in which case iffco.out is never opened), these messages are written to iffco.out as well.

The output in the parameter reporting section looks like this:

```
fscale     =   1.0000000000000
minh       =   2.4414062500000D-04
maxh       =   0.50000000000000
n          =   2
maxit(1)   =   3
maxit(2)   =   5000
```

```
restart    =   0
writ       =   14
termtol    =   1.0000000000000D-04
ncuts      =   5
x(  1)     =   0.6000000000E+01
x(  2)     =   0.5000000000E+01
u(  1)     =   0.1000000000E+02
u(  2)     =   0.1000000000E+02
l(  1)     =   0.0000000000E+00
l(  2)     =   0.0000000000E+00
option(  1)=   1
option(  2)=   3
option(  3)=   1
option(  4)=   1
option(  5)=   0
option(  6)=   0
option(  7)=   0
running IFFCO on   1 processors
---------------------------------------------
There are a total of   0 input errors
---------------------------------------------
```

The next section of output in `iffco.out` is the main one, the algorithm progress section. As IFFCO progresses, it periodically outputs information about what it is doing. For `writ` = 1, 2, 3, or 4, IFFCO prints out an update every time the line search is successful or the scale is reduced. The most basic information, called the "standard information" has this format:

```
 m     ||x||        f           ||g||          h         cuts
 0 0.5000E+00 -.5754E+00 0.3145E-01 0.5000E+00  Stencil Failure
 0 0.5000E+00 -.5754E+00 0.9690E-01 0.2500E+00  Stencil Failure
 0 0.5000E+00 -.5754E+00 0.1503E+00 0.1250E+00  Stencil Failure
 1 0.4800E+00 -.6199E+00 0.1247E+01 0.6250E-01     1
 .
 .
 .
```

The first column, labelled `m`, is the iteration number for the given scale. This is equivalent to the number of successful line searches conducted so far at this scale. The second column, `||x||`, is the Euclidean or L_2 norm of the scaled current iterate, x_c, divided by $\sqrt{n}$. The `f` column is the value of the objective function at the current iterate. This is the scaled value if `writ` is 1 or 2 and the unscaled value if `writ` is 3 or 4. The third column, `||g||`, is the L_2 norm of the *scaled* difference gradient at x_c divided by $\sqrt{n}$. The `h` column is the current scale (difference increment).

If the line search was successful then the last column, labelled `cuts`, gives the number of cuts the line search used to find an acceptable point. If the line search failed or the scale is going to be reduced for some other reason, the last column gives the reason for reducing the scale. Table 4.3 lists messages that might be printed in this column and elaborates on their meanings.

Table 4.3. *Explanation of scale reduction messages*

Message	**Explanation**
Line search failure	Line search could not find an acceptable point using `maxcuts` or fewer cutbacks.
Convergence	The algorithm has converged for this scale.
Maxit(1) exceeded	The maximum iterations on a given scale have been exceeded.
Maxit(2) exceeded	The function evaluation budget for the entire algorithm has been exceeded.

If `writ` is 2 or 4, IFFCO will output the current iterate, x_c, and the number of function evaluations used so far below the standard information. In that case, the output looks like this:

```
  m     ||x||        f          ||g||         h        cuts
  0 0.5000E+00 -.5754E+00 0.3145E-01 0.5000E+00  Stencil Failure
 x(  1)     =  0.5000000000E+00
 x(  2)     =  0.5000000000E+00
 x(  3)     =  0.5000000000E+00
 x(  4)     =  0.5000000000E+00
Total function evaluations:    9
  m     ||x||        f          ||g||         h        cuts
  0 0.5000E+00 -.5754E+00 0.9690E-01 0.2500E+00  Stencil Failure
 x(  1)     =  0.5000000000E+00
 x(  2)     =  0.5000000000E+00
 x(  3)     =  0.5000000000E+00
 x(  4)     =  0.5000000000E+00
Total function evaluations:   17
  .
  .
  .
```

If `writ` is 2, the scaled coordinates of x_c will be output. If `writ` is 4, the unscaled coordinates will be used.

Besides these progress updates, various non-critical but important notifications may also appear in the algorithm progress section of `iffco.out`. The messages that may appear are:

`function failed to evaluate!  Setting f := fscale.`

The objective function routine set `flag` to 1 to indicate an error in evaluation (see Section 4.1.1).

`Minimum Taken:` `f` = function value

> The current iterate has been set to the location of the smallest known function value (see discussion of `option(2)` in Section 4.1.13). If `writ` is 2 or 4, this message will be followed by the unscaled coordinates of the new current iterate.

`B not positive definite`

> The quasi-Newton matrix is no longer positive definite. B will be re-initialized to the identity, I.

After the algorithm is finished, IFFCO outputs the function value history. This consists of the unscaled objective function value at every iteration during the algorithm, starting with the value at the initial iterate and proceeding to the final minimum value IFFCO discovered. This output looks like the following:

```
 f history (unscaled)
   -0.57535140943302
   -0.57535140943302
   -0.57535140943302
  .
  .
  .
```

4.4.2 `points.out`

`points.out` lists the unscaled objective function value, $\hat{f}(x)$, and the `n` unscaled components of x for each function evaluation IFFCO performed. The function values are listed one per line. Here is a sample of some output from a `points.out` file:

```
   1    19.875836249802    0.               0.
   2    1.1517529438694  -10.0000000000000  0.
   3    14.905882611065   10.0000000000000  0.
   .
   .
   .
```

The first column is the evaluation number and the second is the function value. The last `n` are the components of x.

Chapter 5

Parallel Operation of IFFCO

There are two parallel versions of IFFCO, one using PVM (Parallel Virtual Machine) calls and another using MPI (Message Passing Interface) calls. These are referred to collectively as "parallel IFFCO." Parallel IFFCO can use at most $max(maxcuts, 2n+1)$ processors, where n is the number of variables in the objective function and $maxcuts$ is the maximum number of cutbacks allowed in the line search. The calling sequence for parallel IFFCO is exactly the same as for serial IFFCO, and almost all parameters have the same interpretation. The only exception is `option`; parallel IFFCO uses `option(7)`, which is ignored by serial IFFCO (see Section 4.1.13). The return values and output files of parallel and serial IFFCO are also identical.

The code calling parallel IFFCO is responsible for setting up and terminating the parallel communication environment. In the PVM case this means all processors must join the group `iffcogroup` before calling IFFCO and then call `pvmfbarrier` and `pvmfexit` when IFFCO finishes. In the MPI case, all calling processors must call `mpi_init` before calling IFFCO and call `mpi_barrier` followed by `mpi_finalize` when IFFCO finishes. The parallel IFFCO source code files contain the subroutines `comminitIF` and `commexitIF` for performing these initialization and finalization tasks. To use these subroutines, declare them external in the program that calls IFFCO:

```
external comminitIF, commexitIF
```

and then call `comminitIF` before IFFCO and `commexitIF` after IFFCO:

```
      call comminitIF()
      call iffco(. . .)
      call commexitIF()
```

Parallel IFFCO uses a master-slave paradigm. All the processors in the virtual machine must call IFFCO with the same parameters. Once IFFCO starts, the processor with the lowest task id (in the PVM case) or rank 0 (in the MPI case) becomes the master processor and all other processors are slaves. The master processor manages the algorithm, which is the same as in serial IFFCO except for the difference gradient calculation and the line search. In those two subroutines, parallel IFFCO takes advantage of the natural parallelism and performs most of the function evaluations needed in parallel on the slave processors. The master processor makes decisions about how to continue the algorithm based on those function evaluations. The parallel implementation of the difference gradient computation and line search are explained in more detail in the following two sections.

5.1 Parallel Difference Gradient Computation

The parallel IFFCO source code files (`pvm_iffco.f` and `mpi_iffco.f`) contain a subroutine called `par_gradIF` for calculating the difference gradient in parallel. To calculate a difference gradient, IFFCO requires at most $2n$ function values, where n is the number of variables in the objective function. The required function evaluations can all be conducted in parallel.

Since there will not always be $2n$ processors available, the master processor first calculates every point in the domain where the objective function must be evaluated. Then the master sends $max(2n, p-1)$ points to the slave processors, where p is the number of processors available, including the master processor. The master processor evaluates the next point, assuming there are points still to be evaluated.

When the master finishes its function evaluation, it waits until a slave processor returns a value. If there are more points to be evaluated, the master gives that slave another point to evaluate. Giving a processor the next function evaluation to do as soon as it finishes one, instead of waiting for all the slave processors to finish and then sending out $p-1$ function evaluations at the same time, insures that the slaves will be busy as often as possible. After sending out the new point, the master checks to see if the number of function values it has so far is a multiple of p. If so, and if there are points still to be evaluated, the master does another function evaluation. Otherwise, the master continues to wait for results from the slaves. This process continues until the master has all necessary function values.

In some extreme cases, using the master processor to do function evaluations in `par_gradIF` may lead to load balancing problems. We have attempted to approximately balance the work done by each processor by having the master do one function evaluation for every $p-1$ function evaluations done by the $p-1$ slaves. This scheme should be adequate as long as the amount of time required for each function evaluation is approximately constant. If some processors are slower than others or if the input to the objective function greatly affects the amount of time required to evaluate the function, some function evaluations may take much longer than others. In this case, the slave processors could sit idle while waiting for the master to finish a very time-consuming function evaluation. To avoid this, you can set `option(7)` to 1 to prevent IFFCO from using the master processor for function evaluations in `par_gradIF`. IFFCO will still use the master processor for function evaluations in the line search.

If `option(7)` is 0 (i.e. the master processor is used for function evaluations), parallel IFFCO can use up to $2n$ processors in the difference gradient computation.

If `option(7)` is 1, parallel IFFCO can use up to $2n + 1$ processors — $2n$ slaves for function evaluations plus the master processor to manage the algorithm.

5.2 Parallel Line Search

The parallel IFFCO source code files (`pvm_iffco.f` and `mpi_iffco.f`) contain a subroutine called `par_linesearchIF` for performing a line search in parallel. The parallel line search works identically to the serial line search, with two exceptions:

1. function evaluations are done p at a time, where p is the number of processors, and
2. the parallel line search does not use quadratic or cubic models to improve the line search.

In the parallel line search, the master processor determines all trial points that may be evaluated in the line search (there are up to *maxcuts* of these, where *maxcuts* is the maximum number of cutbacks allowed). It then sends the first $p-1$ function evaluations to the slaves and does one function evaluation itself. When all these have completed, the master processor determines if the sufficient decrease condition has been met. If it has, the line search terminates. If it has not, p more function evaluations are carried out. This process continues until sufficient decrease is met or *maxcuts* cutbacks are taken.

`par_linesearchIF` does not attempt to load balance by sending each slave a new job as soon as the old one is finished as `par_gradIF` does. This is because `par_linesearchIF` cannot determine whether or not more cutbacks will be necessary until it receives results from all p function evaluations. Blindly doing extra cutbacks may result in wasted function evaluations (and time). For the same reason, there is no option not to use the master processor for function evaluations in the line search.

Quadratic and cubic models are not used in the parallel line search because this technique involves using function values from previous steps to determine the next point to evaluate. `par_linesearchIF` performs p function evaluations at a time, so it has no previous function values to base a model on. A possible alteration to `par_linesearchIF` might be to use the p function values determined in the first set of evaluations to create a model of the objective function in the line search direction. Then the location of the next batch of cutbacks could be determined using this model.

Although the parallel line search may take the same number of cutbacks as the serial version (if quadratic and cubic models were not used in the serial line search, it always would), it could still do more function evaluations. Suppose a line search requires three cutbacks. Since the serial version completes each function evaluation before deciding whether or not to do the next cutback, serial IFFCO requires three function evaluations to perform this line search. Suppose parallel IFFCO is run on five processors, with $maxcuts = 5$. Since parallel IFFCO does not know in advance how many cutbacks will be needed in the line search, it does as many function evaluations as processors — five. When all five have completed, `par_linesearchIF` determines that the third trial point is the first to meet sufficient decrease, and accepts that point. This is recorded as three cutbacks and five function evaluations. Because the extra function evaluations are done in parallel, this does not necessarily make the parallel line search slower than the serial version.

Chapter 6

Trouble Shooting

This chapter discusses problems you may encounter while running IFFCO and gives advice on fixing them. In diagnosing problems with IFFCO, the output written to `iffco.out` when `writ` is greater than or equal to 1 (see Section 4.4) will be useful. The output examples in this chapter were generated using `writ` = 1.

6.1 Input Errors

The first thing IFFCO does when it is called is check that all the parameters have valid values. If IFFCO finds invalid parameter settings, it will terminate before it begins searching for a minimum. When these errors occur, IFFCO prints a message to `iffco.out` (if `writ` is not 0) and to standard output (regardless of `writ`'s value). After checking all its parameters for errors, IFFCO prints a message indicating the number of errors it found and exits. You should correct the problem(s) specified and try running IFFCO again.

This is an example of the type of output IFFCO generates when there are errors in the input (the number and type of errors may vary):

```
.
.
.
u(  1) is less than l(  1)
x(  1) is out of bounds
fscale negative
---------------------------------------------
There are a total of   3 input errors
---------------------------------------------
```

6.2 Line Search Failure

This section discusses what to do if the line search is not able to find a new point at many scales. Failure in the line search is indicated by the warning "Line search failure" in the output from IFFCO.

6.2.1 Line search failures and poor results

If IFFCO is returning answers that are not as good as the you expect and IFFCO is reporting failure in the line search at many scales then `fscale` may be too small. Here is an example of output from IFFCO when `fscale` is too small for the problem:

```
  m    || x ||        f          || g ||        h        cuts

  0  0.1000D+01  0.8495D+04  0.1134D+05  0.5000D+00   10
  1  0.2357D-01  0.8443D+04  0.1133D+05  0.5000D+00  Convergence
  0  0.2357D-01  0.8443D+04  0.1700D+05  0.2500D+00  Line search failure
  0  0.2357D-01  0.8443D+04  0.1983D+05  0.1250D+00  Line search failure
  0  0.2357D-01  0.8443D+04  0.2125D+05  0.6250D-01  Line search failure
Number of function evaluations:   58
```

Using larger values for `fscale` usually solves this problem. You may have to experiment to find a good value for `fscale`.

6.2.2 Line search failures and reasonable results

If IFFCO is returning acceptable answers but is reporting failure in the line search at many scales, then `termtol` may be too small. Here is an example of output from IFFCO when `termtol` is too small for the problem:

```
  m    || x ||        f          || g ||        h        cuts

  0  0.1000D+01  0.8495D+01  0.1134D+02  0.5000D+00    1
  1  0.3571D+00  0.8009D+01  0.1134D+02  0.5000D+00    0
  2  0.7308D+00  0.7882D+00  0.5182D+01  0.5000D+00  Line search failure
  0  0.7308D+00  0.7882D+00  0.6157D+01  0.2500D+00    5
  1  0.6685D+00  0.1758D+00  0.5112D+00  0.2500D+00    1
  2  0.5145D+00  -.3437D-02  0.7545D-01  0.2500D+00    1
  3  0.4967D+00  -.9604D-02  0.2065D-01  0.2500D+00  Line search failure
  0  0.4967D+00  -.9604D-02  0.2067D-01  0.1250D+00  Line search failure
  0  0.4967D+00  -.9604D-02  0.2067D-01  0.6250D-01  Line search failure
Number of function evaluations:   58
```

6.3 Line search failure at small scales

If IFFCO is returning acceptable answers but is reporting failure in the line search for many of the smaller scales, then `minh` may be too small. Here is an example of output from IFFCO when `minh` is too small for the problem:

```
  m    || x ||        f          || g ||        h        cuts
  0  0.1000D+01  0.8508D+01  0.1134D+02  0.5000D+00    1
  1  0.3477D+00  0.8008D+01  0.1133D+02  0.5000D+00  Convergence
  0  0.3477D+00  0.8008D+01  0.1697D+02  0.2500D+00    4
  1  0.7884D+00  0.7990D+01  0.1697D+02  0.2500D+00    0
  2  0.5091D+00  -.9306D-02  0.5221D-01  0.2500D+00  Convergence
  0  0.5091D+00  -.9306D-02  0.5615D-01  0.1250D+00  Convergence
  0  0.5091D+00  -.9306D-02  0.5754D-01  0.6250D-01  Convergence
  0  0.5091D+00  -.9306D-02  0.5792D-01  0.3125D-01  Convergence
  0  0.5091D+00  -.9306D-02  0.5802D-01  0.1562D-01    1
  1  0.5011D+00  -.9794D-02  0.2134D-01  0.1563D-01  Convergence
  0  0.5011D+00  -.9794D-02  0.8965D-02  0.7813D-02  Convergence
  0  0.5011D+00  -.9794D-02  0.3496D+00  0.3906D-02    3
  1  0.5027D+00  -.9869D-02  0.2015D+00  0.3906D-02    2
  2  0.5027D+00  -.9939D-02  0.2809D-02  0.3906D-02  Convergence
  0  0.5027D+00  -.9939D-02  0.4012D-01  0.1953D-02  Line search failure
  0  0.5027D+00  -.9939D-02  0.5027D-01  0.9766D-03  Line search failure
  0  0.5027D+00  -.9939D-02  0.1020D+00  0.4883D-03    5
  1  0.5027D+00  -.9939D-02  0.6640D-01  0.4883D-03    2
  2  0.5028D+00  -.9939D-02  0.6526D-01  0.4883D-03  Line search failure
  0  0.5028D+00  -.9939D-02  0.7184D-01  0.2441D-03  Line search failure
  0  0.5028D+00  -.9939D-02  0.7356D-01  0.1221D-03    5
  1  0.5028D+00  -.9940D-02  0.6012D-03  0.1221D-03    2
  2  0.5028D+00  -.9940D-02  0.5869D-03  0.1221D-03  Line search failure
  0  0.5028D+00  -.9940D-02  0.4747D-03  0.6104D-04  Line search failure
  0  0.5028D+00  -.9940D-02  0.4466D-03  0.3052D-04  Line search failure
  0  0.5028D+00  -.9940D-02  0.4395D-03  0.1526D-04  Line search failure
Number of function evaluations:   179
```

6.4 Poor Answers

This section discusses what to do if IFFCO is obtaining answers that the user feels are unacceptable but the message "Line search failure" is not reported in the output from IFFCO.

6.4.1 No convergence problems and poor results

If IFFCO reports convergence at many scales but poor answers are being obtained then `termtol` and/or `fscale` may be too big. An example of this behavior follows:

```
   m    || x ||       f          || g ||        h        cuts

  0  0.1000D+01  0.8508D+00  0.1134D+01  0.5000D+00  Convergence
  0  0.1000D+01  0.8508D+00  0.1701D+01  0.2500D+00  Convergence
  0  0.1000D+01  0.8508D+00  0.1984D+01  0.1250D+00  Convergence
  0  0.1000D+01  0.8508D+00  0.2125D+01  0.6250D-01  Convergence
Number of function evaluations:   12
```

6.4.2 Small `maxh`

If IFFCO obtains poor answers and `maxh` is set to a value less than 0.5, then the iterates may have become trapped in a local minima. Using a larger value of `maxh` may help.

6.4.3 Large `minh`

Poor answers may also be caused by choosing `minh` too large. If this is the case, IFFCO may not try enough scales to resolve the minimum as precisely as possible. If the answer IFFCO is returning is in the correct region but not as low as you think it should be, try reducing `minh` by factors of two. The value of `minh` to use depends on the problem, the precision required by the user, and the precision of the hardware being used.

Appendix A

IFFCO Subroutines

Besides the main subroutine, `iffco`, IFFCO uses 29 other subroutines to get its job done. Of these, nine are communication subroutines that provide a wrapper around the PVM and MPI calls. The other 20 perform various computational and input/output functions. All subroutine names in IFFCO are suffixed with 'IF.' All the subroutines are in the main IFFCO source code file (`ser_iffco.f`, `pvm_iffco.f`, or `mpi_iffco.f`). Table A.1 lists the 20 computational and input/output subroutines and Table A.2 lists the communication subroutines.

Table A.1. *IFFCO subroutines*

`inputsIF`	Prints out the parameters passed to IFFCO.
`initIF`	Initializes some variables and checks to see that the parameters passed to IFFCO have valid values.
`statsIF`	Outputs the information in the algorithm progress section of `iffco.out`.
`scaleIF`	Scales x from the search region defined by `u` and `l` onto the unit hyper-cube.
`unscaleIF`	The inverse of `scaleIF`.
`funcIF`	Calls the user's objective function evaluation code.
`ser_gradIF`	Calculates a difference gradient in serial.
`par_gradIF`	Calculates a difference gradient in parallel.
`minIF`	Sets x_m, the location of the smallest function value, to the current iterate, x_c, if $\hat{f}(x_c) < \hat{f}(x_m)$. Also records $\hat{f}(x_m)$.
`maxIF`	If $\hat{f}(x_c)$ is greater than `fmaxIF` (the current largest function value), sets `fmaxIF` to $\hat{f}(x_c)$.
`updateIF`	Updates the current iterate, gradient, and function value.
`ser_linesearchIF`	Performs a serial line search.
`par_linesearchIF`	Performs a line search in parallel.
`eyeIF`	Re-initializes the quasi-Newton matrix to the identity.
`evaltolIF`	Calculates the tolerance, $\|x - P(x - d(x))\|$, at the current iterate.
`quasiIF`	Updates the quasi-Newton matrix.
`stepIF`	Calculates the quasi-Newton step $-H^{-1} \cdot g$ where g is the gradient at x_c and H is an approximation to the Hessian.
`restartIF`	Performs necessary steps before a restart.
`takeminIF`	Replaces the current iterate with the point stored in `xminIF` (the location of the smallest function value encountered so far).
`pointsIF`	Prints $\hat{f}(x)$ and x to `points.out`.

Table A.2. *Parallel communication subroutines used by IFFCO*

`mastersendIF`	Called by the master processor to order a slave processor to do a function evaluation.
`slaverecvIF`	Called by a slave processor to receive an order for a function evaluation from the master processor.
`slavesendIF`	Called by a slave processor to send the result of a function evaluation to the master processor.
`masterrecvIF`	Called by the master processor to receive the result of a function evaluation from a slave processor.
`getmytidIF`	Gets the MPI rank or PVM task id of the calling process.
`getnprocsIF`	Gets the number of processors available to IFFCO.
`gettidIF`	Gets the MPI rank or PVM task id of the ith process.
`comminitIF`	Initializes the parallel environment.
`commexitIF`	Shuts down the parallel environment.

Bibliography

[1] K. R. Bailey and B. G. Fitzpatrick, *Estimation of groundwater flow parameters using least squares*, Tech. Rep. CRSC-TR96-13, North Carolina State University, Center for Research in Scientific Computation, April 1996.

[2] K. R. Bailey, B. G. Fitzpatrick, and M. A. Jeffries, *Least squares estimation of hydraulic conductivity from field data*, Tech. Rep. CRSC-TR95-8, North Carolina State University, Center for Research in Scientific Computation, February 1995.

[3] D. P. Bertsekas, *On the Goldstein-Levitin-Polyak gradient projection method*, IEEE Trans. Autom. Control, 21 (1976), pp. 174–184.

[4] L. J. Campbell, Y. Eyssa, P. Gilmore, P. Pernambuco-Wise, D. M. Parkin, D. G. Rickel, J. B. Schillig, and H. J. Schneider-Muntau, *The US 100-T magnet project*, Physica B, 211 (1995), pp. 52–55.

[5] T. D. Choi, O. J. Eslinger, C. T. Kelley, J. W. David, and M. Etheridge, *Optimization of automotive valve train components with implicit filtering*, Tech. Rep. CRSC-TR98-44, North Carolina State University, Center for Research in Scientific Computation, December 1998. Submitted for publication.

[6] J. W. David, C. Y. Cheng, T. D. Choi, C. T. Kelley, and J. Gablonsky, *Optimal design of high speed mechanical systems*, Tech. Rep. CRSC-TR97-18, North Carolina State University, Center for Research in Scientific Computation, July 1997. Mathematical Modeling and Scientific Computing, to appear.

[7] J. W. David, C. T. Kelley, and C. Y. Cheng, *Use of an implicit filtering algorithm for mechanical system parameter identification.* SAE Paper 960358, 1996 SAE International Congress and Exposition Conference Proceedings, Modeling of CI and SI Engines, pp. 189–194.

[8] L. Dixon and G. Szegö, *The Global Optimisation Problem: An Introduction*, in Towards Global Optimization 2, L. Dixon and G. Szegö, eds., vol. 2, North-Holland Publishing Company, 1978, pp. 1–15.

[9] P. Gilmore and C. T. Kelley, *An implicit filtering algorithm for optimization of functions with many local minima*, SIAM J. Optim., 5 (1995), pp. 269–285.

[10] P. Gilmore, C. T. Kelley, C. T. Miller, and G. A. Williams, *Implicit filtering and optimal design problems: Proceedings of the workshop on optimal design and control, Blacksburg VA, April 8–9, 1994*, in Optimal Design and Control, J. Borggaard, J. Burkhardt, M. Gunzburger, and J. Peterson, eds., vol. 19 of Progress in Systems and Control Theory, Birkhäuser, Boston, 1995, pp. 159–176.

[11] P. Gilmore, P. Pernambuco-Wise, and Y. Eyssa, *An optimization code for pulse magnets*, tech. rep., National High Magnetic Field Laboratory, Florida State University, August 1994.

[12] W. Huyer and A. Neumaier, *Global optimization by multilevel coordinate search*, J. Global Optim., 14 (1999), pp. 331–355.

[13] E. Janka, *Vergleich Stochastischer Verfahren zur Globalen Optimierung*, diplomarbeit, Universität Wien, 1999.

[14] C. T. Kelley, *Iterative Methods for Optimization*, no. 18 in Frontiers in Applied Mathematics, SIAM, Philadelphia, 1999.

[15] C. T. Miller, G. A. Williams, and C. T. Kelley, *Transformation approaches for simulating flow in variably saturated porous media*, Tech. Rep. CRSC-TR98-01, North Carolina State University, Center for Research in Scientific Computation, January 1998. Submitted for publication.

[16] P. Pernambuco-Wise, P. Gilmore, B. Lesch, Y. Eyssa, and H. J. S. hneider Muntau, *Systematic failure testing of internally reinforced magnets*, IEEE Transactions on Magnetics, 4 (1996), pp. 2458–2461.

[17] D. Stoneking, G. Bilbro, R. Trew, P. Gilmore, and C. T. Kelley, *Yield optimization using a GaAs process simulator coupled to a physical device model*, IEEE Transactions on Microwave Theory and Techniques, 40 (1992), pp. 1353–1363.

[18] D. E. Stoneking, G. L. Bilbro, R. J. Trew, P. Gilmore, and C. T. Kelley, *Yield optimization using a GaAs process simulator coupled to a physical device model*, in Proceedings IEEE/Cornell Conference on Advanced Concepts in High Speed Devices and Circuits, IEEE, 1991, pp. 374–383.

[19] T. A. Winslow, R. J. Trew, P. Gilmore, and C. T. Kelley, *Doping profiles for optimum class B performance of GaAs mesfet amplifiers*, in Proceedings IEEE/Cornell Conference on Advanced Concepts in High Speed Devices and Circuits, IEEE, 1991, pp. 188–197.

[20] ——, *Simulated performance optimization of GaAs MESFET amplifiers*, in Proceedings IEEE/Cornell Conference on Advanced Concepts in High Speed Devices and Circuits, IEEE, 1991, pp. 393–402.

[21] Y. Yao, *Dynamic Tunneling Algorithm for Global Optimization*, IEEE Transactions on Systems, Man, and Cybernetics, 19 (1989).

Index