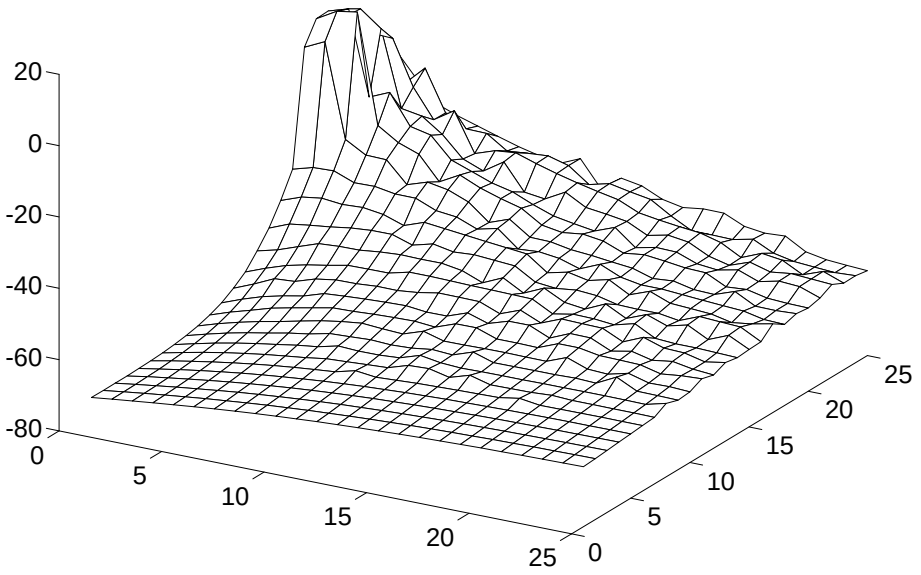

IFFCO: Implicit Filtering for Constrained Optimization, Version 2

Tony Choi, Paul Gilmore, Owen J. Eslinger,
C. T. Kelley, Alton Patrick, and Jörg Gablonsky.
Department of Mathematics
Center for Research in Scientific Computation
North Carolina State University

July 26, 1999

Contents

Preface

This document is a revision of the 1993 guide [9] to version 1 of IFFCO. The major changes in the code are

- inclusion of the BFGS quasi-Newton method,
- addition of several new termination criterion, both at the full iteration level and at the scale level,
- changes in the options and defaults (note the addition of the variable *option* with 4 arguments, changes to *writ*, and *maxit* now has 2 arguments)
- and a parallel version with PVM calls.

The primary contact for IFFCO is

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

Electronic mail is the optimal way to contact Kelley.
The latest versions of IFFCO and this document live on the IFFCO web page:

`http://www4.ncsu.edu/~ctk/iffco.html`

Implicit filtering for unconstrained problems has been implemented in MATLAB (imfil.m) by Kelley and that implementation is also on the IFFCO web page. The MATLAB implementation is not described in this document.

The current keepers of IFFCO are Alton Patrick

`hapatric@unity.ncsu.edu`

and Joerg Gablonsky

`jmgablon@unity.ncsu.edu.`

Patrick is the primary contact for questions about using IFFCO.

This project was supported by National Science Foundation grants #DMS-9700569 and #DMS-9714811.

Chapter 1

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:

$$\min_{x \in Q} \hat{f} : R^n \to R \quad \text{where} \quad Q = \{ x \in R^n \mid l^i \leq x^i \leq u^i \ , i = 1, \ldots, n \} \ , \tag{1.1}$$

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:

$$\hat{f}(x) = f(x) + \phi(x) \ . \tag{1.2}$$

In (1.2) $f(x)$ 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

$$\max_{x \in Q} |\phi(x)| \ll \max_{x \in Q} |f(x)| \ . \tag{1.3}$$

$\phi(x)$ 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 1.2. An example (taken from [19]) of the type of function that IFFCO is designed to minimize is shown on the cover.

Implicit filtering has been successfully applied to problems in semiconductor design [16–19], high-field magnets [4, 12, 15], automotive engineering [5–7], and geosciences [1, 2, 11, 14]. The algorithm used in IFFCO is analyzed in [10, 13]. 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. A more detailed description of the algorithm is given in Section 3.2.

Algorithm 1 BriefIFFCO

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 linesearch in the direction d, and signal success if some criteria are met.
  if linesearch 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 1 is to restart Algorithm 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.

Chapter 2

Using IFFCO

This chapter provides information about using IFFCO, including details on calling IFFCO and selecting appropriate values for parameters. The troubleshooting section describes common problems and advises the user on how to solve them.

To use IFFCO the user must have access to two LINPACK subroutines: dchdc and dposl, and to the BLAS subroutines and functions: daxpy, ddot, and dswap. These subroutines can be found at (http://netlib2.cs.utk.edu/). The user must also provide a subroutine for evaluating the function the user wishes to minimize. This subroutine must be in the format described in Section 2.1.1.

2.1 Calling IFFCO

The calling sequence for IFFCO is

```
call iffco(func,x,u,l,fscale,minh,maxh,n,maxit,restart,writ,termTol,f,
                maxcuts,option)
```

2.1.1 Function and initial iterate

func – is the argument containing the name of the user-supplied subroutine that returns values for the function to be minimized. *func* must be declared external in the calling program.

The name of this subroutine is determined by the user and must be declared in an external statement in the user's calling program. The subroutine should have the form:

```
subroutine  subroutine_name(n,x,f,flag)
integer n,flag
real*8 x(n),f
.
.
.
return
end
```

There are four variables used when calling the subroutine, n, x, f, and $flag$. n is the dimension of the problem, x is the point at which the function should be evaluated, f is the function value that is returned to IFFCO, and $flag$ is a flag. $flag$ should equal zero if the function evaluation was successful, and should be an integer greater than zero if the function failed to evaluate. If that does happen, IFFCO will set the function value to $fscale$.

If there are additional variables needed by *subroutine_name* then those variables can be passed to the subroutine in two different ways. The easiest way to deal with this situation would be to declare these variables as global in the calling routine and in *subroutine_name*. It would then be necessary to declare those variables as global in the subroutines $iffco$ and $funcIF$. If there are reasons against declaring these variables global, then it is still possible to pass these variables onto *subroutine_name*. This can be done, for example, by adding all of the extra variables to the calling sequence of $iffco$. If this is done, then it will also be necessary to adjust the calling sequence of $funcIF$, $gradIF$, and $linesearchIF$ and all of the calls to those subroutines.

x – is a user-supplied double-precision vector of length n. x is the initial point in the iterative process. x must be within the hyper-box defined by the constraints.

2.1.2 Bounds on the variables

u – is a user-supplied double-precision vector of length n. u is the vector containing the upper bounds for the n independent variables. The hyper-box defined by the constraints on the variables is mapped to the unit cube in IFFCO. IFFCO performs all calculations on points within the unit cube. The final solution is mapped back to the original hyper-box before being returned to the user.

l – is a user-supplied double-precision vector of length n. l is the vector containing the lower bounds for the n independent variables.

2.1.3 Scaling the function

fscale – is a user-supplied double-precision constant used to scale the function. $fscale$ should be an approximation to the maximum size in absolute value the function to be minimized obtains in the hyper-box Q. The default value for $fscale$ is 1. If $fscale$ is set to zero, then the default is used.

$fscale$ is also used as the function value when the function to be minimized, $func$, does not return a value. When $func$ fails to return a value, $func$ should return $flag > 0$. See the description of $func$ above.

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. If $fscale$ is too large, the function appears flattened, and IFFCO may not be able to obtain a solution acceptable to the user. Choosing $fscale$ too large often manifests itself by IFFCO obtaining 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 manifests itself by the line search being

unable to find a suitable new point for many point scale pairs. For more details concerning problems with $fscale$ see Section 2.3.

2.1.4 Difference increment limits

$minh$ – is a user-supplied double-precision constant. $minh$ is a lower bound for the last and smallest finite difference step (scale) that IFFCO uses to calculate the finite difference gradient. When a scale is less than $minh$, the algorithm either terminates or restarts. Note, an optimization run is always done with the scale $maxh$ regardless of the value of $minh$. $minh$ must satisfy:

$$0 < minh < maxh \; . \tag{2.1}$$

If $minh$ is too large, the algorithm may not obtain an answer acceptable to the user. 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 $\phi(x)$.

If the amplitude of $\phi(x)$ is approximately constant, then optimally, $minh$ should be a low estimate of $O(\max_{x \in Q}(|\phi(x)/fscale|^{1/3})$. 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 precision. 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.

As mentioned earlier, poor answers may indicate that $minh$ is too large. 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.

Determining a suitable value for $minh$ is a complicated problem involving the curvature of $f(x)$, the amplitude of $\phi(x)$, the amount of accuracy needed by the user, and the cost per function evaluation. Therefore users may have to experiment.

Remember that $minh$ must satisfy Equation 2.1.

$maxh$ – is a user-supplied double-precision constant. $maxh$ is the first and largest scale the algorithm uses to calculate the finite difference gradient. $maxh$ must satisfy:

$$0 < maxh \leq 0.5 \; . \tag{2.2}$$

The upper bound in Inequality 2.2 is necessary to ensure that the finite difference steps taken using the larger scales do not violate the constraints on the variables. The lower bound in Inequality 2.2 ensures that the finite difference gradient is well defined.

If the user chooses $maxh \leq 0$ or $maxh > 0.5$, then an error message will be returned by IFFCO. If $maxh = minh$, IFFCO performs a one scale optimization run using $h = maxh$.

The user should choose a large value for $maxh$ if the initial point may be far from a minimum of $f(x)$, or if the amplitude of $\phi(x)$ is large. Usually, $maxh$ should be a high estimate of $O(\max_{x \in Q} |\phi(x)/fscale|^{1/3})$. However if the user is confident that $\phi(x)$ is small or the initial point is close to an acceptable answer, small values of $maxh$ may be chosen.

If the algorithm is unable to reduce the function while using $maxh$ and other large scales then $maxh$ may be too large. If the user has chosen a small value of $maxh$ and IFFCO is returning answers unacceptable to the user then increasing $maxh$ might help.

Remember that $maxh$ must satisfy Inequality 2.2.

2.1.5 Problem size

n – is the dimension of the problem.

2.1.6 Iteration Limits

maxit – has two arguments.

maxit(1) – specifies the maximum number of iterations the algorithm is allowed to take at a given scale. The default value for $maxit(1)$ is 100, and will be taken if $maxit(1)$ is set to 0.

maxit(2) – specifies the maximum number of function evaluations the algorithm is allowed to take over the course of the entire iteration. In general, if $maxit(2)$ causes the iteration to terminate, the actual number of function evaluations will be close to $maxit(2)$ and not necessarily exactly $maxit(2)$.

restart – Number of restarts to be done.

Restarts should be done only if IFFCO is returning answers that are not as good 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.

2.1.7 Termination

termTol – constant used for determining convergence of the algorithm at a given point for a given scale. Determining appropriate values for *termTol* is problem-dependent. The default value of *termTol* is 1.0 and will be used if *termTol* is set to 0.

Convergence at a point x is determined using,

$$\|x - P(x - d(x))\| \leq termTol \times h \ .$$

Where $P(x - d(x))$ is the projection of the steepest descent step onto the hyper-box Q. At this point no heuristic exists for determining *termTol*. In many computations, using the default value for *termTol* ($termTol = 1$) 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 answer unacceptable to the user. If this phenomena is observed, reducing *termTol* may help. 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. If this phenomena is observed, increasing *termTol* may help.

2.1.8 Line search control

maxcuts – specifies the maximum number of cutbacks in the line search the algorithm may take. The default value for *maxcuts* is 3 and it will be used if *maxcuts* is set to 0.

2.1.9 The *option* Argument

option – has seven arguments.

option(1) – Quasi-Newton update (For default, set to 1.)

option(1) =0 Gradient Projection

option(1) =1 SR1

option(1) =2 BFGS

option(2) – Minimum strategy (For default, set to 2.)

option(2) =0 Do nothing

option(2) =1 Take min as current point at a restart.

option(2) =2 Take min as current point at new scale.

option(2) =3 Take min as current point at new step.

option(3) – Initial Point (For default, set to 1.)

option(3) =0 Use user-supplied x as initial point.

option(3) =1 Use center of box as initial point.

option(4) – Quasi-Newton Variation, B is the approximate inverse Hessian (For default, set to 1.)

option(4) =0 Re-initialize B at each new scale.

option(4) =1 Re-initialize B if active set changes.

option(4) =2 Re-initialize B only if positivity is lost (in SR1 case) or if $y^T * s \leq 0$ (in BFGS case).

2.2 Output

x – is the final point obtained in the optimization process. x should be a good approximation to the global minimum for the function within the hyper-box.

f – is the value of the function at x.

2.2.1 The *writ* variable

IFFCO records the progress of the iteration and it reports on that progress in two ways. The first way is simply writing to standard output. Generally this default medium is the screen, if IFFCO is run interactively, or it is whatever file is designated in a queued run. The second way the progress of the iteration is reported is in the file "iffco.out". This file generally contains more information than the output sent to the screen. To control the amount of information reported, we use the variable *writ*. The following settings apply. "Standard info" refers to iteration number, norm of x, f, norm of g, h, and cutbacks taken (See examples for more details).

writ = 0 - suppress all output
= 1 - standard info
= 2 - standard info + x + #f evals
= 3 - standard info, unscaled
= 4 - standard info + x + #f evals, unscaled
Add 10 to writ for more info to the screen.

Therefore possible settings would be $writ = 0, 1, 2, 3, 4, 11, 12, 13, 14$. If $0 < writ < 10$, the information sent to the screen will be minimal. However, error messages and other vital information will appear. More information will instead be sent to the file "iffco.out". This information is described above. For example, if $writ = 2$ then the standard info will be sent to the screen, as well as x and the number of function evaluations performed. More details are given in Section 2.2.2. If we do as the last line says, and $writ$ is between 11 and 14, we will output more information to the screen. To continue the example just given, assume that we make $writ = 2 + 10 = 12$. Then the standard info (plus x, etc.) will be sent to the file "iffco.out", just as if $writ = 2$. But the standard info will also be sent to the screen, as well as x and the number of function evaluations performed.

2.2.2 Output Files

The current iteration of IFFCO has the ability to output information from a run to text files, namely "iffco.out" and "points.out". This is a guide to interpreting the output. We start with the general information coming out of IFFCO and "iffco.out", and then we will move on to "points.out".

The IFFCO.OUT file

On the page 10 we see an annotated example of an "iffco.out" file. Explanations can be found below. In the example shown, we have set $writ = 4$. This gives the standard information and unscaled x and also the number of function evaluations. See the explanations below for details on how a different setting for $writ$ would change the look to the file.

Key to Annotation IFFCO.OUT

1. The program will begin by listing the values of all of the variables that were input into IFFCO.

2. It continues by listing the initial value of x, as input into the program, as well as the upper and lower bounds on the variables, u and l respectively.

3. Once the main part of the program begins to run, a short history of the iteration is recorded to the file. It does this every time that the line search is successful and upon reduction in the stepsize h. The first line in this section is a header, explaining the

order of the next six numbers. The first number is the number of iterations taken at this scale n, the second $|x|$ is the two-norm of x, the third is the function value f, the fourth is the two-norm of the gradient $|GF|$, the fifth is the value of the current scale h, and the last is a message about the cutbacks under the label *cuts*. A number is returned if the the line search is successful, or a message about the convergence is returned. These numbers correspond to the current "best" point for the iteration.

4. If $writ = 2, 4$ then the components of x are listed explicitly. For other values of $writ$, (4) is skipped entirely.

5. Finally the number of function evaluations performed up to this point in the iteration is recorded.

6. Occasionally during the course of the iteration, messages concerning the progress of the iteration will be displayed when they are reached. Such messages could be similar to the one shown, or could relate other information about the iteration, e.g. "B not positive definite"

7. Steps (3–6) are repeated until the iteration terminates. As mentioned in item 3, parts (3–5) are recorded every time the linesearch is successful and when there is a reduction in h.

8. After the iteration has terminated, the history of the function values is recorded to the file. It does so by listing the f values reported during the run (see item 3) above.

Sample IFFCO.OUT

(1) ↓ fscale = 1.
minh = 1.0000000000000002E-3
maxh = 0.5
n = 14
⋮
ncuts = 10

(2) ↓ x(1) = 0.5000000000E+08
⋮
x(14) = 0.4067844000E+01
u(1) = 0.2500000000E+09
⋮
u(14) = 0.6016695312E+01
l(1) = 0.1000000000E+08
⋮
l(14) = 0.5544986400E+01
option(1) = 0.1000000000E+01
⋮
option(10) = 0.0000000000E+00

There are a total of 0 input errors

---

(3) ↓ m $\|x\|$ f $\|g\|$ h cuts
1 0.6099E+00 0.1834E-01 0.2985E-01 0.5000E+00 0

(4) ↓ x(1) = 0.1370386082E+09
⋮
x(14) = 0.5943178447E+01

(5) Total function evaluations: 30

(6) In line search

(7) ↓ n $\|x\|$ f $\|g\|$ h cuts
⋮
⋮
Total function evaluations: 332

(8) ↓ f history (unscaled)
0.1575815648425897
2.74558305786277192E-2
2.74558305786277192E-2
⋮
8.6319955848176283E-3

The STANDARD OUTPUT: SCREEN

Information sent to standard output generally appears on the screen. This is changed if the program is not run interactively and the queued run sends that output to a file. However, those decisions are up to the user. We only care about the output.

In general, the information sent to standard output will contain very few details about the individual points and their function values. Instead, only vital error messages and a small amount of information are sent there.

As mentioned in Section 2.2.1, we can send more information to the standard output. If we do this, then item 3 below would be replaced by items 3 through 5 from the description of "iffco.out". However, the following should apply (if $writ < 10$), unless the user desires larger amounts of information.

Key to Annotation SCREEN

1. The program will begin by listing the values of all of the variables that were input into IFFCO.

2. It continues by listing the initial value of x, as input into the program, as well as the upper and lower bounds on the variables, u and l respectively.

3. When the scale changes or the linesearch is successful, the number of function evaluations performed up to that point will be listed. If $writ = 12, 14$ then IFFCO will also list the current components of x. See the Annontation for "iffco.out" for more datails.

4. If the scale changes, the value of the new scale will be listed.

5. Occasionally important error messages will appear. The nature of the error messages sent to the screen differs from the error messages sent to the file "iffco.out". Those sent to the screen contain more vital information, e.g. "function failed to evaluate". Although IFFCO deals with this situation, it is important that the user is aware that the function they supplied is not returning a value. Other such messages might contain information about restarts, etc.

6. Steps (3–5) are repeated until the iteration terminates.

7. After the iteration has terminated, the history of the function values is recorded to the screen.

8. If there are errors with the variables sent to IFFCO, then IFFCO will not start up. It will instead report the error to the screen and terminate.

Sample output to SCREEN (success,$writ < 10$)

There are a total of 0 input errors

---

(3) Total function evaluations: 30
(4) New h value = .25d0
(5) ↓ function failed to evaluate!
Setting f := fscale = 7.5
(6) ↓ New h value = .125d0
⋮
⋮
Total function evaluations: 332
(7) ↓ f history (unscaled)
0.1575815648425897
2.7455830578627192E-2
2.7455830578627192E-2
⋮
8.6319955848176283E-3

Sample output to SCREEN (success,$writ > 10$)

(1) ↓
```
fscale      = 1.
minh        = 1.00000000000000002E-3
                ⋮
ncuts       = 10
```
(2) ↓
```
x(1)        = 0.5000000000E+08
                ⋮
x(14)       = 0.4067844000E+01
u(1)        = 0.2500000000E+09
                ⋮
u(14)       = 0.6016695312E+01
l(1)        = 0.1000000000E+08
                ⋮
l(14)       = 0.5544986400E+01
option(1)   = 0.1000000000E+01
                ⋮
option(10)= 0.0000000000E+00
```

There are a total of 0 input errors
---
(3) ↓
```
m     ‖x‖        f      ‖g‖      h     cuts
1 0.6099E+00 0.1834E-01 0.2985E-01 0.5000E+00 0
x(1)        = 0.1370386082E+09
                ⋮
x(14)       = 0.5943178447E+01
Total function evaluations: 30
```
(4) New h value = .25d0

(5) ↓ function failed to evaluate!

Setting f := fscale = 7.5

(6) ↓ New h value = .125d0
```
                ⋮
                ⋮
Total function evaluations: 332
```
(7) ↓ f history (unscaled)
```
0.1575815648425897
2.74558305786277192E-2
2.74558305786277192E-2
                ⋮
8.6319955848176283E-3
```

Sample output to SCREEN (failure,$writ < 10$)

(8) ↓ $minh > maxh$

There are a total of 1 input errors

Sample output to SCREEN (failure,$writ > 10$)

(1) ↓ fscale = 1.
minh = 0.5
maxh = 1.00000000000000002E-3
n = 14
⋮
ncuts = 10
(2) ↓ x(1) = 0.5000000000E+08
⋮
x(14) = 0.4067844000E+01
u(1) = 0.2500000000E+09
⋮
u(14) = 0.6016695312E+01
l(1) = 0.1000000000E+08
⋮
l(14) = 0.5544986400E+01
option(1) = 0.1000000000E+01
⋮
option(10) = 0.0000000000E+00
(8) ↓ $minh > maxh$

There are a total of 1 input errors

The POINTS.OUT file

IFFCO will also record all of the function evaluations performed. It does so through a file called "points.out". In this file there are multiple columns of numbers recorded, and those columns are explained below.

Key to Annotation POINTS.OUT

(1) This column is the output of $nevalsIF$ within IFFCO. Therefore the first line will be the first function evaluation, the second will correspond to the second function evaluation, etc. The order of the function evaluations is totally determined by the order called within IFFCO.
(2) $f * fscale$ - the unscaled function value at this point.
(3) $x(1)$ - This is the first coordinate of x.
(4) $x(2)$ - This is the second coordinate of x.
(5) This continues until the nth component is reached.

Sample POINTS.OUT

(1) ↓	(2) ↓	(3) ↓	(4) ↓
1	0.1000000E+02	0.1000000E+01	0.1396263E-04
2	0.1000000E+07	0.5000000E+00	0.1396263E-04
3	0.1000000E+07	0.1000000E+01	0.5000139E+00
4	0.1000000E+07	0.7500000E+00	0.1396263E-04
5	0.1000000E+07	0.1000000E+01	0.2500139E+00
6	0.8750000E+01	0.8750000E+00	0.1396263E-04
7	0.1000000E+07	0.1000000E+01	0.1250139E+00
8	0.0000000E+00	0.0000000E+00	0.0000000E+00
⋮	⋮	⋮	⋮

2.3 Troubleshooting

This section discusses common problems and gives advice on fixing them. Set $writ > 0$ to obtain a history of the iterative process in the output from IFFCO. If $writ = 1$, then as discussed in Section 2.2.2, IFFCO will display the history of the iteration in the file "iffco.out" without displaying the components of x each time. The first column in the output, labeled with m, gives the number of iterates at the current scale. The second column, labeled with $||x||$ gives the norm of the current point within the unit cube. All norms used are the two-norm divided by the square-root of n. The third column, labeled with f, is the function value at the current iterate divided by $fscale$. The fourth column, labeled with $||g||$, gives the norm of the finite difference gradient, calculated using the current scale, at the current point. The fifth column, labeled with h gives the current scale being used. The sixth column, labeled $cuts$ gives information on the line search. In this column IFFCO either returns the number of cutbacks used in the line search or one of the following two messages:

1. Convergence. This message indicates that the current point is a minimum for the current scale.

2. Line search failed. This message indicates that the line search was unable to find an acceptable new point.

2.3.1 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 failed" in the output from IFFCO.

Line search failures and poor results

If IFFCO is returning answers that are not as good as the user expects 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 failed
  0  0.2357D-01  0.8443D+04  0.1983D+05  0.1250D+00  Line search failed
  0  0.2357D-01  0.8443D+04  0.2125D+05  0.6250D-01  Line search failed
Number of function evaluations:   58
```

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

Line search failures and reasonable results

If IFFCO is returning answers acceptable to the user but IFFCO 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 failed
  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 failed
  0  0.4967D+00  -.9604D-02  0.2067D-01  0.1250D+00  Line search failed
  0  0.4967D+00  -.9604D-02  0.2067D-01  0.6250D-01  Line search failed
Number of function evaluations:   58
```

2.3.2 Line search failure at small scales

If IFFCO is returning answers acceptable to the user but IFFCO 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 failed
  0  0.5027D+00  -.9939D-02  0.5027D-01  0.9766D-03  Line search failed
  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 failed
  0  0.5028D+00  -.9939D-02  0.7184D-01  0.2441D-03  Line search failed
  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 failed
  0  0.5028D+00  -.9940D-02  0.4747D-03  0.6104D-04  Line search failed
  0  0.5028D+00  -.9940D-02  0.4466D-03  0.3052D-04  Line search failed
  0  0.5028D+00  -.9940D-02  0.4395D-03  0.1526D-04  Line search failed
Number of function evaluations:   179
```

2.3.3 Poor Answers

This section discusses what to do if IFFCO is obtaining answers that the user feels are unacceptable but the message "Line search failed" is not reported in the output from IFFCO. To determine if poor answers are being obtained the user must have some idea as to what constitutes a poor answer in their problem.

No convvergence problems and poor results

If IFFCO reports convergence at many scales but poor answers are being obtained then $termTol$ may be too big 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
```

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.

Large *minh*

If IFFCO returns poor answers and *minh* has been set to a value larger than 10^{-4}, the answer returned by IFFCO may not have the accuracy required by the user. Using the default value for *minh* (10^{-4}) or another small value may help. The size of *minh* to use depends on the problem, the accuracy needed by the user, and the precision of the hardware being used.

Chapter 3

Algorithm Description

3.1 Overview

IFFCO is a projected quasi-Newton algorithm that uses a decreasing sequence of finite difference steps (scales) to approximate the gradient. It uses an approximation to the Hessian and a line search algorithm that gives the code global capabilities. IFFCO is based on an algorithm presented in [3].

As mentioned in Chapter 1 the function to be minimized is of the form:

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

$f(x)$ is a smooth function with only a few minima within the hyper-box defined by the constraints. $\phi(x)$ contains the low-amplitude high-frequency terms of $\hat{f}(x)$. $\phi(x)$ creates local minima in $\hat{f}(x)$. These local minima will trap most gradient based algorithms. However by using a sequence of finite difference steps (scales) to approximate the gradient these local minima can be avoided and the global minima can be found up to the level of noise caused by $\phi(x)$. The first elements in the sequence of scales are fairly large, typically half the length of the hyper-box. Thus, the approximate gradient $d(x)$ gives global information about the problem and allows the sequence of quasi-Newton steps to avoid local minima. As the sequence continues, the algorithm becomes more and more of a local algorithm having the fast local convergence properties for which quasi-Newton methods are known.

Because analytic second derivatives may not be available and because the cost of function evaluations may be too high to use a difference Hessian, SR1 (Symmetric Rank one) approximations and BFGS spproximations to the Hessian can be used. The SR1 approximation is given by the formula:

$$S^c = S^- + \frac{rr^T}{r^T s}$$

where S^- is the previous SR1 approximation to the Hessian,

$$s = x^c - x^- \ ,$$

$$r = y - S^- s \ ,$$

$$y = d(x^c) - d(x^-) \ ,$$

and x^- is the iterate previous to x^c the current iterate. The BFGS approximation is given by the formula:

$$B^c = (I - \frac{sy^T}{y^T s})B^-(I - \frac{sy^T}{y^T s}) + \frac{ss^T}{y^T s}$$

where B^- is the previous BFGS approximation. Unlike the SR1 approximation, with BFGS we track the approximate inverse Hessian. Therefore B^c is the current approximation to the inverse Hessian. Here s and y are still defined as in the SR1 case. To accommodate the constraints a reduced approximation is used in both cases. The reduced approximation is obtained by setting the off diagonal elements in the *ith* row and column to zero if the constraints on the *ith* variable are active.

To keep the sequence of quasi-Newton steps within the hyper-box, each step is projected onto the hyper-box using the following formula:

$$P(x)_i = \begin{cases} u_i, & \text{if } x_i > u_i \\ x_i, & \text{if } l_i \leq x_i \leq u_i \\ l_i, & \text{if } x_i < l_i \end{cases} .$$

3.2 Algorithm Outline

In Chapter 1 there was a very brief outline of IFFCO, and Chapter 2 dealt with the arguments sent to IFFCO. In this section there is a more detailed description of the algorithm. Refer back to Chapter 2 for specifics about the variables involved.

Algorithm 2 subroutine iffco

Check for errors in user-supplied parameters.
$h \leftarrow hmax$
Initialize B to be the Identity. If a BFGS update is being used then B is the Approximate Inverse Hessian. If a SR1 update is being used then B is the Approximate Hessian.
Scale x to the unit box.
Find function value at the initial point.
Record point and function value in iteration history.
for $i = 1$ to $restarts(1)$ **do**
  while $h \geq minh$ **do**
    $its \leftarrow 0$
    $ncuts \leftarrow 0$
    Calculate $\nabla_h f(x)$. Refer to Algorithm 3.
    Evaluate the Tolerance $\|x - P(x - \nabla_h f(x))\|$.
    while (Stencil Okay[1]and $ncuts \geq 0$ and $its < maxit(1)$ and
    $\|x - P(x - \nabla_h f(x))\| > termTol \times h$) **do**
      Calculate the negative Quasi-Newton Step p. Refer to Algorithm 6.
      Perform Linesearch. Refer to Algorithm 5.
      if maxit(2) is exceded **then**
        Accept newest point.
        $h \leftarrow hmin/2$
        $ncuts \leftarrow -5$
      else
        if linesearch was successful **then**
          Linesearch returned a new point $x_{linesearch}$.
          $x \leftarrow x_{linesearch}$
          Calculate new $\nabla_h f(x)$. Refer to Algorithm 3.
          Update B. Refer to Algorithm 4.
          Evaluate the new Tolerance $\|x - P(x - \nabla_h f(x))\|$.
          $its \leftarrow its + 1$
          Record point and function value in iteration history.
        else
          Linesearch was unsuccessful, and therefore $ncuts < 0$.
        end if
      end if
    end while
    $h \leftarrow h/2$
    Record point and function value in iteration history.
  end while
end for
Write out function history.
Unscale final iterate and function value, and return to calling program.

[1] The Stencil is the set of points used to compute the difference gradient. A Stencil Failure occurs when all of the points in the (centered difference!) stencil have a larger function value than the central point.

Algorithm 3 subroutine gradIF

for $i = 1$ to n $\{n$ is the dimension of $x\}$ **do**
  Take a forward and backward step in the ith coordinate direction, i.e. $x + he_i$ and $x - he_i$.
  if the forward step violates a constraint **then**
    Use a backwards difference gradient only for the ith direction.
  else if the backward step violates a constraint **then**
    Use a forward difference gradient only for the ith direction.
  else
    Calculate a centered difference gradient for the ith coordinate direction.
  end if
end for

Algorithm 4 subroutine quasiIF

This is done with either a SR1 or a BFGS update, and is done in a manner consistant with the discussion in Section 3.1. Remember, if a BFGS update is being used then B is the Approximate Inverse Hessian. If a SR1 update is being used then B is the Approximate Hessian.

$s = x^c - x^-$
$r = y - B^- s$
$y = d(x^c) - d(x^-)$
if a SR1 Update is to be performed **then**
  $B^c = B^- + \frac{rr^T}{r^T s}$
else
  Perform a BFGS Update
  $B^c = (I - \frac{sy^T}{y^T s})B^-(I - \frac{sy^T}{y^T s}) + \frac{ss^T}{y^T s}$
end if

Algorithm 5 subroutine linesearchIF

$j \leftarrow 1$.
while $j \leq maxcuts$ **do**
  if $j = 1$ **then**
    $\alpha \leftarrow 1$
  else if $P(x - \hat{\alpha}p)$ activates a previously unactivated constraint **then**
    $\alpha \leftarrow 0.5\hat{\alpha}$ $\{\hat{\alpha}$ is the previous step size parameter.$\}$
  else if $\hat{\alpha}$ was the first step size parameter such that $P(x - \hat{\alpha}p)$ did not activate a previously unactivated constraint **then**
    calculate α using a quadratic model of the function. See [8] for details on the formulation of the quadratic model.
  else
    use a cubic model of the function to calculate α. See [8] for details on the formulation of the cubic model.
  end if
  if $\|\alpha p\| < 0.01h$ **then**
    need to exit linesearch and report linesearch failure.
  end if
  Calculate $P(x - \alpha p)$ and $f(P(x - \alpha p))$.
  if $f(P(x - \alpha p)) \leq f(x) - 10^{-4}\alpha d(x)^T p$ **then**
    $x \leftarrow P(x - \alpha p)$
    Return reporting success in the line search.
  end if
  $j \leftarrow j + 1$
end while
Failure Mode
$ncuts \leftarrow -1$

Algorithm 6 subroutine stepIF

p is the Quasi-Newton Step.
if a SR1 update is being used **then**
  if B is positive definite **then**
    $p = -B^{-1}\nabla_h f(x)$
  else
    $p = -\nabla_h f(x)$
  end if
else
  a BFGS update is being used
  if $y^T s > 0$ **then**
    $p = -B\nabla_h f(x)$
  else
    $p = -\nabla_h f(x)$
  end if
end if

Chapter 4

IFFCO in Parallel

We have posted a parallel version of IFFCO on the web page in addition to the serial version. The parallel version of the code is not fully supported. It was included as a means to supply the user with an alternative implementation of the code, and we want this to serve as a model for any parallel implementation the user would like to undertake on their own. However, due to the fact that PVM is handled differently on different machines, the user should not expect this code to work in all environments. The code we have included was written for a Cray T3E, and works on that platform only. Note the Cray T3E allows intrinsic commands, and the message passage is handled in a simple manner. With that machine, k processors are running the code, and each one is assigned a number between 0 and $k-1$. Message passing utilizes these numbers, and this makes assigning different jobs to different processors rather easy. This is not the case on all platforms. We have also run this code in parallel on a loose network of PCs. This can also be effective, but such a switch requires the user to start up the code on each box as well as requiring some changes to the code. The general form of this parallel IFFCO remains the same, but the user will need to create groups within PVM and messages will be passed using group identifiers. The code has also been run on other machines with multiple processors, e.g. an Orgin 2000. This machines also required changes in the code. Instead of starting a group of PCs separately or starting iffco on k processors at once, it was necessary to start the code on one processor and to then spawn other processors off of the first one. Messages are then passed using identification tags between the parents and their children. In addition to this, the user may have to start a PVM daemon running in some environments. Again, the general placement of commands in IFFCO did not change, but it was necessary to change the message passing routines. This should allow the user to see understand why we have included this code, but left it unsupported. We do not discourage users from using a parallel implementation of IFFCO, and we encourage any attempts to use this parallel version as a model. But, we do want to warn the user that each platform is different, and that the probability that changes will have to be made is close to 100%. This code should act as a good model for developing a parallel version that works for the user. We continue, though, with a description of how the parallel version of the code differs from the serial version.

There are probably several clever ways to take advantage of multiple processors with IFFCO. We implemented the two most obvious, and also easiest, of all of these.

The first one of these is implemented in $gradIF$. At each iterate, IFFCO calculates a difference gradient using the current scale h. In the serial version of the code, the difference gradient is calculated one piece at a time. That is, the forward difference step is taken in one coordinate direction and the function is evaluated at that point. Then a backwards difference step is taken, etc. This is then done in all coordinate directions. For one difference gradient, at least n functions evaluations must be performed (if all constraints are violated by a difference step) and at most $2n$ function evaluations will be performed (if no constraints are violated). A significant amount of time could be saved if these $2n$ function evaluations were performed at the same time. This is how we exploited parallelism in IFFCO. When the difference stencil is calculated, each function evaluation is sent to a different processor. More details are given below.

The other implementation of parallelism was added to $linesearchIF$. When a linesearch is performed, as many as $maxcuts$ cutbacks are allowed. In the parallel version of the code, these $maxcuts$ function evaluations are all performed simultaneously. Further, these $maxcuts$ points are then compared and the point with the lowest function value is accepted as the new iterate. If none of the points meet the decrease condition, then the linesearch indicates failure and the scale, h, is decreased.

There are most likely many other areas of IFFCO that could be manipulated to take advantage of the multiple processors. The two most obvious were discussed above, and here we talk about details of their implementation. The user might ask: How many processors should be used? That decision obviously lies with the user. However, we generally used $k + 1$ pocessors, where k divided the dimension n evenly. The extra processor was used as the master processor, as it handled the linear algebra, dealt with calling functions, and collected the returned function values from the other processors. This relation was not strict in any sense. In order to handle the programming aspect of these issues, information was passed between processors using commands from the PVM subroutine libraray. One could also use MPI commands for the same purpose. The code that we have was written for a Cray T3E. Therefore the message passing might vary for other architectures.

Bibliography

[1] K. R. Bailey and B. G. Fitzpatrick. Estimation of groundwater flow parameters using least squares. Technical Report 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. Technical Report 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*, 21:174–184, 1976.

[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*, 211:52–55, 1995.

[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. Technical Report 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. Technical Report 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] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Number 16 in Classics in Applied Mathematics. SIAM, Philadelphia, 1996.

[9] P. Gilmore. IFFCO: Implicit Filtering for Constrained Optimization. Technical Report CRSC-TR93-7, Center for Research in Scientific Computation, North Carolina State University, May 1993. available by anonymous ftp from math.ncsu.edu in FTP/kelley/iffco/ug.ps.

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

[11] 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 J. Borggaard, J. Burkhardt, M. Gunzburger, and J. Peterson, editors, *Optimal Design and Control*, volume 19 of *Progress in Systems and Control Theory*, pages 159–176. Birkhäuser, Boston, 1995.

[12] Paul Gilmore, Paul Pernambuco-Wise, and Yehia Eyssa. An optimization code for pulse magnets. Technical report, National High Magnetic Field Laboratory, Florida State University, August 1994.

[13] C. T. Kelley. *Iterative Methods for Optimization.* Number 18 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 1999.

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

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

[16] 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*, pages 374–383. IEEE, 1991.

[17] 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. *IEEE Transactions on Microwave Theory and Techniques*, 40:1353–1363, 1992.

[18] 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*, pages 188–197. IEEE, 1991.

[19] T. A. Winslow, R. J. Trew, P. Gilmore, and C. T. Kelley. Simulated performance optimization of GaAs MESFET amplifiers. In *Proceedings IEEE/Cornell Conference on Advanced Concepts in High Speed Devices and Circuits*, pages 393–402. IEEE, 1991.

Index