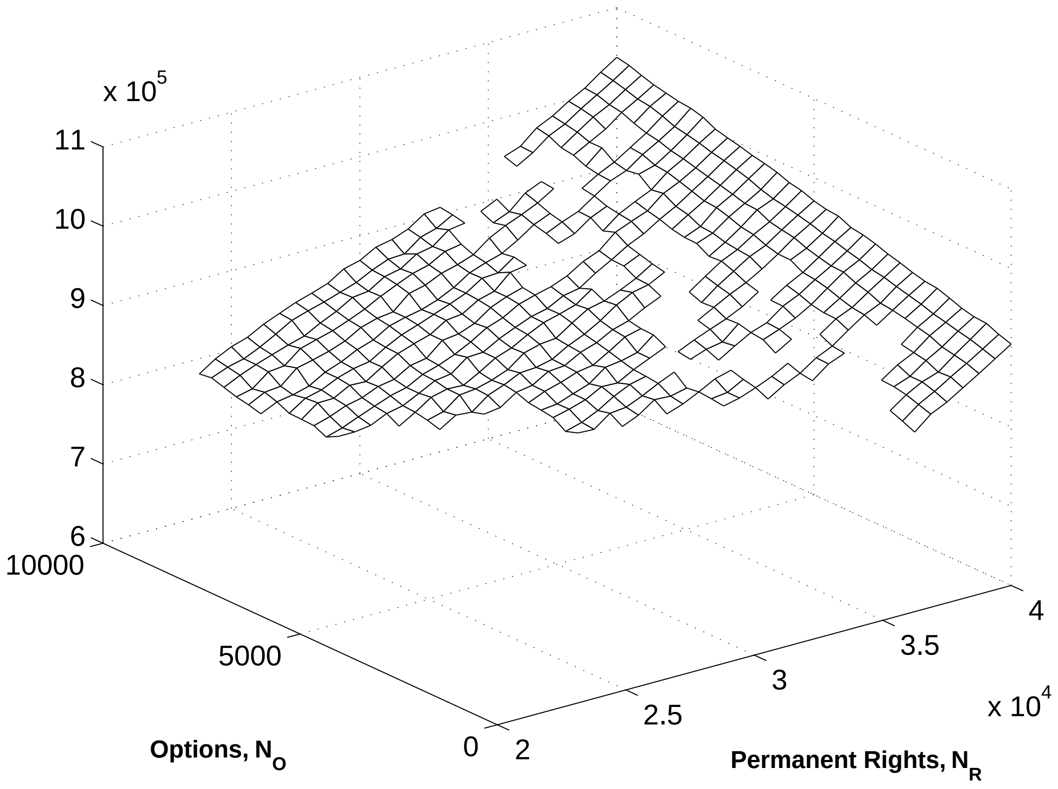


Users' Guide for imfil Version 1.0

C. T. Kelley

Version of May 29, 2011

Contents

Preface

This is the users' guide for the MATLAB version of implicit filtering **imfil.m**. The book [19] is based on this version of the code and contains a much more complete account of **imfil.m**, including a review of the important ideas from traditional optimization, details of the algorithmic decisions, and some of the theory. The code will be updated and maintained, and as that happens this manual will be updated as well. Updates to the book will happen more slowly.

I assume that you have a background in optimization at the level of [10, 18]. If you do not, and simply want to use **imfil.m** as a consumer, I have tried to make that possible, but make no guarantees.

Implicit filtering is a hybrid of a projected quasi-Newton or Gauss-Newton algorithm for bound constrained optimization and nonlinear least squares problems and a deterministic grid-based search algorithm. The gradients for the quasi-Newton method and the Jacobians for the Gauss-Newton iteration are approximated with finite differences, and the difference increment varies as the optimization progresses. The points on the difference stencil are also used to guide a direct search.

Implicit filtering, like coordinate search, is a **sampling method**. Sampling methods control the progress of the optimization by evaluating (sampling) the objective function at feasible points. Sampling methods do not require gradient information, but may, as implicit filtering does, attempt to infer gradient and even Hessian information from the sampling.

imfil.m is a MATLAB implementation of the implicit filtering method. This version differs in significant ways from our older FORTRAN code [7]. This document is a complete reference to Version 1.0 of **imfil.m**, covering installation, testing, and its use in both serial and parallel environments.

As implicit filtering has evolved since its introduction [33], so have several related approaches. The current version of implicit filtering, as reflected in this book and in **imfil.m**, uses ideas from [1, 9, 16, 24].

The development of implicit filtering has been supported by several grants from the National Science Foundation and the Army Research Office, most recently NSF grants DMS-0707220, CDI-0941253, and OCI-0749320, ARO grants W911NF0910159, W911NF-07-1-0112, and W911NF-06-1-0412, and USACE contracts W912HZ-10-P-0221 and W912HZ-10-P-0256.

C. T. Kelley
Raleigh, North Carolina

March, 2011

How to get the software

I maintain and update the software. The examples in this book were done with Version 1.0. The code will evolve as I and others use it in applications, hence we do not include source code as part of the manual.

You can get the most recent archival version of the software from SIAM at

`http://www.siam.org.book.se23/`

I will put slightly more frequent updates on

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

but the SIAM version is the version of record and the one I support.

On those pages you will find

- A pdf file of this document.
- **imfil.m**
  This is the main implicit filtering code.
- **imfil_optset.m** handles the options.
- Several examples within the `Examples` directory:
  - simple example from § 1.4 in the `Simple_Example` subdirectory
  - example for linear constraints from § 3.1 and § 3.4.1 in the `Linear_Constraints` subdirectory
  - case study for parameter identification problem for the simple harmonic oscillator (see Chapter 4) in the `Case_Study_PID` subdirectory
  - case study for the hydrology example from [19] in the `Case_Study_HC` subdirectory
  - case study for the water resources policy example from [19] `Case_Study_Water` subdirectory
- The `Imfil_Tools` directory has examples and useful programs for the advanced options in Chapter 3.

You can download the whole works as a .tar.gz. If you do that the examples are in clearly labeled subdirectories.

One can obtain MATLAB from
The MathWorks, Inc.
3 Apple Hill Dr.
Natick, MA 01760,
(508)653-1415
Fax: (508)653-2997
Email: info@mathworks.com
WWW: http://www.mathworks.com

Chapter 1

Getting Started with imfil.m

In this chapter we give a brief description of what **imfil.m** does, how to install it, and what the computing environment should be. We will illustrate its use for two simple problems. We do not describe all the options, the details of the algorithms, or the ways to extend the code on your own. Chapter 2 is the complete reference manual for **imfil.m**.

1.1 Computing Environment and Installation

In order to use **imfil.m**, you will need to get the software, put the codes in your MATLAB path, and be running a recent version of MATLAB. We have tested **imfil.m** on versions 6.5 and higher. Higher is better, especially if you want to use the parallel toolbox.

imfil.m uses very little memory on its own. The codes which define your problem may use much more. MATLAB will complain if it runs out of memory, which is less likely if you run version 7.5 or later.

Installation is easy. Download the MATLAB files for **imfil.m** from

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

Then put **imfil.m** and **imfil_optset.m** in a directory and put that directory in your MATLAB path.

1.2 What imfil.m does

Implicit filtering solves **bound constrained optimization** problems :

$$\min_{x \in \Omega} f(x), \tag{1.1}$$

by which we mean that the goal is to minimize the **objective function** f subject to the condition that $x \in R^N$ is in the **feasible region** (or **nominal design space**)

$$\Omega = \{x \in R^N \,|\, L_i \leq (x)_i \leq U_i\}, \tag{1.2}$$

which is a **hyperrectangle** in R^N. **imfil.m** is a MATLAB implementation of implicit filtering.

Implicit filtering, and the other methods that are derived from coordinate search, are best used in cases where f is either not smooth, not everywhere defined, discontinuous, or when derivatives of f are too costly to obtain. The motivating examples for the construction of implicit filtering were problems in which f was a smooth function corrupted by low-amplitude, high-frequency noise, or which was not defined (*i.e.* the code for computing f failed) at many points in the nominal design space Ω.

In the classical nonlinear programming problem [12,15,28] f is a smooth (*i.e.* twice Lipschitz continuously differentiable) function and Ω can be described by smooth inequality constraints, *i.e.*

$$\Omega_C = \{x \in R^N \,|\, c_i(x) \leq 0, 1 \leq i \leq P\}. \tag{1.3}$$

There are several good gradient-based methods and codes for solving this classical problem [3,4,6,8,14,32]. Sampling methods such as implicit filtering are not among them, and one should use a gradient-based code for such problems.

Implicit filtering is a **sampling method**. By this we mean that the optimization is controlled only by evaluating f at a cluster of points in Ω. That evaluation determines the next cluster. Implicit filtering's samples are arranged on a stencil, and it is important to understand how that stencil is built. We begin with a current iterate x_c and the value of the function $f(x_c)$. Then, the default algorithm is to sample the $2N$ points

$$x_c \pm h v_i \, 1 \leq i \leq N,$$

where

$$v_i = (L_i - U_i) e_i,$$

e_i is the unit vector in the ith coordinate direction, and h, the **scale** varies as the optimization progresses. The default sequence of scales is

$$\{2^{-n}\}_{n=\texttt{scalestart}}^{\texttt{scaledepth}}.$$

The algorithmic parameters `scaledepth` and `scalestart` can be changed from the defaults of 7 and 1 with the `imfil_optset` command. The optimization will terminate when the sequence of scales has been exhausted.

imfil.m uses the values of f on the stencil in several ways, one of which is to construct a difference gradient and use that in a quasi-Newton method. **imfil.m** reports results after each quasi-Newton iteration is complete. When the supply of scales has been exhausted, the optimization terminates.

imfil.m scales the bounds by changing variables so that $L_i = 0$ and $U_i = 1$ for all i. Scaling helps **imfil.m** take steps of relatively equal size in all the variables. **You do not have to scale the variables. imfil.m does that for you.** The scaling of x is transparent to you unless you use the `executive_function` (§ 3.5) or `explore_function` options (see Chapter 3).

1.2.1 Constraints

Implicit filtering is able to respond to the function's failure to return a value. When this happens, we say that a **hidden constraint** has been violated. **imfil.m** treats a point in Ω for which f has no value as missing data, and will proceed without the value. Your implementation of f (see § 1.2.3 and 2.2.2) must communicate a failure to **imfil.m**.

Explicit constraints are those that can be evaluated by simply testing the variables and not calling an expensive simulator within f. These are the kinds of constraints one sees in nonlinear programming

$$c_i(x) \le 0, 1 \le i \le P,$$

where $c_i : R^N \to R$ and the inequality is understood componentwise. If you have explicit constraints you must communicate infeasibility to **imfil.m** by signaling failure.

You could also use a penalty function to inform **imfil.m** about explicit constraints [12, 28]. In this approach one replaces f by

$$f_p(x) = f(x) + p(x)$$

where p is an **penalty function** which measures the deviation from feasibility. For example, the **exact l_1 penalty function** for smooth inequality constraints is

$$p(x) = \frac{1}{\mu} \sum_{i=1}^{P} \max(-c_i(x), 0).$$

Here μ is the **penalty parameter**. Selecting μ requires some thought [28].

Constraints can cause problems for stencil-based sampling methods by hiding descent directions from the stencil, and enriching the set of directions is necessary for convergence theory [1, 24] and useful in practice as well. The `add_new_directions` (§ 3.1), `vstencil` (§ 2.9.1), and `random_stencil` (§ 2.9.2) options to **imfil.m** are three ways to do this.

1.2.2 The Budget for the Iteration

The most common way to terminate a sampling algorithm is to assign a **budget** of function evaluations to the optimization, and to stop the computation when that budget is exceeded. When the function may fail, keeping track of the budget requires more care, and your code for f must help **imfil.m** with that. One thing to consider, for example, is that sometimes a failed point is significantly cheaper to detect than a complete call to f.

So, at a minimum, you must give **imfil.m** an initial iterate, the objective function, the function value at the inital iterate, the budget, and the bounds. **imfil.m** will return the optimal point x, and (optionally) a history of the iteration. You can use the history to evaluate the performance of the algorithm or to understand what has happened if the iteration stagnates.

Assigning the budget can be tricky. If the budget is too large, the iteration will waste function evaluations while making very little progress. A small budget, on the other hand, can clearly hide a good solution. We illustrate the effects of poorly sized budgets in § 4 and § 4.4.

1.2.3 The Objective Function

You must write a MATLAB code for f, which will take as its input $x \in R^N$ and return

- a value $fout = f(x)$,
- a flag $ifail$ to signal a failed evaluation ($ifail = 0$ unless the evaluation fails, if the evaluation fails set $ifail = 1$ and $fout = NaN$), and
- $icount$, an estimate of the cost.

The NaN notation ("not a number") comes from the IEEE floating point standard [17, 29]. We use it to indicate missing data in a way that allows MATLAB to propagate it through the computation.

So, the call to f would look like

```
[fout,ifail,icount]=f(x)
```

If your function never fails and the cost of evaluation is independent of x, you can omit the $ifail$ and $icount$ arguments by setting the `simple_function` option to 1. After doing that you may use a function with only one output argument. Use

```
options=imfil_optset('simple_function',1);
```

and then **imfil.m** will accept

```
fout=f(x),
```

and set $ifail = 0$ and $icount = 1$. If you use the `parallel` option, **imfil.m** will count the evaluations correctly.

1.3 Basic Usage

At a minimum, **imfil.m** requires the objective function f, the bounds in an $N \times 2$ array, with L in the first column and U in the second, and a budget. **imfil.m** will examine a cumulative cost estimate (which uses $icount$) and terminate the optimization when the budget is exceeded. **imfil.m** will not interrupt an iteration in the middle, so you should expect a modest overshoot in the cost of the optimization. **imfil.m** will also terminate when the list of scales has been exhausted. We describe other ways to terminate the iteration in § 2.10 and § 2.11.

A complete call would look like

```
x=imfil(x0,f,budget,bounds);
```

or

```
[x,histout]=imfil(x0,f,budget,bounds);
```

if you want the history of the iteration. We will use the `histout` array in the examples in this chapter. You can get more information by asking for the `complete_history` structure. We explain the details of the `histout` array and the `complete_history` structure in § 2.3.1 and § 2.3.2.

Remember that if your objective function is a MATLAB .m file, say `myfun.m`, you'll use the MATLAB function handle notation (the @ symbol) before name of the function. Then the call would look like

```
x=imfil(x0,@myfun,budget,bounds);
```

`myfun.m` would have to be either in your MATLAB path on in the current directory. Note that we call the function with a MATLAB function handle, rather than using the name of the file in quotes. The reason for this is accommodation of the way way MATLAB handles optional extra arguments to functions.

The `histout` array is an $IT \times (N + 5)$ dimensional array, where IT is simply a counter of the number of times the array is updated. The histout array is created after the first function evaluation and updated after each approximate gradient computation. For now we will concentrate on the first two columns, which contain the cumulative number of function evaluations $fcount$ and the value of f at the end of the iteration.

1.3.1 Termination

There are two iterations which require termination parameters. The **inner iteration** is the quasi-Newton iteration for each value of h. The **outer iteration** is the implicit filtering iteration. In this section we will explain the default termination criteria and list some other ways to terminate these iterations. The details are in § 2.10.

The inner iteration will terminate

- if the value of f at the current point is smaller than the values elsewhere on the finite difference stencil, a condition we will call **stencil failure**,
- if the internal termination criteria of the quasi-Newton iteration are satisfied.

One can tune both of these criteria, and a user interested in doing that should look at [19] to understand the details.

The outer iteration, by default, terminates when either

- a budget of calls to the function has been exceeded or
- the list of scales has been exhausted.

The budget is an input argument to **imfil.m** and this mode of termination usually works well. One can do more, and set various options to terminate the iteration when

- the function value has been decreased to a desired target or
- the variation in the function on the stencil is sufficiently small.

See § 2.10 for the details.

1.4 A Very Simple Example

The files for this example are in the `Examples/Simple_Example` directory of the software collection.

In this section we apply **imfil.m** to a simple example to show you how to set up the data and look at the results. We will minimize

$$f(x_1, x_2) = (x_1^2 + x_2^2) * (1 + .1 * \sin(10 * (x_1 + x_2)))$$

subject to the bound constraints

$$-1 \leq x_1, x_2 \leq 1.$$

It's not hard to see that the optimal point is $x^* = 0$.

To begin we write a MATLAB .m file for f, which we will call `f_easy.m`. The .m file is

```
function [fv,ifail,icount]=f_easy(x)
% F_EASY
% Simple example of using imfil.m
%
fv=x'*x;
fv=fv*(1 + .1*sin(10 * (x(1) + x(2) ) ));
%
% This function never fails to return a value
%
ifail=0;
%
% and every call to the function has the same cost.
%
icount=1;
```

Note that we include the $ifail$ and $icount$ in the output arguments to `f_easy.m`, even though they are not really needed. We could avoid that by using the `simple_function` option.

We will use **imfil.m** to minimize *f_easy* and then use the `histout` array to study the details of the iteration. To use **imfil.m** we will need to specify a budget and an initial iterate, which in this example are

$$x_0 = (.5, .5)^T \text{ and } budget = 40.$$

Our code `driver_easy.m` runs **imfil.m** and then prints the first two columns of the `histout` array.

```
function [x,histout]=driver_easy;
% DRIVER_EASY
% Minimize f_easy with imfil.m
%
% Set the bounds, budget, and initial iterate.
bounds=[-1, 1; -1 1];
budget=40;
x0=[.5,.5]';
%
% Call imfil.
%
[x,histout]=imfil(x0,@f_easy,budget,bounds);
%
% Use the first two columns of the histout array to examine the
% progress of the iteration.
%
histout(:,1:2)
```

The output directly from MATLAB is

```
   1.0000e+00   4.7280e-01
   3.0000e+00   4.7280e-01
   8.0000e+00   4.7280e-01
   1.5000e+01   2.6572e-01
   2.0000e+01   9.6363e-04
   2.5000e+01   9.6363e-04
   3.0000e+01   9.6363e-04
   3.5000e+01   9.6363e-04
   4.0000e+01   5.7334e-04
   4.5000e+01   1.2430e-04
```

The call to **imfil.m** returns

$$x = (8.8 \times 10^{-3}, -6.8 \times 10^{-3})^T$$

as the solution.

The first column is the function evaluation counter, the second the value of the function, and the third the norm of the approximation of the gradient **imfil.m** computes using the function values on the stencil. One might think that the function evaluation counter should increase by at least four with each iteration, since the stencil has four points. However, if a point in the stencil is infeasible, as two are in the first iteration, the evaluation is skipped. Hence on the first iteration **imfil.m** reports the function value at the initial iterate and the norm of an approximate gradient based on three points (the initial iterate and the two feasible points in the stencil).

We see a decrease in the function in the second column in the early phase of the iteration, as one would expect. Note also that there is a middle part of the iteration where no visible progress is made. This "flat spot" is, unfortunately, common in sampling methods. At the end, the function decreases again. Had we terminated after 20 iterations, we would have missed this improvement. If we increase the budget and the number of scales (see Chapter 4 and § 1.5.4) we'd see further improvement. The `histout` array for this computation indicates the progress.

Note that the iteration terminated over budget. The reason for this is that the function evaluation counter is compared to the budget only after each iteration, so one may expect to exceed the budget by a bit.

1.5 Setting Options

You can set several algorithmic parameters with the `imfil_optset` options command. Many of these are rarely needed or are intended for the specialist. We will discuss only the most useful and important in this section. We will explain the details for all the options in Chapter 2.

If you want to accept the default options, you need do nothing. If you want to explicitly modify the default options structure, you can get if from the `imfil_optset` command by calling that command with no arguments.

```
options=imfil_optset;
```

You need only do this once; additional calls to `imfil_optset` will update the the options structure you've already created. For example, if you want to change `scalestart` to 3 and `scaledepth` to 10, you could call `imfil_optset` three times:

```
options=imfil_optset;
options=imfil_optset('scalestart',3,options);
options=imfil_optset('scaledepth',10,options);
```

prior to the call to **imfil.m**. You can also put all three of the calls to `imfil_optset` in the code fragment above on a single line

```
options=imfil_optset('scalestart',3,'scaledepth',10);
```

If you want to change an existing set of options, you would add the name of the options structure to the `imfil_optset` command. For example, to change `scaledepth` from 10 to 8, in the options structure you created with the call to `imfil_optset` above, the call would be

```
options=imfil_optset('scaledepth',8,options);
```

You might try to modify `driver_easy.m` by increasing the budget and the number of scales. If you change the call to **imfil.m** to

```
options=imfil_optset('scaledepth,20);
bounds=[-1, 1; -1 1];
budget=100;
```

```
x0=[.5,.5]';
%
% Call imfil.
%
[x,histout,complete_history]=imfil(x0,@f_easy,budget,bounds,options);
```

you will see a smaller function value and one more flat spot. You might also try the `smooth_problem` (see § 2.6.4) option.

1.5.1 Nonlinear Least Squares

Many problems, such as the example in Chapter 4, are best formulated as nonlinear least squares problems, where F returns an vector of residuals in R^M and the function to be minimized is

$$f(x) = \|F(x)\|^2/2 = F(x)^T F(x)/2. \tag{1.4}$$

You can tell **imfil.m** that your problem is a nonlinear least squares problem by setting the `least_squares` option to 1 with the command

```
options=imfil_optset('least_squares',1);
```

If you do this you need to write your function so that $F \in R^M$ is returned. **imfil.m** will construct $f(x) = F(x)^T F(x)/2$ for you. The optimization method is also tuned to a nonlinear least squares computation, and the underlying method is a damped finite-difference Gauss-Newton iteration [10, 18]. Chapter 4 has a simple nonlinear least squares example.

1.5.2 Parallel Computing

The `parallel` option tells **imfil.m** that f can be called with multiple arguments, and will return a matrix whose columns are the values of f, $ifail$, and $icount$. So if x is an $N \times P$ array of P arguments to f and `parallel` is set to 1, a call to $f(x)$ will return three $1 \times P$ vectors of values and flags. It is the your responsibility to write f to do the parallel evaluation in an efficient way. Our example of a parallel call in § 4.2 shows how **imfil.m** responds to

```
options=imfil_optset('parallel',1);
```

If you are solving a nonlinear least squares problem, where a call to f returns an $M \times 1$ column vector, your parallel function should return an $M \times P$ array of residual values as well as vectors $iflag$ and $icount$. The parallel algorithm is not the same as the serial method because all the line search possibilities are examined at the same time (see § 2.7 for the details). One implication of this is that more function evaluations can be used even if the final result is the same as in the serial case and the total runtime is significantly less. One should interpret graphs like Figure 4.1 with care when one does the function evaluations in parallel. The default is $parallel = 0$.

The latest versions of MATLAB support some parallelism. The MATLAB parallel computing toolbox **matlabpool** command lets you build a pool of "workers" or "labs", which are separate copies of MATLAB running on each core of a multicore computer. The toolbox also provides the `parfor` loop. A `parfor` loop executes each iteration of the loop on a separate worker, if an idle worker is available. For example, suppose you have an 8 core computer and want to evaluate f at several points $\{x_i\}_{i=1}^k \subset R^1$, you might do something like

```
parfor i=1:k
   f(i) = f(x(i));
end
```

after calling `matlabpool(8)` **once** for your MATLAB session. This, in fact, is how one would use `parfor`, but you must pay attention to global variables and memory conflicts among statements in the loop. The example in Chapter 4 uses the `parfor` loop.

Here is an example of a MATLAB session which calls `f_easy` 16 times in parallel on an 8 core computer. After opening MATLAB we begin with a `matlabpool` command to open 8 labs:

```
>> matlabpool(8)
Starting matlabpool using the 'local' configuration
... connected to 8 labs.
```

If you use `matlabpool` while labs are open from a previous call, MATLAB will complain, and you should close all open labs with `matlabpool close`. Our code `parallel_easy` then uses a `parfor` loop to evaluate `f_easy` at some random points and print the results. The codes are in the subdirectory `Examples/Simple_Example` in the software collection.

```
% PARALLEL EASY
%
% A simple matlab parfor example.
% You must call matlabpool if you want this to run in parallel.
%
x=rand(2,16);
f=zeros(16,1);
parfor i=1:16
   f(i) = feval(@f_easy,x(:,i));
end
f
```

The `parallel_easy` script could easily be converted into a parallel version of `f_easy`

```
function [fv, ifail, icount]=f_easy_p(x)
% F_EASY_P
%
```

```
% A parallel version of f_easy.
% You must call matlabpool if you want this to run in parallel.
%
%function [fv, ifail, icount]=f_easy_p(x)
%
% fv must be a ROW vector. This makes scalar optimization consistent
% with what imfil does for nonlinear least squares.
%
% If you make fv a column vector, you will get some very interesting
% error messages.
%
[nr,nc]=size(x);
fv=zeros(1,nc);
%
% Like f_easy, this function never fails and all calls to f have the
% same cost. However, we have to count the number of calls.
%
ifail=zeros(nc,1);
icount=nc*ones(nc,1);
%
parfor i=1:nc
   fv(i) = feval(@f_easy,x(:,i));
end
```

MATLAB does the sensible thing if you don't have the parallel toolbox and simply executes a `for` loop. This means that `f_easy_p` will simulate parallel execution even if it actually works in serial mode.

To change `driver_easy.m` into a code which calls **imfil.m** in parallel, we change the call to **imfil.m** to

```
%
% Turn the parallel option on.
%
options=imfil_optset('parallel',1);
%
% Call imfil.
%
[x,histout,complete_history]=imfil(x0,@f_easy_p,budget,bounds,options);
%
```

to build `driver_easy_p.m`, the parallel version of `driver_easy.m`.

The output is a little different from that of the serial code.

```
   1.0000e+00   4.7280e-01
   3.0000e+00   4.7280e-01
   8.0000e+00   4.7280e-01
   1.6000e+01   7.3599e-03
   2.1000e+01   7.3599e-03
   2.6000e+01   7.3599e-03
   3.1000e+01   7.3599e-03
   3.9000e+01   1.5944e-05
   4.4000e+01   1.5944e-05
```

because the parallel and serial versions are slightly different algorithms (see § 2.7).

There are also some very useful resources in the **MATLAB Central File Exchange**. This is a software repository maintained by the Mathworks at

`http://www.mathworks.com/matlabcentral/fileexchange/`

MULTICORE [5] is a package that lets you use multiple cores with MATLAB. Each core runs its own copy of MATLAB. The package moves data between cores with file I/O, an approach with can slow down the computation if function calls are very inexpensive. The MATLAB `parfor` construct uses memory for the interprocess communication, and is significantly faster. However, MULTICORE is free. The software associated with [20] has the `pRUN` program, which allows you to run the same MATLAB code on multiple processors. These approaches do not support fine-grained parallelism (*i.e.* the use of many processors to speed up the internal computations within f), but should work well for very expensive function evaluations.

1.5.3 Scaling f

If the values of $|f|$ are very small or very large, the quality of the difference gradient which **imfil.m** uses in its search can be poor. **imfil.m** attempts to solve this problem by **scaling** the function by dividing it by the size of a "typical value". Unless you tell **imfil.m** otherwise, this value is 1.2 times the absolute value of the value at the initial iterate.

You can change this by setting the `fscale` option. Setting `fscale` to a negative value will tell **imfil.m** to use $|fscale| \times |f(x_0)|$ as the typical value for f. Setting `fscale` to a positive value will tell **imfil.m** to use $fscale$ as the typical value. If you blunder and set $fscale = 0$, **imfil.m** will restore the default. If $f(x_0) = 0$ then **imfil.m** will set $fscale$ to 1. See § 2.5.1 for more details on `fscale` and its role in **imfil.m**.

Scaling f to order 1 means that we can compare the variation in f (or the change in f from one iteration to the next) to a tolerance (which may depend on the scale) and make a termination decision. See § 2.10 for the details and § 4.4.1 for an example.

1.5.4 Changing the Scales

imfil.m uses a stencil that is build from the bounds. If your current point is x_c, **imfil.m**'s default behavior is to sample the $2N$ points

$$x_c \pm h(L_i - U_i)e_i \, 1 \leq i \leq N, \tag{1.5}$$

where e_i is the unit vector in the ith coordinate direction, and h, the **scale** varies as the optimization progresses. Implicit in the definition of the stencil (1.5) is the finiteness of the bounds.

The sequence of scale is

$$\{2^{-n}\}_{n=\texttt{scalestart}}^{\texttt{scaledepth}}.$$

`scalestart` and `scaledepth` can be changed with the options command. The defaults are *scalestart* = 1 and *scaledepth* = 7. You can use your own array of scales with the option.

1.5.5 Looking at the Iteration History

You have already seen how the `histout` array can be used to examine the performance of the iteration. **imfil.m** maintains an internal `complete_history` structure which contains the entire history of the iteration (see § 2.3.2). You can access that data either as an optional output argument to **imfil.m** or within the iteration by using one of the advanced options (see Chapter 3).

1.5.6 Scale-Aware Functions

Your function may be able to adjust its own accuracy or resolution. In this case we will say that your function is **scale-aware** . One example of this possibility is if the tolerance in a solver can be reduced as the scale is reduced. This is the way in which **imfil.m** can be used as a a multi-fidelity solver. If your function has this capability, you may enable communication between **imfil.m** and the function call by adding the scale as an extra argument to f, making the call look like

```
[fout,ifail,icount]=f(x,h)
```

You must tell **imfil.m** that f is taking the extra argument by setting the `scale_aware` option to 1, the default is 0. See § 2.6.3 and § 4.3 for examples.

1.6 Passing Data to the Function

You may need to pass data from your calling program directly to f. For example, the data for a nonlinear least squares problem is part of the least squares residual, but you may not want to hard-code that data into the function. **imfil.m** permits an optional final argument which you may use for that purpose. The calling sequence looks like

```
[x,histout]=imfil(x0,f,budget,bounds,options,extra_data);
```

Here `extra_data` can be an array, a function handle, or a structure (see 2.8 for more details). Chapter 4 an example of how this can be used.

You might think that you could use MATLAB global variables for this purpose. However, global variables can cause problems with parallel computing in MATLAB, and we recommend that you avoid them.

Chapter 2

Using imfil.m

This chapter is about **imfil.m** and its use. As in the earlier chapters, the notation in the code fragments is different from the mathematical notation in the text. So, for example, the initial iterate in the code is `x0` instead of x_0, which is the notation we will use in the text.

The full calling sequence for **imfil.m** is

```
[x,histout,complete_history]=
          imfil(x0,f,budget,bounds,options,extra_data);
```

The last two input arguments `options` (§ 2.4) and `extra_data` (§ 2.8) are optional. You must use the `options` argument if you want to use the `extra_data` argument. If the `options` argument is omitted, **imfil.m** will use the defaults for `options` and make `extra_data` an empty array.

The output argument `complete_history` is also optional, but is useful for debugging and performance analysis.

2.1 Installation and Testing

Download the MATLAB files for **imfil.m** from

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

or

```
http://www.siam.org.book.se23/
```

Put **imfil.m** and **imfil_optset.m** in a directory and put that directory in your MATLAB path.

2.2 Input

The input data are

- $x_0 \in R^N$: the initial iterate,
- `f` : $R^N \to R^M$: the objective function f if $M = 1$ or, in the case of nonlinear least squares problems with $M > 1$, the nonlinear residual F,
- *budget*, the maximum number of function evaluations allowed to the optimization,
- the bounds array `bounds`, and
- the `options` structure.
- the `extra_data` structure.

We will discuss all but the `options` and `extra_data` arguments in this section. We will explain the `options` at length in § 2.4 and how to send extra data to the function in § 2.8.

2.2.1 The Initial Iterate

imfil.m requires a feasible initial iterate. This means that $x0$ must satisfy the bound constraints, *i.e.*

$$bounds(j, 1) \leq x_0(j) \leq bounds(j, 2),$$

for all j, and that $f(x_0)$ must be defined, *i.e.* f will return a value for $x0$ with $ifail = 0$.

2.2.2 The Input Function `f`

If the `simple_function` option is off (*i.e.* $= 0$), the calling sequence for tt f should be

```
[fout,ifail,icount]=f(x);
```

or

```
[fout,ifail,icount]=f(x,h);
```

if your function is **scale-aware** (see § 2.6.3) *i.e.* can use the scale to manage its own internal control of accuracy. If your function is scale-aware, set the `scale_aware` . option to 1.

You may omit the $ifail$ and $icount$ arguments if you set the `simple_function` option to 1. In that event, the calling sequence is

```
fout=f(x);
```

or

```
fout=f(x,h).
```

In most cases the procedures for optimization problems and nonlinear least squares problems are the same, so we will express things in terms of f, the objective function for an optimization problem. When the difference between optimization and nonlinear least squares is important, we will discuss both cases.

If $f(x)$ successfully returns a value, $fout = f(x)$ should be that value, the failure flag $ifail$ should be 0, and $icount$ should be an estimate of the cost. **imfil.m** uses $icount$ when comparing the cost of the optimization to the `budget` and to build the first column of the `histout` array, and you have the flexibility to assign non-integer values to $icount$. If, for example, a function call fails after performing half of the normal work, you might set $icount = .5$.

$ifail = 1$ is the signal that the function cannot return a value, *i.e.* a **hidden constraint** has been violated. You must return a NaN as the value when this happens. **imfil.m** will eliminate failed points from the stencil when computing the stencil gradient.

If the `parallel` option is on (*i.e.* set to 1, 'on', or 'yes'), then **imfil.m** will send an array of input arguments to f. For the evaluation of the stencil derivative, **imfil.m** will send the elements of the stencil that do not violate the bound constraints to f before it computes the stencil gradient. During the line search, **imfil.m** will send every point that could be queried in the line search to f all at once; the default being the four points $\{x + \lambda d\}$ for $\lambda = 1, 1/2, 1/4, 1/8$. You can change this by setting the `maxitarm` option (see § 2.11.2).

Your parallel function must be able to accept an $N \times P$ array of P arguments to f, and return three $P \times 1$ arrays of values for $fout$, $ifail$, and $icount$. If your are solving a least-squares problem where $F : R^N \rightarrow R^M$, then $fout$ should be $M \times P$. It is your job to construct your function to use what parallelism you have efficiently. The `simple_function` option does the right thing when the `parallel` option is on. If you send f P vectors, then **imfil.m** will set $icount = P$.

2.2.3 The Budget

The optimization will terminate when the cumulative cost (as measured by $icost$) exceeds the `budget`. A budget that is too small will force premature termination (as will a list of scales that is too short). A budget that is too large will waste function evaluations and the iteration will make very little progress in the latter stage (see the discussion in § 4.4). The optimization is likely to finish over budget because **imfil.m** does not stop the outer (optimization) loop in mid-stream. The example in Chapter 4 shows how to set the budget and some effects of making the budget (or the number of scales) too small or too large.

2.2.4 The Bounds

The `bounds` array is a $N \times 2$ array with the lower bounds in the first column and the upper bounds in the second column. For example, if $N = 100$ and the bounds are $2 \leq x(i) \leq 3$, you would use

```
bounds(:,1)=2*ones(100,1); bounds(:,2)=3*ones(100,1);
```

Keep in mind that **imfil.m** requires you to provide finite bounds for all the variables.

2.3 Output and Troubleshooting

The output of **imfil.m** includes x, an approximation of the solution, and two optional ways to look at the history of the iteration. The `histout` array is a simple an optional iteration history. The `complete_history` structure contains every point where **imfil.m** has evaluated f and either the value of f or a failure flag.

2.3.1 The `histout` array

The `histout` array is an $IT \times (N+5)$ dimensional array, where IT is simply a counter of the number of times the array is updated. The histout array is created after the first function evaluation and updated with a new row after each approximate gradient computation.

For optimization problems

$$histout(:, i) = [fcount, fval, \|\nabla f(x, V, h)\|, \|s\|, iarm, x^T]$$

and for nonlinear least squares

$$histout(:, i) = [fcount, F(x)^T F(x)/2, \|DF(x, V, h)^T F(x)\|, \|s\|, iarm, x^T].$$

For each iteration (row) the first five elements are $fcount$, the number of function evaluations so far (the sum of *icount* from each call to f), $fval$, the current value of the objective function, the norm of the stencil gradient, the norm of the step, and `iarm` the number of times the steplength was reduced in the line search for that iteration. The remaining N elements are x^T, where x is the current iteration. We used the `histout` array for the iteration history plots in the book. Keep in mind that the norm of the step is reported in **imfil.m**'s internal scaling (*i.e.* bounds between 0 and 1).

When **imfil.m** reduces the scale after a stencil failure, **imfil.m** sets `iarm` $= -1$ in the `histout` to indicate that no quasi-Newton work at all was done.

The example `pid_example_chapter_1.m` in the `Examples/Case_Study_PID` directory of the software collection shows how to use the `histout` array to plot the iteration history.

2.3.2 The complete_history Structure

The `complete_history` structure records the successful points (*i.e.* those for which f returns a value), the values at the successful points, and the points where f failed to return a value. The fields in the structure are `complete_history.good_points`, `complete_history.good_values`, and `complete_history.failed_points`.

imfil.m uses the complete history structure internally to avoid evaluation of f at the same point more than once. This is a possibility if the poll of the points on the stencil is finding better points and the quasi-Newton iteration is not. When the quasi-Newton method succeeds, it is very unlikely that the new point or the stencil around it will have been sampled before.

The example `history_test.m` in the `Examples/Case_Study_PID` directory of the software collection illustrates the use of the `complete_history` structure to examine the difference between the parallel and serial versions of **imfil.m**.

You may disable the `complete_history` structure, and save some storage, by setting the `complete_history` option to 'off' or 0 with

```
options=imfil_optset('complete_history','off',options);
```

Only do this if you are having serious problems with storage. **imfil.m** can be much less efficient with `complete_history` turned off.

2.3.3 Slow Convergence or No Convergence

When the optimization fails to converge or performs poorly, the `histout` array may indicate the reasons. If, for example, you see that `iarm` $= -1$ for several iterations in a row, that means that the stencil failed on those iterations. That is an indicator that you could terminate the optimization earlier by either changing `scaledepth` (§ 2.6.1) , `target` (§ 2.10.1), `function_delta` (§ 2.10.3), or `stencil_delta` (§ 2.10.2).

If `iarm` = `maxitarm` (see § 2.11.2) for several consecutive iterations, then the line search is failing often but the poll is finding better points on the stencil. This is a signal that the quasi-Newton/Gauss-Newton step is poor, and it may be that your function is not well modeled by a smooth surrogate. In that case, **imfil.m** is reverting to a direct search and you may want to reduce `maxitarm`. If this happens only when the scales become small, then the noise in your function may be large enough to render numerical differentiation ineffective. If you can control the accuracy in f, you should do that and make f scale-aware (§ 2.6.3). Your function may also be poorly scaled, and changing `fscale` (§ 2.5.1) can help.

2.4 Setting Options

You can change **imfil.m**'s algorithmic parameters with the `imfil_optset` command. One way to do this is to begin with a call with no arguments.

```
options=imfil_optset;
```

The output of this call is a MATLAB structure with the default options for **imfil.m**. You need only do this once and then use `imfil_optset` to update the options structure you've created. For example, if you want to change `scaledepth` to 20 and use the SR1 quasi-Newton update, you could call `imfil_optset` three times

```
options=imfil_optset;
options=imfil_optset('quasi','sr1',options);
options=imfil_optset('scaledepth',20,options);
```

prior to the call to **imfil.m**. You can also put all the calls to optset on a single line when you initialize the `options` structure

```
options=imfil_optset('quasi','sr1','scaledepth',20);
```

If you wish to use the `options` structure, you add that as an argument to **imfil.m** when you call it. So your call would look like

```
[x,history]=imfil(x0,f,budget,bounds,options);
```

instead of

```
[x,history]=imfil(x0,f,budget,bounds);
```

Many of the options are toggles, which are either on or off. You may turn a toggle on with any of 1, 'on', or 'yes' and off with any of 0, 'no', or 'off'. For example,

```
options=imfil_optset('least_squares',1);
```

and

```
options=imfil_optset('least_squares','yes');
```

are equivalent. Note that 1 is a numerical value and 'yes' is a string.

2.5 The Inner Iteration

The inner iteration is the optimization loop. **imfil.m** solves general bound constrained optimization problems with a quasi-Newton method and nonlinear least squares problems with the Gauss-Newton iteration. You can replace the built-in methods for the inner iteration with the `executive_function` option (see § 3.5).

2.5.1 Scaling f with `fscale`

If the values of $|f|$ are very small or very large, the quality of the difference gradient which **imfil.m** uses in its search can be poor. **imfil.m** attempts to solve this problem by **scaling** the objective function by dividing it by the size of a "typical value", which we call $imfil\_fscale$. In order to do this for nonlinear least squares we scale the least squares residual by $\sqrt{fscale}$. We will discuss the optimization case here and present the complete details in § 3.2.1.

The default is

$$imfil\_fscale = 1.2|f(x_0)|,$$

which is usually fine.

If $imfil\_fscale$ is too large, the inner iteration within **imfil.m** may terminate too soon, and you may fail to exhaust the information in the current scale. This can lead to poor results, or even complete stagnation (*i.e.* x_0 is never changed).

If $imfil\_fscale$ is too small, the optimization steps may be too large, and the line search may fail. In this case **imfil.m** becomes a form of coordinate search, and the performance will suffer.

You can change this by setting the `fscale` option. Setting `fscale` to a negative value will tell **imfil.m** to use

$$imfil\_fscale = |fscale||f(x_0)|,$$

so $fscale = -1.2$ is the default. If $fscale > 0$ then

$$imfil\_fscale = fscale.$$

$fscale = 0$ is not a sensible value; if you blunder and set $fscale = 0$, **imfil.m** will restore the default. If $f(x_0) = 0$, then **imfil.m** will set $imfil\_fscale$ to 1.

2.5.2 Quasi-Newton Methods for General Problems

For general optimization problems you may set the `quasi` option to 0 (steepest descent, *i.e.* the model Hessian is the identity matrix), 'bfgs' (BFGS) or 'sr1' (SR1). The default is $quasi = `bfgs'$, the BFGS update.

Because **imfil.m** is intended for small problems, **imfil.m** maintains an approximation to the full model Hessian and does not use a sparse or limited-memory [18] formulation of the quasi-Newton methods.

2.5.3 Nonlinear Least Squares

imfil.m will also solve nonlinear least squares problems where the objective function is

$$f(x) = F(x)^T F(x)/2.$$

You tell the code that you have a nonlinear least squares problem by setting `least_squares` option to 1 with the command

```
options=imfil_optset('least_squares',1,options);
```

And write your function so that the **column vector** $F \in R^M$ is returned. **imfil.m** will compute the objective function $F(x)^T F(x)/2$ for you.

The internal nonlinear squares solver in **imfil.m** is a projected damped Gauss-Newton iteration [10, 18, 19].

2.5.4 Which best point to take?

If the current point is x_{base}, the best point in the stencil is x_{min}, and the point selected by the quasi-Newton (or Gauss-Newton) iteration is x_{newt}, **imfil.m** will select x_{newt} to be the new point as long as the line search succeeds, *i.e.*

$$f(x_{newt}) < f(x_{base}).$$

If you prefer to let x_{min} be the new point if

$$f(x_{min}) < f(x_{newt}),$$

set `stencil_wins` to 'yes'. The default is 'no'.

This option is useful both for very rough and very smooth problems. If your optimization landscape has severe discontinuities (as do some of the examples in [13]), then setting `stencil_wins` to 'yes' will help you jump over discontinuities. On the other hand, if the objective function is smooth or very nearly so, setting `stencil_wins` to 'yes' will help avoid local minima when the scale h is large and make no difference if h is very small and the quasi-Newton iteration is working well. That is why the `smooth_problems` option (see § 2.6.4) sets `stencil_wins` to 'yes'. The default is 'no' because **imfil.m** is designed to be a hybrid of search and gradient based methods, and setting `stencil_wins` to 'yes' for all scales can obscure the benefits of the quasi-Newton iteration. The reader can try this for the examples in Chapter 4.

2.5.5 Limiting the Quasi-Newton Step

If the quasi-Newton (or Gauss-Newton) step is too long, the line search may fail repeatedly and you will lose the benefits of the quasi-Newton direction. In that case, the iteration will become coordinate search. You may increase the number of stepsize reductions by changing the `maxitarm` (see § 2.11.2) option from its default of 3, which is a good idea for problems that are very close to smooth problems. Alternatively, the `limit_quasi_newton` option lets you limit the size of the quasi-Newton step before the line search begins. If you set `limit_quasi_newton` to 'yes' the quasi-Newton direction will be no longer than $10h$, where h is the current scale. The default is 'yes', which is a good choice for noisy problems. For nearly smooth problems, 'no' may be better.

2.6 Managing and Using the Scales

2.6.1 Scalestart and Scaledepth

imfil.m samples f on a stencil centered at the current point. The size of that stencil varies at the optimization progresses. The default shape of the stencil is a central difference stencil with $2N$ points. The range of sizes can be controlled by the `scalestart` and `scaledepth` option.

If the directions in the stencil are vectors $\{v_i\}_{i=1}^m$, **imfil.m** will sample f at the points

$$x_c + h(L_i - U_i)v_i$$

for $1 \le i \le m$. The default vectors are the $2N$ unit vectors in the positive and negative coordinate directions. The **scale** h varies as the optimization progresses. The sequence of scale is

$$\{2^{-n}\}_{n=\texttt{scalestart}}^{\texttt{scaledepth}}.$$

`scaledepth` can be changed with the `imfil_optset` command. The defaults are *scalestart* = 1 and *scaledepth* = 7. If you see stagnation in the iteration, reducing `scaledepth` will save some effort, but be aware of the risk of early termination.

2.6.2 `custom_scales`

If you want to use a custom sequence of scales $\{h_n\}_{n=1}^{smax}$ you may do so by setting the `custom_scales` array. This a MATLAB array H with the scales

$$1 > h_1 > h_2 > ... > h_{smax} > 0.$$

$h_1 < 1$ is important because **imfil.m** scales the bounds to 0 and 1, so a choice of $h > 1$ would certainly put all points in the stencil outside of the bound constraints. You can be sure that at least one point (other than the center) is within the bounds by setting $h_1 \leq 1/2$. You set this option with

```
options=imfil_optset('custom_scales',H,options);
```

2.6.3 Scale-Aware Functions

The `scale_aware` option tells **imfil.m** that your function is scale-aware. This means that f can adjust its internal cost or accuracy with knowledge of the scale h. The calling sequence for a scale-aware function is If `scale_aware` is set to 1, **imfil.m** will use the scale as a second input argument to f. Your function should look like

```
[fout,ifail,icount]=f(x,h);
```

See § 4.3 for an example.

2.6.4 Smooth Problems

If you must apply **imfil.m** to a smooth problems, setting `smooth_problem` to `yes` will adjust several parameters, most importantly the scales. The result is a good, but not optimally tuned, finite-difference quasi-Newton (or Gauss-Newton) code. **imfil.m** has been used in this mode to solve suites of artificial test problems [30]. Setting `smooth_problem` to 'yes' is equivalent to this block of MATLAB code:

```
bscales=[.5, .01, .001, .0001, .00001];
options=imfil_optset(...
              'custom_scales',bscales,...
              'stencil_wins','yes',...
              'limit_quasi_newton','no',...
              'armijo_reduction',.25,...
              'maxitarm',5,options);
```

If you use this option, you should probably increase the budget and consider both the default BFGS quasi-Newton method and SR1. Of course, as we said in the introduction, you are better off if you use a code which has been designed for smooth problems.

2.7 Parallel Computing

The `parallel` option tells **imfil.m** that f can be called with multiple arguments, and will return a matrix whose columns are the values of f, $ifail$, and $icount$. So if x is an $N \times P$ array of P arguments to f and `parallel` is set to 1, a call to $f(x)$ will return a $1 \times P$ **row** vector of values, and $P \times 1$ vector of cost estimates, and a $P \times 1$ vector of failure flags. You must return a row vector with P columns for consistency with the nonlinear least squares option.

If you are solving a nonlinear least squares problem, where a call to f returns an $M \times 1$ column vector, your parallel function should return an $M \times P$ array of residual values as well as the $P \times 1$ vectors for $iflag$ and $icount$. The parallel algorithm is not the same as the serial method because all the line search possibilities are examined at the same time.

The default is $parallel = 0$.

The examples in § 1.5.2 and Chapter 4 illustrate the simplest way to make a serial function parallel.

You must keep in mind that this can be much more complicated than simply putting multiple calls to your function inside a parallel for loop (like the MATLAB `parfor` construct). Parallel for loops typically require that the multiple calls to the function do not compete for the same data, and therefore things like global variables inside your function will likely cause the parallel loop to fail.

2.8 Passing Data to f

You can pass data from your calling program directly to f by adding an optional final argument to the call to **imfil.m**. The calling sequence looks like

```
[x,histout,complete_history]=
          imfil(x0,f,budget,bounds,options,extra_data);
```

The MATLAB ode and quadrature codes also let you pass data to a function in this way. You must make the extra argument the final argument to f. For example

```
[fout,ifail,icount]=f(x,extra_data)
```

or, if f is scale-aware,

```
[fout,ifail,icount]=f(x,h,extra_data).
```

Using the optional final argument is a much better idea than communicating with f with global variables. One reason is that global variables can cause problems with parallelism.

In the examples in Chapter 4 the additional argument is a structure which contains parameters for an initial-value problem solver and the data for a nonlinear least squares problem.

2.9 Stencils

imfil.m offers three stencils. You can change from the default centered difference stencil with the `stencil` option. The choices are a one-sided difference stencil, which uses the positive coordinate e_i if $x_c + he_i$ satisfies the bound constraints, and $-e_i$ otherwise, and the **positive basis stencil** [22,23] which uses the $N+1$ points $\{e_i\}_{i=1}^N$ and

$$v_{N+1} = -\frac{1}{\sqrt{N}} \sum_{i=1}^{N} e_i.$$

The `stencil` options are 0 for the default central-difference stencil, 1 for the one-sided stencil (for compatibility with the old FORTRAN code), and 2 for the positive basis stencil.

2.9.1 `vstencil`

You may create your own custom stencil by setting the `vstencil` option to a matrix with your directions in the columns.

To do that, create a matrix VS with your directions in the columns, and then

```
options=imfil_optset('vstencil',VS,options).
```

The example `lc_imfil.m` in the `Examples/Linear_Constraints` directory of the software collection shows how to use the `vstencil` option to avoid stagnation when the default stencil directions are insufficient.

2.9.2 `random_stencil`

You can augment the stencil with k random vectors by setting the `random_stencil` option to k. The theory from [1,11] will apply if $k \geq 1$.

The default is $k = 0$ (no random vectors) because we have seen better performance overall with the basic centered difference stencil. One reason for this is that more vectors will delay stencil failure and cause the iteration to spend too much time in the line search.

If you suspect that the optimal point is on a constraint boundary, especially a hidden constraint boundary, and are seeing stagnation in the iteration, you might use this option and play with various values of k. Adding as few as one random vector will make the algorithm provably convergent [11,19]. This option augments the stencil with k uniformly distributed points on the unit sphere in R^N [26,27]. See § 3.4.1 for an example of this option's overcoming stagnation on a hidden constraint boundary.

2.10 Terminating The Outer Iteration

Most problems can be solved with the default termination criteria for the optimization (or outer) iteration. However, if the iteration is terminating too soon (*i.e.* while it's still making progress) or too late (*i.e.* taking many iterations while making very little progress), there are several things you can do. You may know things that can help **imfil.m** do its job better or may learn things by looking at the iteration history.

The options in this section let you use what you might know about the function to avoid wasted effort (see § 4.4 for an example). Termination is a tricky problem for sampling methods, which is why **imfil.m** offers many, maybe too many, ways to do it.

Two obvious things are changing the list of scales using the `scaledepth` or `custom_scales` options. The `smooth_problems` option, for examples, uses the `custom_scales` option to do part of its job. The options we discuss in this section may help you if working on the scales does not or you have information about your problem that is best communicated to **imfil.m** with these options.

2.10.1 `target`

You may set a `target` value for the optimization. The optimization will terminate once f is below the target. The default value is -10^8 which means that `target` will play no role in the optimization.

2.10.2 `stencil_delta`

If you know how accurate your function is, you may want to terminate once the variation of the function is smaller than your estimate for the error in the function. Setting `stencil_delta` to your estimate of the absolute error in f will terminate the optimization when the maximum absolute difference of function values on the stencil is smaller than `stencil_delta`. To turn this optional termination test on set the `stencil_delta` option to your estimate of the error. The default is -1 which means the option is off.

2.10.3 `function_delta`

Another way to use your estimate of the function's accuracy is to terminate the outer iteration when the change in best function value from one successful quasi-Newton iteration to the next is small. Setting `function_delta` to a non-zero value will terminate the optimization when the change in best values is less than $function\_delta$. This is the approach which helps in the example in § 4.4. The difference between this and the `stencil_delta` options is subtle, but important. If you choose to terminate the iteration when `stencil_delta` is small, you are testing changes in the optimal point. On the other hand, using `function_delta` is testing the quasi-Newton (or Gauss-Newton) iteration by looking at the change in the optimal value. If your problem is nearly smooth, use `function_delta` if the iteration seems to be making very little progress in the terminal phase (*i.e.* stagnating). On the other hand, if your problem is not smooth (or is a coarse discretization of a smooth problem), `stencil_delta` may be a better choice for solving your stagnation problems. Look at the example in § 4.4 (and experiment with the code) to see how to use these options.

2.10.4 `maxfail`

The outer iteration will terminate after `maxfail` consecutive line search or stencil failures. The default is 3. You may want to increase this limit if you think the iteration is terminating too soon. On the other hand, if the iteration is making no progress at all in the terminal phase, you may want to either decrease `maxfail`, reduce the number of scales, or change the sequence of scales.

2.11 Terminating the Inner Iteration

The nonlinear (or inner) iteration will terminate when the norm of the difference gradient is sufficiently small, when maximum iterations have been taken for the entire iteration, or when stencil failure is detected. All of these termination criteria can be changed, but you should take care before messing about with these options.

2.11.1 `maxit`

`maxit` is the upper limit on the number of quasi-Newton (or Gauss-Newton) iterations. The default is 50. You may never have to change this limit. Typically the inner iteration will usually terminate with a stencil failure before 50 quasi-Newton iterations. However, for a nearly smooth problem and a small number of scales, 50 might not be enough.

2.11.2 `maxitarm`

The line search will reduce the step at most `maxitarm` times before returning a failure. The default is 3. The line search is limited in this way for good reason. If your problem is noisy and you don't find something useful after three reductions, you're not likely to do better with more effort. However, if your problem is nearly smooth, you should increase `maxitarm`. The `smooth_problem` option, for example will increase `maxitarm`.

2.11.3 Noise-Aware Functions and the `svarmin` Option

A function is **noise-aware** if it can communicate the size of the noise to **imfil.m**. The function does this via an additional output argument. So, the call looks like

```
[fout,ifail,icount,noise_level]=f(x)
```

where `noise_level` is the function's estimate of the noise.

If the `noise_aware` option is 'on', then **imfil.m** uses the estimate of the noise to tighten the criterion for stencil failure. To do this we evaluate the variation of the objective function on the stencil

$$var = \max_j f(x + hv_j) - \min_j f(x + hv_j)$$

and declare stencil failure when

$$var < noise\_level. \tag{2.1}$$

If f is also **scale-aware** , then you may let the `noise_level` may depend on the scale h. **imfil.m** is prepared for this and queries the `noise_level` before each inner iteration.

If the noise in f does not depend on h, you may set it directly with the `svarmin` option. If $svarmin > 0$ then the inner iteration will declare stencil failure when $var < svarmin$. This is equivalent to setting `noise_aware` to 1 or 'on' and having f return `svarmin` for `noise_level`.

2.11.4 Terminating the quasi-Newton Iteration with `termtol`

The quasi-Newton iteration will terminate when

$$\|\nabla f(x, V, h)\| \leq \tau h,$$

which is intended to mimic the necessary conditions for optimality. The constant in the termination criteria is scaled with a typical value for f. So

$$\tau = \texttt{imfil\_fscale} * \texttt{termtol}.$$

`imfil_fscale` is a "typical value" of f and is set with the `fscale` option (see § 2.5.1), and plays an important role in **imfil.m**'s internal scaling. `termtol`, on the other hand, only affects the termination of the quasi-Newton loop.

2.12 `verbose`

The `verbose` option lets you watch **imfil.m** at work. If you set *verbose* $= 1$, you will see the the first five columns of the rows `histout` array appear on the screen as they are computed. The default is *verbose* $= 0$, which tells **imfil.m** to print only the most serious warnings on the screen.

This is a useful option when troubleshooting, as it is easy to see problems with the line search or stagnation when *verbose* $= 1$, and then stop the optimization in mid-stream to fix the problems.

Chapter 3

Advanced Options

In this chapter we discuss some options to **imfil.m** whitch are powerful enough to do harm. Before using some of these options, you must understand how **imfil.m** manages its scaling and take great care with the calling sequence. Some of these options use **imfil.m**'s internal data structures. You will need to understand how they work to use the advanced options well. This is especially true with the `explore_function` (§ 3.4) and `executive_function` (§ 3.5) options.

3.1 Adding New Directions to the Stencil

You may add new points to the stencil before the computation of the stencil derivative with the `add_new_directions` option. You set this option to the name of the MATLAB function you want **imfil.m** to call before computing the stencil derivative. The `random_stencil` option (see § 2.9.2) is a special case of adding new directions.

If your function is `my_directions.m` and you are updating and existing options structure, you would set the `add_new_directions` option with

```
options=imfil_optset('add_new_directions',@my_directions,options);
```

The calling sequence for your function should be

```
Vnew = my_directions(x, h, V)
```

$Vnew$ is the matrix with the new directions in its columns. In the input, x is the current point, h is the current scale, and V is the current set of directions.

You have to be somewhat careful with this. **imfil.m** will call your function to add directions immediately before computing the stencil derivative, and will use these directions in that computation. One use of this option is to capture tangent directions to explicit constraints. It is not the way to do a global search. Use the `explore_function` option (see § 3.4) to make the search more global. **imfil.m** provides x and V in in your original coordinates (so do not attempt to scale them yourself). When you return your new directions to **imfil.m** they are rescaled and normalized interally.

The example `lc_imfil_driver.m` in the `Examples/Linear_Constraints` directory of the software collection shows how to use the `add_new_directions` option for a linearly constrained problem. The `tangent_directions` functions in the example follows [24] and uses the tangent directions to a linear constraint when it is nearly active, and ignores the linear constraint otherwise. This avoids stagnation when the stencil directions are insufficient and, unlike using `vstencil` does not add more directions when they are not needed.

In the example we seek to minimize

$$\begin{aligned} f(x) &= (1/2-(x)_1)^2+(1-(x)_1)^2(1-x(2))^2/4 \\ &+(1/2-(x)_1)^2(1+(x)_2-2(x)_2^2)/10, \end{aligned} \tag{3.1}$$

subject to the bound constraints

$$0 \le (x)_1 \le 1, 0 \le (x)_2 \le 1,$$

and the linear constraint

$$(x)_1+(x(2)-1) \ge 1. \tag{3.2}$$

Clearly the minimizer is $x^*=(.5,1)^T$. With the default options and an intial iterate of $x_0=(1,0)^T$ on the constraint boundary, the iteration will stagnate with stencil failure at each iteration.

We incorporate the linear constraint into the objective function with the extreme barrier approach. The code for the objective is

```
function [fout,ifail,icount]=lc_obj(x)
% LC_OBJ
%
% Hardwire the linear constraint x_1 + x_2 >= 1 into
% the objective to apply the extreme barrier approach
% to constraints.
%
if x(1)+x(2) < 1
   fout=NaN;
   ifail=1;
   icount=0;
else
   fout1=(x(1)-.5)^2;
   fout2=.25*(1-x(1))^2*(1 - x(2))^2;
   fout3= .1*(x(1)-.5)^2*(1 + x(2) - 2* x(2)^2);
   fout=fout1+fout2+fout3;
   ifail=0;
   icount=1;
end
```

The terms in the function are designed to put the optimal point at x^* and force stagnation if one uses the standard basis in the positive and negative coordinate directions.

The file `tangent_directions.m` adds the tangent directions to the linear constraint if any point in the stencil violates the constraints.

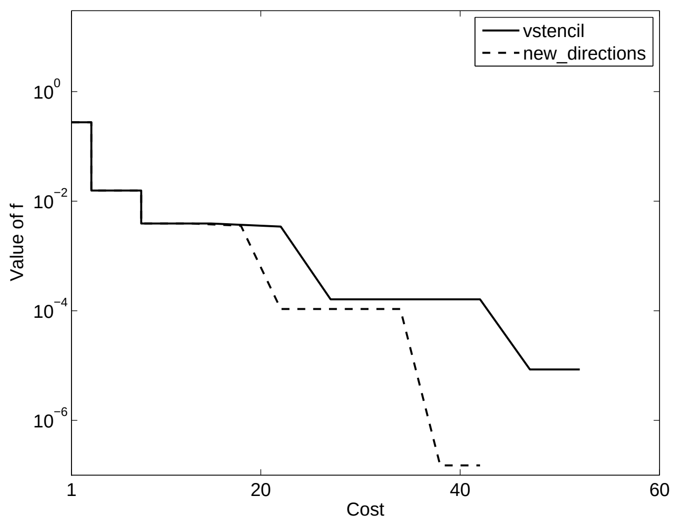
```
function vnew=tangent_directions(x,h,v)
% TANGENT_DIRECTIONS
%
% This is an example of a way to add new directions.
%
% If any point in the stencil does not satisfy the linear constraints,
% I will add tangent directions to the stencil.
%
% The linear constraints, which we handle with the extreme barrier
% method, are x(1) + x(2)  >= 1
%
vnew=[];
[mv,nv]=size(v);
yesno=1;
for i=1:nv
    x_trial=x + h * v(:,i);
    yesno=yesno*test_constraint(x_trial);
end
if yesno==0
   vnew=zeros(2,2);
%
% The linear constraints are (1, 1)^T x >= 1.
% So a tangent vector is  (-1, 1)^T. We do not have to normalize
% this vector because imfil_core will do that.
%
   vnew(:,1)=[-1,1]';
   vnew(:,2)=-vnew(:,1);
end
yesno


function yesno=test_constraint(x)
%
% No deep thinking here. Either the constraint is violated (yesno = 0) or
% it's not (yesno = 1).
%
val = x(1) + x(2) ;
yesno = (val >= 1);
```

The code `ld_driver.m` compares the use of the `add_new_directions` option to the `vstencil` option. To use `vstencil` you would set the options by

```
VS=[0 1; 0 -1; 1 0; -1 0; -1 1; 1 -1]';
options=imfil_optset('vstencil',VS);
```

Figure 3.1. *Iteration History with Extra Directions*



and you would use `add_new_directions` in this way

```
options=imfil_optset('add_new_directions','tangent_directions');
```

Both options will work, but `add_new_directions` is more efficient, as you can see in Figure 3.1. The reason for this is that using the `vstencil` option forces evaluation of f in the extra directions for every iteration, whereas the `add_new_directions` option only uses the extra direction if a point on the stencil violates the constraints.

3.2 The `iteration_data` Structure

imfil.m keeps track of the status of the iteration with the internal `iteration_data` structure. This structure is used heavily within the core of **imfil.m**. The options in § 3.4 and § 3.5 also may need to read some of the fields in that structure. We will explain how these components of `iteration_data` are used in the following sections. For now, we tabulate in Table 3.1 those fields of the structure which are useful for the `executive_function` and `explore_function` options.

Most of the items in Table 3.1 should be clear to a reader who has made it this far in the book. We explain `f_internal` completely in § 3.2.1.

3.2.1 Internal Scaling and `f_internal`

imfil.m scales both the variables and the function value. You will need to understand that scaling to use the `executive_function` and `explore_function` options. This is especially the case if you decide to revert to your original scaling in these functions or rescale the `complete_history` to examine the iteration in your original coordinates.

Table 3.1. *Components of the* `iteration_data` *structure*

`iteration_data.f_internal`	Function handle to $f_{internal}$
`iteration_data.core_data`	Structure passed to $f_{internal}$
`iteration_data.complete_history`	Complete history structure
`iteration_data.xb`	xb is the best point found so far in the optimization
`iteration_data.funsb`	$f_{internal}(xb)$
`iteration_data.fobjb`	Objective function value at xb
`iteration_data.h`	Current scale
`iteration_data.itc`	the inner iteration counter
`options`	the `options` structure

Suppose your function is f and your feasible set is

$$\Omega = \{x \in R^N \mid L_i \leq (x)_i \leq U_i\}.$$

imfil.m begins by transforming Ω to

$$\Omega_{internal} = \{z \in R^N \mid 0 \leq (z)_i \leq 1\}.$$

The transformation is

$$x = Dz + L,$$

where D is the diagonal matrix with entries

$$D_{ii} = (U_i - L_i).$$

imfil.m also scales f by multiplication by $imfil\_fscale$ (see § 2.5.1). The inner iteration and the search within **imfil.m** operate on the internal function `f_internal`. The scaling is

$$f_{internal}(z) = f(Dz + L)/imfil\_fscale$$

for optimization problems and

$$f_{internal}(z) = F(Dz + L)/\sqrt{imfil\_fscale}$$

for nonlinear least squares.

MATLAB functions you write for the `executive_function` and `explore_function` options interrupt **imfil.m** in mid-stream, and hence will see the internal function and the internal variables. **imfil.m** rescales the `history` array and the `complete_history` structure to your original variables after the optimization is complete. If your functions want to see, for example, the `complete_history` structure in the original coordinates or explore design space in the original coordinates, you will have to do that rescaling yourself. This can get messy and we do not recommend that.

imfil.m informs $f_{internal}$ about the bounds with the `core_data` structure, which is an extra argument to $f_{internal}$. **imfil.m** also treats $f_{internal}$ as if were both scale-aware and noise-aware. The `options` structure is part of the `core_data` structure, so the call to $f_{internal}$ will send the correct arguments to f and do the proper things with the other options (such as the `parallel` option). Your call to $f_{internal}$ should look like

```
[fx,iff,icf,tol]=f_internal(x,h,core_data).
```

You should never have to work with `core_data` directly, only be prepared to pass it to `f_internal`. `core_data` is a substructure of the `iteration_data` structure, which you get by

```
core_data = iteration_data.core_data;
```

If your original function f is neither scale-aware or noise-aware, you may set $h = 1$ with no harm and ignore the output argument *tol*.

3.3 Updating the `complete_history` Structure

The final two advanced options will update the `complete_history` structure and may want to read it as well. You access this structure as a substructure of the `iteration_data` structure.

```
complete_history = iteration_data.complete_history
```

You'll need to loook at § 2.3.2 if you want to read the data in the structure.

If you wish to write to the structure, you must pass a history structure back to **imfil.m**. The sections on the `executive_function` and `explore_function` options explain where your structure must appear in the list of output arguments. In this section we explain how you must build that structure.

The `complete_history` structure uses the scaled coordinates, so it is based on evaluations of `f_internal`. The function values in the structure are scalars for optimization problems and vectors in R^M for nonlinear least squares problems,

We provide a tool `build_history` in the `Imfil_Tools` directory which you should use for this purpose. As you evaluate the `f_internal` you should accumulate the history of the evaluations in three arrays

- `xarray`, a matrix with N rows whose columns are the evaluation points,
- `funmat`, a matrix with M rows whose columns are the evaluations of `f_internal` at the columns of `xvec`,
- `failvec`, a vector of zeros and ones with `failvec(i) = 0` if the evaluation of `f_internal` succeeded and `=1` if the evaluation failed.

Once you have assembled these three arrays, the history of your evaluations can be recorded with

```
my_history = build_history(xarray, funmat, failvec);
```

Your function would send `my_history` back to **imfil.m** as an output argument. **imfil.m** will update the `complete_history` structure, which you cannot update yourself.

For example, if the `parallel` option is off and you wish to evaluate `f_internal` at the P points in `xarray`, you would build the other arrays by

```
farray=[];
failvec=[];
for i=1:P
    [fout, ifail, icount, tol] = ...
           feval(f_internal, xarray(:,i), h, core_data);
    farray=[farray, fout];
    failvec = [failvec, ifail];
end
```

If the `parallel` option is on, this is much easier

```
[farray, failvec, icount, tol] ...
          = feval(f_internal, xarray, h, core_data);
```

Before you build your function's history array, you should determine if the `complete_history` option is off (rare, but possible). To do this query the `options`, which is substructure of `iteration_history`.

```
imfil_complete_history = iteration_history.options.complete_history;
if imfil_complete_history == 1
   my_history = build_history(xarray, funmat, failvec);
else
   my_history=[];
end
```

If you want to build your `my_history` array on your own, you must carefully examine the `build_history.m` file and make sure you construct your history structure correctly. We will invoke a classical warning from [21]: "The world will end if you get this wrong." Therefore, we recommend that you accumulate the function evaluation data and build your `my_history` structure as the last step in your function with the `build_history` function in the `Imfil_Tools` directory.

In the sections that follow we show how to use `build_history` in the context of complete functions, so you can see the context and how to use the `iteration_data` structure.

3.4 Testing More Points with the Explore_Function Option

After the inner iteration at a given scale you may explore more globally with the `explore_function` option. You must write a function to select new points in the scaled feasible set

$$0 \leq (x)_i \leq 1,$$

evaluate f at these points, and then record the results so that **imfil.m** may update `complete_history` structure. This function is called after the inner iteration and lets you replace the best point from the inner iteration with the results of your exploration or the current best point, whichever is better. You may also want to examine the `complete_history` structure to guide your selection of new points. You may read (but not write to) `complete_history` which is the

```
iteration_data.complete_history
```

field of the `iteration_data` structure.

The complete history structure is not simple and you must do the update with care, especially if you rescale back to your original coordinates (which is a very bad idea). The `complete_history` is in the scaled coordinates, so the points all have coordinates in $[0, 1]$. The values of the function are also scaled by $imfil\_fscale$ (see S 2.5.1) Right before **imfil.m** returns, **imfil.m** rescales the vectors and function values in the `complete_history` structure into the original coordinates, but if you access it before the optimization is complete, you must use the scaled coordinates.

If, for example, your function is `my_search` you set the option with

```
options=imfil_optset('explore_function',@my_search,options);
```

Your function call must look like

```
[xs, fs, my_cost, explore_history] = ...
       my_search(f_internal,iteration_data,my_search_data);
```

The inputs are `f_internal` which is **imfil.m**'s internal function, the `iteration_data` structure, and an (optional) final argument `my_search_data` for any data you wish to pass to your explore function. You tell **imfil.m** about `my_search_data` by setting the `explore_data` option:

```
options=imfil_optset('explore_data',my_search_data,options);
```

In the output x_s is the best point from your search, $f_s = f(x_s)$, and my_cost is the number of function evaluations your exploration needed. Remember that if you are solving a nonlinear least squares problem, $f_s = F(x_s)$ will be a vector in R^M.

If you wish to turn the `explore_function` option off after you have used it, you may reset the options structure or turn it off explicitly with

```
options=imfil_optset('explore','off');
```

3.4.1 Random Search Example

We return to the example from § 3.1. Our exploration function simply evaluates f at random points in Ω. The number of random points is an extra argument to the function. The MATLAB code for this explore function is in `Imfil_Tools` directory. The function is more general than we need for this example. In particular, it will do the right things for nonlinear least squares problems and parallel evaluation.

We list the entire function here in order to show you how to get the data you'll need from the `iteration_data` structure, do the main work of the function, and then record the results. The first several lines of the function are devoted to harvesting data. The function evaluation tests for parallelism by querying the `parallel` option and will use parallel evaluation if the `parallel` is on. The next block of code determines if you've found a new best point. Finally, the code builds the `explore_history` structure. You will see this pattern again when we discuss the `executive_function` option in § 3.5.

```
function [xs, fs, my_cost, explore_history] = ...
       random_search(f_internal,iteration_data,my_search_data);
% RANDOM_SEARCH
% function [xs, fs, my_cost, explore_history] = ...
%      random_search(f_internal,iteration_data,my_search_data);
%
% This is an example of an explore_function.
% This function samples f at a few random points and returns the best
% thing it found. I store the number of random points in the
% my_search_data structure.
%
options=iteration_data.options;
parallel = options.parallel;
imfil_complete_history=options.complete_history;
npoints=my_search_data;
xarray=rand(2,npoints);
farray=[];
my_cost=0;
%
% Extract what you need from the structures.
%
% Pass h and core_data to f_internal
%
h=iteration_data.h;
core_data=iteration_data.core_data;
%
% What's the current best point and best objective function value?
%
xb=iteration_data.xb;
funsb=iteration_data.funsb;
fvalb=iteration_data.fobjb;
%
% Am I solving a least squares problem?
%
least_squares=iteration_data.options.least_squares;
%
% Sample the points. Keep the books for the build_history function.
%
failvec=zeros(1,npoints);
```

```
funmat=[];
if parallel == 0
   for i=1:npoints
      x=xarray(:,i);
      [funmati,failvec(i),icount,tol] = ...
                  feval(f_internal,x,h,core_data);
      funmat=[funmat,funmati];
      my_cost=my_cost+icount;
   end
else
   [funmat,fail,icount]=feval(f_internal,xarray,h,core_data);
   my_cost = my_cost+sum(icount);
end
%
% Do the right thing for least squares problems.
%
for i=1:npoints
    if least_squares == 1
        fval=funmat(:,i)'*funmat(:,i)/2;
    else
        fval=funmat(i);
    end
    farray=[farray',fval]';
end

%
% Now see if you've made any progress. If not, return the
% the current best point.
%
[ft,imin]=min(farray);
if failvec(imin) == 0
   xs=xarray(:,imin);
   fs=funmat(:,imin);
else
   fs=funsb;
   xs=xb;
end
%
% Finally, build the explore_data structure.
%
if imfil_complete_history == 1
   explore_history = build_history(xarray, funmat, failvec);
else
   explore_history=[];
end
```

Note that, following the recommendation in § 3.3, `explore_function` checks to see if the `complete_history` option is set to the default of `yes`.

The MATLAB code for the example is

```
Examples/Linear_Constraints/explore_driver.m.
```

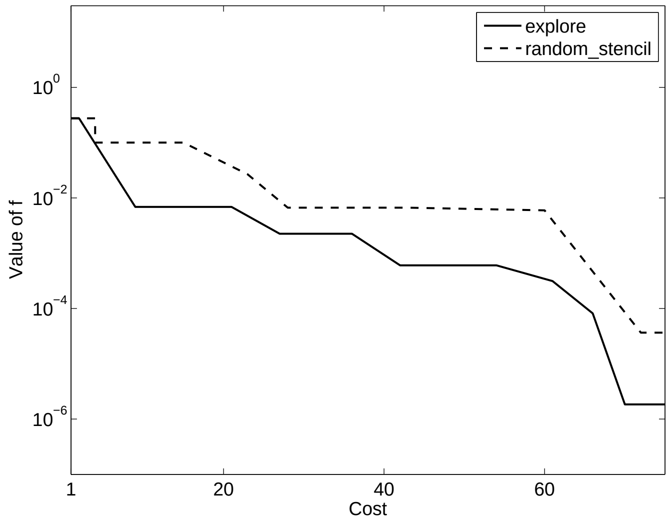
In the example we set

```
npoints=10;
options=imfil_optset('explore_function',@my_search,...
                     'explore_data',npoints);
```

which tells the search function to look for the number of points in its last input argument.

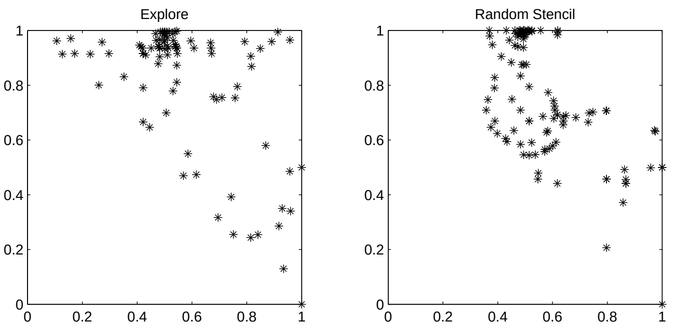
Figure 3.2 shows the iteration history with 10 random directions and compares it with one with `random_stencil` (see § 2.9.2) set to 10. The exploration is faster than the `random_stencil` option with a similar number of function evaluations, but neither random option is a good as the deterministic methods `vstencil` (§ 2.9.1) or `add_new_directions` (§ 3.1) which we compared in Figure 3.1. This is no surprise. The deterministic methods use knowledge of the structure of the problem, and the randomized methods just guess.

Figure 3.2. *Iteration History with Random Exploration*



Finally, in Figure 3.3 we present scatter plots of the successful points from the `complete_history` structures. As you can see the random exploration nicely captures the constraint boundary.

Figure 3.3. *Complete History with Random Exploration*



3.5 The Executive Function

If you don't like the gradient-based inner iterations in **imfil.m**, you may replace them with one you like better. The quasi-Newton and Gauss-Newton solvers built into **imfil.m** use the same interface as the one we describe here. These functions take **a single iteration** when called and then return control to **imfil.m**. Your replacement must do that as well.

To do this you use the `executive_function` option to pass **imfil.m** a handle to your solver and, if needed, the `executive_data` option for any data, such as a quasi-Newton model Hessian or a Levenberg-Marquardt parameter, you solver will update as the iteration progresses. You initialize the data when you set the option. The call to `imfil_optset` will look like

```
options=imfil_optset('executive_function',@my_solver,...
               'executive_data',my_data);
```

where `my_solver` is your solver and `my_data` is your data. `my_data` may be a matrix, a function handle, or any structure you need.

Suppose, for example, your solver is `my_solver.m` and you maintain a quasi-Newton Hessian. If you wish to initialize the Hessian to the identity matrix, you would set up the executive like this:

```
Hess = eye(n,n);
options=imfil_optset('executive_function',@my_solver, ...
                     'executive_data',Hess);
```

The data you set with the `executive_data` option may be any matrix or structure you need.

In § 3.5.3 and 4.4.2 we show how to incorporate the Levenberg-Marquardt method into **imfil.m**. The solver `lev_mar_exec.m` is in the `Imfil_Tools` subdirectory. In the example we initialize the Levenberg-Marquardt parameter to 1 with the call

```
options=imfil_optset('least_squares',1,...
   'executive_function',@lev_mar_exec,'executive_data',1.0);
```

Similarly to the `explore_function` option, your executive function must manage a history structure and conform to a rigid calling sequence. The calling sequence gives you enough information to update a quasi-Newton model Hessian, and you must include that information in the input even if you don't plan to use it. The calling sequence is

```
function [xp, fvalp, funp, fcost, iarm, solver_hist, nfail, new_data] ...
         = my_solver(f, x, fun, sdiff, xc, gc, iteration_data, old_data)
```

3.5.1 Input to the Executive Function

The input string must be somewhat long to accommodate the differing needs of quasi-Newton methods for optimization, which need some history of the iteration, and other methods, such as Gauss-Newton, which do not. We describe the input arguments in the list below.

- f is a handle to the objective function (optimization) or the least squares residual (nonlinear least squares). **imfil.m** will pass `f_internal` to you with this argument.
- x is the most recent iteration.
- $fun = f(x)$.
- $sdiff$ is the simplex derivative of x. This is the gradient for optimization problems and the Jacobian for nonlinear least squares.
- xc is the previous point at which a simplex derivative was computed.
- gc is the gradient of the objective function at xc.
- `iteration_data` is the iteration data structure (see § 3.2).
- `old_data` is the data your executive function will update.

The input list should be self-explanatory with the exception of the two points and two derivatives, which are for quasi-Newton methods. If your data is a quasi-Newton model Hessian, you will update that model Hessian using x, xc, gc, and $sdiff$ before computing the new point. This is the one time it might help to examine the source of **imfil.m** and, in particular the function `imfil_qn_update`.

3.5.2 Output from the Executive Function

The output arguments are used by **imfil.m** to update its internal history structures and manage the iteration. The next (and final) list describes the output arguments.

- xp is the new point, which may be the same as x if the iteration fails.
- $fvalp$ is the objective function value at xp. $fvalp = funp$ for optimization problems, but not for nonlinear least squares.

- $funp = f(xp)$.
- $fcost$ is the total cost of the function evaluations done in your function.
- $iarm$ is the counter of step-size reductions, Levenberg-Marquardt parameter increments, or other global convergences changes. You may elect to declare a failure when $iarm > maxitarm$.
- `solver_hist` is the update to the `complete_history` structure. You build it the same way as you would the `explore_history` structure. See § 3.3 for the rules.
- $nfail$ is the failure flag. $nfail = 0$ if the iteration succeeds. Otherwise $nfail = 1$.
- `new_data` is your function's update of `old_data`. For example in the quasi-Newton code, `new_data` is the update of the model Hessian. The Gauss-Newton solver does not update the data at all.

An executive function builds its history structure in the same way an explore function does and we refer the reader to § 3.3 and § 3.4.1 for the details.

If you wish to turn the `executive_function` option off after you have used it, you may reset the options structure or turn it off explicitly with

```
options=imfil_optset('executive','off');
```

3.5.3 Levenberg-Marquardt Example

We have put an example the `Imfil_Tools` directory. The code `lev_mar_exec.m` is am implementation of the Levenberg-Marquardt algorithm. Our implementation is standard [10,18,19]. We apply `lev_mar_exec.m` to an example in § 4.4.2, and will only discuss a few details of the implementation here.

The function definition statement is

```
function [xp, fvalp, funp, fcost, iarm, levmar_hist, nfail, nunew] ...
         = lev_mar_exec(f, x, fun, jac, xc,  gc, ...
                iteration_data, nuold)
```

Here we have followed the directions, while naming the history structure and the Levenberg-Marquardt parameter in an appropriate way. This is a least squares computation, so $sdiff$ is a stencil Jacobian and we have named the variable accordingly. You should study this code before writing an executive function on your own.

Our implementation is serial. A parallel implementation (which is an exercise for you) would test several candidate Levenberg-Marquardt parameters at once time by evaluating the functions, computing $ared/pred$, and then choosing one parameter or rejecting them all.

One thing any executive function should so is check for input errors. The example function `lev_mar_exec.m` makes sure that the `least_squares` option is on, for instance.

Chapter 4

Parameter Identification Example

The example in this chapter is a **parameter identification** (PID) problem from [2, 18]. In this example $N = 2$. The goal is to identify the damping constant c and spring constant k of a linear spring by minimizing the difference between a numerical prediction and measured data. The experimental scenario is that the spring-mass system will be set into motion by an initial displacement from equilibrium and measurements of displacements will be taken at equally spaced increments in time.

In this chapter we go into considerable detail about both the application and the MATLAB programming.

The MATLAB codes for this chapter are in the `Examples/Case_Study_PID` subdirectory of the software collection.

4.1 Problem Formulation

We consider an unforced harmonic oscillator where the displacement u is the solution of the initial value problem

$$u'' + cu' + ku = 0; u(0) = u_0, u'(0) = 0, \tag{4.1}$$

on the interval $[0, 10]$. In (4.1) $u' = du/dt$ and $u'' = d^2u/dt^2$.

We let $x = (c, k)^T$ be the vector of unknown parameters and, when the dependence on the parameters needs to be explicit, we will write $u(t : x)$ instead of $u(t)$ for the solution of (4.1). If the displacement is sampled at $\{t_i\}_{i=1}^M$, where $t_i = (i-1)T/(M-1)$, and the observations for u are $\{u_i\}_{i=1}^M$, then the objective function is

$$f(x) = \frac{1}{2}\sum_{i=1}^{M} |u(t_i : x) - u_i|^2. \tag{4.2}$$

We will use MATLAB's `ode15s` [31] to solve (4.1), and use the solution from `ode15s` to compute F. The first step in using `ode15s` is to convert (4.1) to a first order system for

$$y = \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} u \\ u' \end{pmatrix}.$$

The resulting first order system is

$$y' = \begin{pmatrix} v \\ -cv - ku \end{pmatrix}, \tag{4.3}$$

with initial data $y(0) = (u_0, 0)^T$.

Here is a MATLAB code for the right side of the differential equation. Note that the parameters c and k are passed to the right side of the differential equation as a third argument. The ode solvers within MATLAB let you pass the parameters to the function in this way by means of an optional final argument in the call to the solvers.

```
function yp=yfunsp(t,y,pid_parms)
%
% simple harmonic oscillator for parameter id example
%
% first-order system form of y'' + c y' + k y = 0
%
yp=zeros(2,1);
yp(1)=y(2);
c=pid_parms(1);
k=pid_parms(2);
yp(2)= - k* y(1) - c*y(2);
```

One might think that it is easier to pass the parameters as MATLAB global variables. However, global variables can cause problems with parallel computing. So, don't do that.

We configure the problem so that the solution is $x^* = (c, k)^T = (1, 1)^T$. For these values of the parameter the solution for $u_0 = 10$ is

$$u = e^{-t/2}\left(10\cos(\sqrt{3}t) + (5/\sqrt{3})\sin(\sqrt{3}t)\right).$$

We will begin by letting the data $u_i = u(t_i : x^*)$, where $t_i = (i-1)/100$ for $1 \le i \le 101$. So, we compare the output of `ode15s` with the exact solution. The MATLAB codes let you vary the initial data, and the code `yfunex.m` will compute the exact solution for any initial data, c, and k.

The least-squares formulation is the most natural. To compute the residual we must solve the initial value problem (4.3) with `ode15s` and then compare the results with the data. In this example we assume that `yfunsp.m` is a file in the MATLAB path. Our main program `driver_pid.m` builds a data structure `pid_info` with several parts:

- `pid_data`: a column vector of size 101 with the data $\{u_i\}_{i=1}^{101}$,
- `time_pts`: the points in time where `ode15s` reports the solution,
- `pid_y0`: the column vector with the initial data, $(10, 0)^T$, and

- pid_tol: a scalar for the tolerances for ode15s. We set both the relative and absolute tolerances to 10^{-3}.

This structure will not change throughout the optimization. The driver builds the structure with the lines:

```
%
% pid_parms contains the zero-residual solution to the noise-free problem
% pid_tol is the tolerance given to ode15s
%
m=100; t0=0; tf=10;
%
% Construct the data for the integration.
% pid_data is a sampling of the "true" solution.
%
pid_parms=[1,1]'; pid_y0=[10,0]'; pid_tol=1.d-3;
time_pts=(0:m)'*(tf-t0)/m+t0;
%
% Find the analytic solution.
%
pid_data=exact_solution(time_pts,pid_y0,pid_parms);
%
% Pack the data into a structure to pass to serial_pidlsq
%
pid_info=struct('pid_y0',pid_y0,'pid_tol',pid_tol,...
                'time_pts',time_pts,'pid_data',pid_data);
```

and then runs **imfil.m** four times to make Figure 4.1. The call to **imfil.m** uses the optional extra argument to send the pid_data structure to the function.

```
[x,histout]=imfil(x0,@serial_pidlsq,budget,bounds,options,pid_info);
```

The tolerance for the initial value problem solver is coarse, and we should not expect to be able to reduce the norm of the nonlinear residual by much more than a factor of 10^3, even in the zero-residual case.

The serial code serial_pidlsq computes the residual using the pid_info structure. This structure is passed to **imfil.m** as an optional final argument, and **imfil.m** sends it directly to serial_pidlsq (see § 2.8). The input for serial_pidlsq is is the vector of parameters $x = (c, k)^T$ and the pid_info structure. serial_pidlsq must then pass x to yfunsp.m, which it will do with the same optional final argument approach.

```
function [f,ifail,icount]=serial_pidlsq(x,pid_info)
%
% Parameter ID example formulated as nonlinear least squares problem.
%
% Unpack the pid_info structure and get organized.
%
```

```
pid_data=pid_info.pid_data;
tol=pid_info.pid_tol;
time_pts=pid_info.time_pts;
y0=pid_info.pid_y0;
%
% Call the integrator only if x is physically reasonable, ie if
% x(1) and x(2) are nonnegative. Otherwise, report a failure.
%
ifail=0; icount=1;
if min(x) < 0
   ifail=1; icount=0; f=NaN;
else
   options=odeset('RelTol',tol,'AbsTol',tol,'Jconstant',1);
   [t,y]=ode15s(@yfunsp, time_pts, y0, options, x);
   f=y(:,1)-pid_data(:,1);
end
```

The calling sequence follows the format in § 2.2.2. Note that there is a failure mode. If either c or k is negative, then the spring is not physical and the solution is exponentially increasing. The code traps this and returns without calling the integrator. You could fix this yourself, as we did in the driver program, by making sure that the lower bounds you give to **imfil.m** are all nonnegative.

An integrator like `ode15s` asks you to provide a local truncation error tolerance via the `odeset` command. This tolerance controls the accuracy of the integration, and thereby the resolution in f. We therefore have an opportunity to experiment with the `scale_aware` option by letting the accuracy of the integrator depend on the scale. We will do that in § 4.3. For the present we will fix the tolerance to the scalar `tol`, which `serial_pidlsq` harvests from the `pidinfo` structure as `pidinfo.pid_tol`. We have also used `odeset` to set the option `Jconstant` in `ode15s` to 1, indicating that the differential equation is linear. Finally, we put x in as an optional final argument, which is then passed to `yfunsp` as its third argument.

If you plan to use the MATLAB initial value prolblem solvers in your work, study the help files. Typing `help odeset` and `help ode15s` at the MATLAB prompt will make this section easier to follow.

4.1.1 Calling imfil.m and Looking at Results

We now show how the simplest call would work. The plots in the upper row of Figure 4.1 reflect the a case with an intentionally poor choice of bounds

$$L = \begin{pmatrix} 2 \\ 0 \end{pmatrix} \text{ and } U = \begin{pmatrix} 20 \\ 5 \end{pmatrix},$$

which exclude the solution. We gave the optimization a budget of 100 calls to the integrator and an artificially low upper limit of five scales $\{2^{-n}\}_{n=1}^{5}$. We changed the set of scales from the default set $\{2^{-n}\}_{n=1}^{7}$ by using the `imfil_optset` command to change `scaledepth`. The MATLAB commands which follow the construction of the `pid_info` structure are

```
bounds=[2 20; 0 5];
x0=[5,5]'; budget= 100;
options=imfil_optset('scaledepth',5,'least_squares',1);
```

Note that the `least_squares` option is on in this example. Having set the options, the call to **imfil.m** looks like

```
[x,histout]=imfil(x0,@serial_pidlsq,budget,bounds,options,pid_info);
```

Note that the structure `pid_info` is the final argument and is sent directly to `serial_pidlsq`.

As you can see from the plot on the upper left of Figure 4.1, the iteration terminated before the budget had been exhausted. We can return to the default set of scales by reinitializing the `options` structure, but making sure that `least_squares` is still on,

```
options=imfil_optset('least_squares',1);
```

and calling `imfil.m` again. The picture on the upper right reflects the results of this change. Now the optimization requires less than the entire budget, the final value of the objective function is lower (but not by much), and the value of f seems to have stabilized. However, the graph of f also has a flat region earlier in the iteration, but the iteration had not converged at that point. The upper two images in Figure 4.1 illustrate the difficulty in terminating the iteration.

The `histout` array records the progress of the optimization. All the plots were made with the first two columns of the `histout` array. The plots at the top of Figure 4.1 were made with the command

```
plot(histout(:,1),histout(:,2),'-');
```

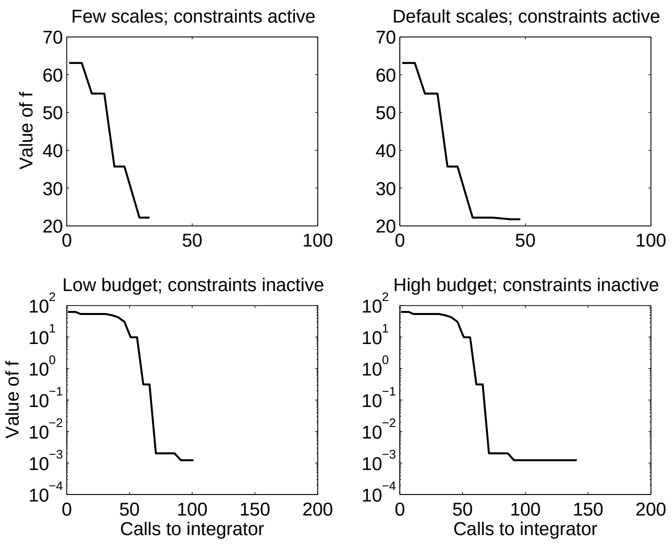
The first two columns of the `histout` array are the function values and the cumulative cost, measured in this case by calls to `ode15s`. When we look at the plots we see that the optimization has made very little progress after 75 or so calls to `ode15s`. You may modify the example code to add more scales and increase the budget, but the value of the function will decrease only a little, if at all. The reason for this is that we have resolved the optimal point as far as the resolution in the integrator will allow.

The plots on the bottom of Figure 4.1 are from an optimization where the global minimum is within the bounds. Here we set

```
bounds=[0 20; 0 5];
```

The lower left plot in Figure 4.1 shows the progress of the optimization with a budget of 100 and the default set of scales. In this case the budget and the number of scales are sufficient to fully resolve the optimal value. The lower right plot shows the results with a budget of 200 and 20 scales. The results are not very different, and

Figure 4.1. *Iteration History: Parameter ID*



the iteration seems to spend over half the time at the same place. This returns us to the issue of termination. How do we know when to stop the optimization? How can we tell if the budget is too small? Should we change the set of scales? These are open research questions at this time (2011). We will return to the termination issue in this chapter in § 4.4.

4.2 Parallelism

The MATLAB Parallel Toolbox makes parallel computing in MATLAB very accessible, but **NOT EASY**. Even the basic `parfor` loop requires attention to data dependencies and data types. The only way to master the MATLAB tools, or any other parallel environment, is to play with the software, make mistakes, and try to understand the (sometimes opaque) error messages.

4.2.1 Parallelizing the Serial Code

As an example of the use of the `parallel` option, we report on results obtained with the `parfor` loop from the MATLAB Parallel Toolbox. If you don't have that toolbox, you may have to replace the `parfor` loop in `parallel_pidlsq.m` with a `for` loop if you have an older version of MATLAB, and thereby mimic the true parallel version in the sense that you will get the same results as the parallel algorithm **for this example**. However, as a general rule you cannot duplicate the parallel results with this technique.

```
function [fa,ifaila,icounta]=parallel_pidlsq(xa,pid_info)
% PARALLEL_PIDLSQ uses parfor to parallelize the serial code.
```

```
%
% The code will accept multiple input vectors
% and return a matrix of outputs.
%
[nr,nc]=size(xa);
fa=[];
ifaila=zeros(nc,1);
fcounta=zeros(nc,1);
parfor i=1:nc
    [fap,ifaila(i),icounta(i)]=serial_pidlsq(xa(:,i),pid_info);
    fa=[fa, fap];
end
```

Now one needs to make only a few changes to `driver_pid`. Turn the `parallel` option on and call `parallel_pidlsq.m`. The new lines are

```
options=imfil_optset('parallel',1,options);
[x,histout]=imfil(x0,@parallel_pidlsq,budget,bounds,options);
```

Of course, before using the parallel toolbox, you must create a matlabpool. Here's an example of how one does that

```
>> matlabpool(8)
Starting matlabpool using the 'local' configuration ...
             connected to 8 labs.
>>
```

In this example, a new matlabpool with eight cores is ready for your parallel job. If you invoke the `matlabpool` command and already have a pool in place, MATLAB will close the existing pool and build a new one.

4.2.2 Looking at the Parallel Results

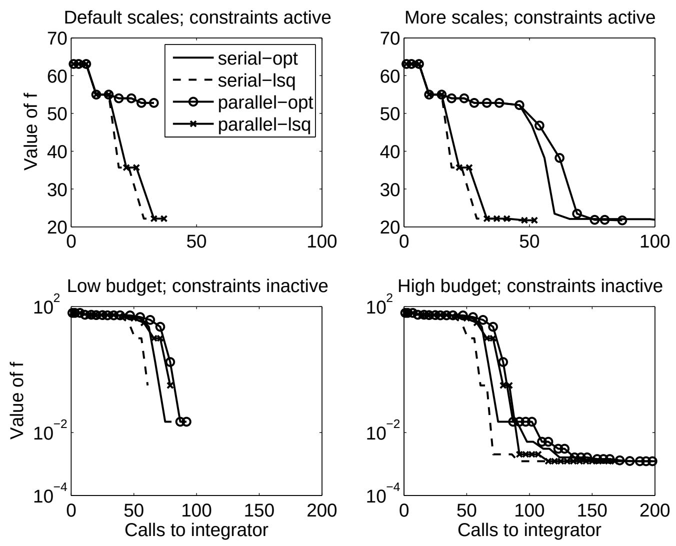
We will now compare the parallel and serial algorithms. We can do this using the parallel function `parallel_pidlsq` even if the `parallel` option is off. The reason is that if `parallel` is set to 0, the function is evaluated in serial mode. We also compare the least squares formulation to the alternative formulation of using the quasi-Newton optimization algorithm to minimize

$$f(x) = \|F(x)\|^2/2.$$

The MATLAB code for this is `pidobj.m`.

```
function [fa,ifaila,icounta]=pidobj(xa,pid_info)
% PIDOBJ calls PARALLEL_PIDLSQ to build an objective
% function that does not use the least squares structure.
%
[nr,nc]=size(xa);
```

Figure 4.2. *Optimization History: Parameter ID Revisited*



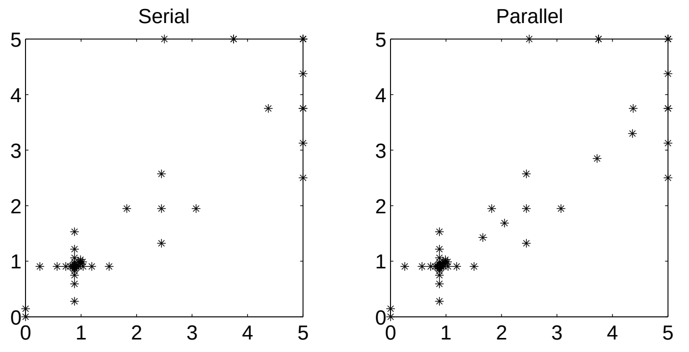
```
fa=zeros(1,nc);
[fl,ifaila,icounta]=parallel_pidlsq(xa,pid_info);
for i=1:nc
    fa(i)=fl(:,i)'*fl(:,i)/2;
end
```

The call to **imfil.m** would be exactly the same as for the least squares formulation, except you would not turn on the `least_squares` option. Ignoring the least squares structure is a bad idea, as you can see from the plots in Figure 4.2.

We now revisit Figure 4.1 by comparing the serial least squares results in that figure with a serial optimization computation and parallel results. Clearly the least squares formulation is better because of the rapid convergence of the Gauss-Newton iteration for this small-residual problem. As one can see from the lower right plot in Figure 4.2, the number of function evaluations in serial and parallel iteration histories can differ by over 30%, in favor of the serial algorithm, which is not surprising since $N = 2$ and the parallel line search queries three or more possibilities at once. This is an example of the difference between the parallel and serial algorithms. The plots for the active constraint cases show that the parallel and serial algorithms need roughly the same number of iterations. However, the parallel method will, in general, take less time, especially if calls to the function are expensive. The code `driver_parallel_pid.m` generates these plots.

One can use the optional output argument `complete_history` (see § 2.3.2) to examine the difference between the parallel and serial algorithms in more detail. The `complete_history` structure records the successful points (*i.e.* those for which f returns a value), the values at the successful points, and the points where f failed to return a value. The fields in the structure are `complete_history.good_points`,

Figure 4.3. *Where is the function evaluated?*



complete_history.good_values, and complete_history.failed_points. The call to **imfil.m** looks like

```
[x,histout,complete_history]=imfil(x0,f,budget,bounds,options);
```

In Figure 4.3 we plot the good points for both the serial and parallel optimizations for the nonlinear least squares formulation of the parameter identification problem where the constraints are inactive at the solution. This is the computation from the lower right of Figure 4.2. We harvested the data from the `complete_history` structure with the MATLAB program `history_test.m`. This is another view of the example from the lower right corner of Figure 4.2 and shows that the function is evaluated in somewhat different places. The parallel method requires more function evaluations (58) than the serial (51), which is no surprise. Note how the evaluations cluster near the solution in both cases.

4.3 Using the `scale_aware` Option

The `scale_aware` option lets you design functions which can use the scale h to, for example, adjust their internal tolerances. This is very useful if the function is very expensive to evaluate, because then a coarse tolerance early in the iteration can save a significant amount of work. This is an example of how to use **imfil.m** as a multi-fidelity solver.

Here is a simple example. `sa_serial_pidlsq.m` is a scale-aware version of `serial_pidlsq.m`. The scale-aware version adjusts the tolerance sent to `ode15s` with the formula

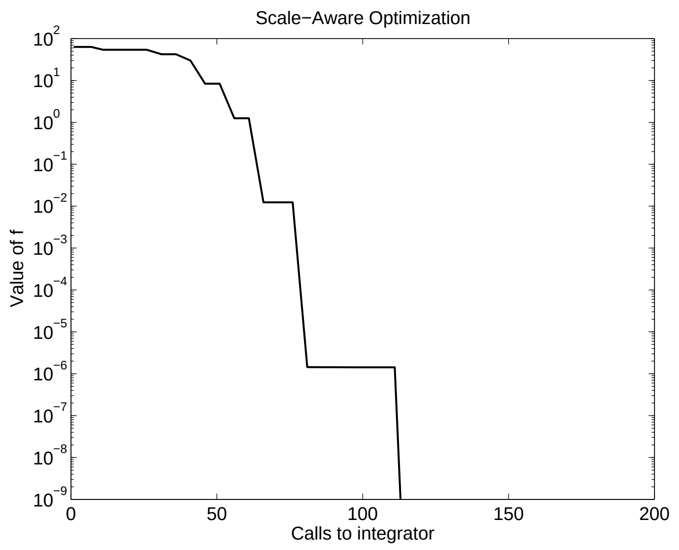
$$tol = h^2/10,$$

rather than using the tolerance from the `pid_info` structure. The calling sequence is

```
[f,ifail,icount]=sa_serial_pidlsq(x,h,pid_info)
```

Note that the scale is the second argument to the function. The final argument must always be the extra argument, if you are using one. We have also included a parallel version `sa_parallel_pidlsq.m`.

Figure 4.4. *A Scale-Aware Function*



The driver code `driver_sa.m` repeats the experiment for the unconstrained problem with a budget of 200 and `scaledepth` set to 20. This corresponds to the lower right plot in Figure 4.2. In Figure 4.4 you can see that the value of the function decreases beyond the leval at which it stagnated in the earlier computation. The reason for this is that the tolerance for the simulation keeps pace with the decrease in scales.

You should be aware that if you change h for a scale-aware function, you are also changing the function itself. One artifact you may see is an increase in the function value after you change scales. This should not be a surprise, since the function changes each time the tolerance changes, so `imfil` is solving a different problem with every change in scale. This did not happen in this example, but it certainly could have. While you might think that the accuracy of the simulator increases monotonically as the tolerances are tightened, there is no guarantee for that [25].

4.4 Termination Revisited

One striking feature of Figure 4.1 is how the least squares optimizations reached an optimal value by 50 or so iterations, but the iteration did not terminate until the list of scales or the budget was exhausted. Controlling this wasted effort is one of the important and unresolved issues in this field. The examples in this section illustrate some of your options and also show that no single approach will solve all the problems.

If you know something about the size of the noise in your function, **imfil.m** provides three options, `target`, `stencil_delta` and `function_delta`, which let you exploit that knowledge. For this example, we know the tolerance ($atol = rtol = 10^{-3}$) we've given to `ode15s` and can assume that the function evaluation is not much more accurate than that.

4.4.1 Using `function_delta` to Terminate the Iteration

Setting `function_delta` to $\delta > 0$ with the command

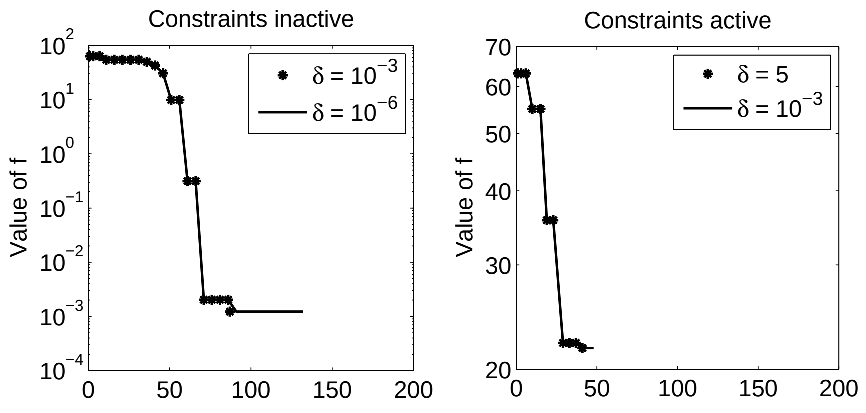
```
options=imfil_optset('function_delta',delta,options);
```

will cause the optimization to terminate as soon as the difference in f between successive iterations is $< \delta$. So, what's δ for this example? In Figure 4.5 we try two values, 10^{-3} and 10^{-6} for the unconstrained case and 5 and 10^{-3} for the constrained (large residual) case. If the constraints are active and the residual at optimality is large, the two values 10^{-3} and 10^{-6} did not save any calls to f, so we tried our luck with 5, a much larger value, which, as you can see from the plot on the left of Figure 4.5, was a very good choice. In the small residual case, setting `function_delta` is also very effective in eliminating the wasted function calls. Note that setting $\delta = 10^{-6}$ results in very little additional progress in this case.

The problem, of course, is picking the appropriate values of the parameter. While this example shows that you can do that, there is no theory to guide you in picking the right parameter, even if you have full knowledge of the errors in f.

Figure 4.5 was generated with the code `driver_term.m`. You might want to modify that code to experiment with the `target` and `stencil_delta` options.

Figure 4.5. *Terminating with* `function_delta`



4.4.2 Using the Executive Function

In this section we show how the `executive_function` option from § 3.5 can be used to replace the default Gauss-Newton nonlinear least squares solver with a Levenberg-Marquardt code. Our solver `lev_mar_exec.m` is in the `Imfil_Tools` directory. The driver `driver_lm.m` is in the `Examples/Case_Study_PID` directory. Look at § 3.5 and § 3.5.3 to see how the function is defined.

Very little needs to be done to use the solver. The driver compares the Levenberg-Marquardt code to the default Gauss-Newton solver with a scaledepth of 20. To use the default solver, set the options with

```
options=imfil_optset('least_squares',1,'scaledepth',20);
```

To use the new solver, you add settings for `executive_function` and `executive_data`. The `executive_data` in this case is the Levenberg-Marquardt parameter, which we initialize to 1. So the options are set with

```
options=imfil_optset('least_squares',1,'scaledepth',20,...
   'executive_function',@lev_mar_exec,'executive_data',1.0);
```

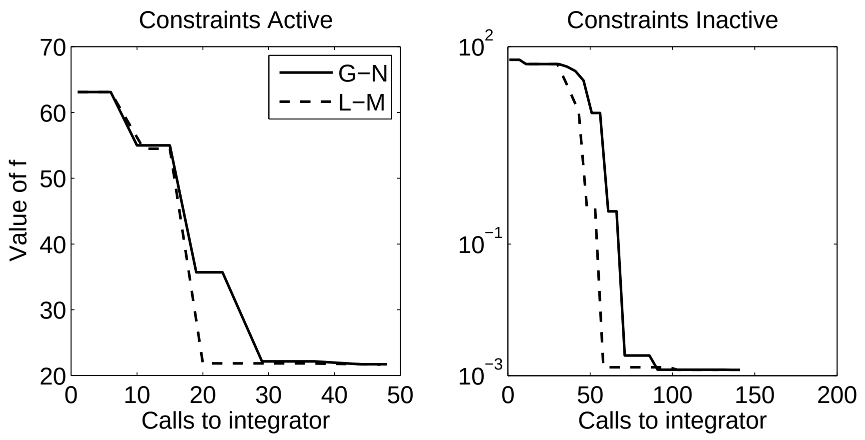
The call to **imfil.m** is the same for either choice

```
[x,histoutgx]=imfilv1(x0,@serial_pidlsq,budget,bounds,options,pid_info);
```

and is the same call we have used throughout this case study.

In Figure 4.6 we compare the two methods. As you can see the Levenberg-Marquardt iteration reduces the residual more rapidly, and significantly so in the case where the constraints are active.

Figure 4.6. *Levenberg-Marquardt Executive Results*



Bibliography

[1] C. Audet and J. E. Dennis, *Mesh adaptive direct search algorithms for constrained optimization*, SIAM J. Optim., 17 (2006), pp. 188–217.

[2] H. T. Banks and H. T. Tran, *Mathematical and experimental modeling of physical processes.* Department of Mathematics, North Carolina State University, unpublished lecture notes for Mathematics 573-4, 1997.

[3] J. T. Betts, *Practical Methods for Nonlinear Control Using Nonlinear Programming*, no. 3 in Advances in Design and Control, SIAM, Philadelphia, 2000.

[4] J. T. Betts, M. J. Carter, and W. P. Huffman, *Software for nonlinear optimization*, Tech. Rep. MEA-LR-083 R1, Mathematics and Engineering Analysis Library Report, Boeing Information and Support Services, June 6 1997.

[5] M. Buehren, *MULTICORE - parallel processing on multiple cores*, 2007. Available from the MATLAB Central File Exchange.

[6] R. Byrd, J. C. Gilbert, and J. Nocedal, *A trust region method based on interior point techniques for nonlinear programming*, Mathematical Programming A, 89 (2000), pp. 149–185.

[7] T. D. Choi, O. J. Eslinger, P. Gilmore, A. Patrick, C. T. Kelley, and J. M. Gablonsky, *IFFCO: Implicit Filtering for Constrained Optimization, Version 2*, Tech. Rep. CRSC-TR99-23, North Carolina State University, Center for Research in Scientific Computation, July 1999.

[8] A. R. Conn, N. I. M. Gould, and P. L. Toint, *LANCELOT: A Fortran Package for Large-Scale Nonlinear Optimization (Release A)*, no. 17 in Springer Series in Computational Mathematics, Springer Verlag, Heidelberg, Berlin, New York, 1992.

[9] A. R. Conn, K. Scheinberg, and L. N. Vicente, *Introduction to Derivative-Free Optimization*, MPS-SIAM Series on Optimization, SIAM, Philadelphia, 2009.

[10] J. E. Dennis and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, no. 16 in Classics in Applied Mathematics, SIAM, Philadelphia, 1996.

[11] D. E. Finkel and C. T. Kelley, *Convergence analysis of sampling methods for perturbed Lipschitz functions*, Pacific J. Opt., 5 (2009), pp. 339–350.

[12] R. Fletcher, *Practical methods of optimization*, John Wiley and Sons, New York, 1987.

[13] K. R. Fowler, J. P. Reese, C. E. Kees, J. E. Dennis, C. T. Kelley, C. T. Miller, C. Audet, A. J. Booker, G. Couture, R. W. Darwin, M. W. Farthing, D. E. Finkel, J. M. Gablonsky, G. Gray, and T. G. Kolda, *A comparison of derivative-free optimization methods for groundwater supply and hydraulic capture problems*, Advances in Water Resources, 31 (2008), pp. 743–757.

[14] P. E. Gill, W. Murray, and M. A. Saunders, *SNOPT: An SQP algorithm for large-scale constrained optimization*, SIAM Review, 47 (2005), pp. 99–131.

[15] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*, Academic Press, London, 1981.

[16] P. D. Hough, T. G. Kolda, and V. J. Torczon, *Asynchronous parallel pattern search for nonlinear optimization*, SIAM J. Sci. Comput., 23 (2001), pp. 134–156.

[17] *IEEE Standard for Binary Floating Point Arithmetic, Std 754-1885*, 1985.

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

[19] ——, *Implicit Filtering*, no. 23 in Software Environments and Tools, SIAM, Philadelphia, 2011.

[20] J. Kepner, *Parallel MATLAB for Multicore and Mulitnode Computers*, no. 21 in Software Environments and Tools, SIAM, Philadelphia, PA, 2009.

[21] B. W. Kernighan and L. L. Cherry, *Typesetting Mathematics – User's Guide*, AT&T Bell Laboratories, Murray Hill, New Jersey, 1979. In Unix Seventh Edition Manual, Volume 2.

[22] T. G. Kolda, R. M. Lewis, and V. J. Torczon, *Optimization by direct search: New perspectives on some classical and modern methods*, SIAM Review, 45 (2003), pp. 385–482.

[23] R. M. Lewis and V. Torczon, *Rank ordering and positive bases in pattern search algorithms*, Tech. Rep. 96-71, Institute for Computer Applications in Science and Engineering, December 1996.

[24] ——, *Pattern search algorithms for linearly constrained minimization*, SIAM J. Optim., 10 (2000), pp. 917–941.

[25] W. Lioen, J. de Swart, and W. van der Veen, *Test set for IVP solvers*, tech. rep., Centrum voor Wiskunde en Informatica, Department of Numerical Mathematics, Project Group for Parallel IVP Solvers, December 23 1996.

[26] G. Marsaglia, *Choosing a point from the surface of a sphere*, Ann. Math. Stat., 43 (1972), pp. 645–646.

[27] M. E. Muller, *A note on a method for generating points unformly on N-dimensional spheres*, Comm. ACM, 2 (1959), pp. 19–20.

[28] J. Nocedal and S. J. Wright, *Numerical Optimization*, Springer, New York, 1999.

[29] M. L. Overton, *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM, Philadelphia, 2001.

[30] L. M. Rios and N. V. Sahinidis, *Derivative-free optimization: A review of algorithms and comparison of software implementations.* unpublished draft manuscript, 2009.

[31] L. F. Shampine and M. W. Reichelt, *The MATLAB ODE suite*, SIAM J. Sci. Comput., 18 (1997), pp. 1–22.

[32] R. J. Vanderbei, *LOQO: An interior point code for quadratic programming*, Optimization Methods and Software, 11 (1999), pp. 451–484.

[33] 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.

Index