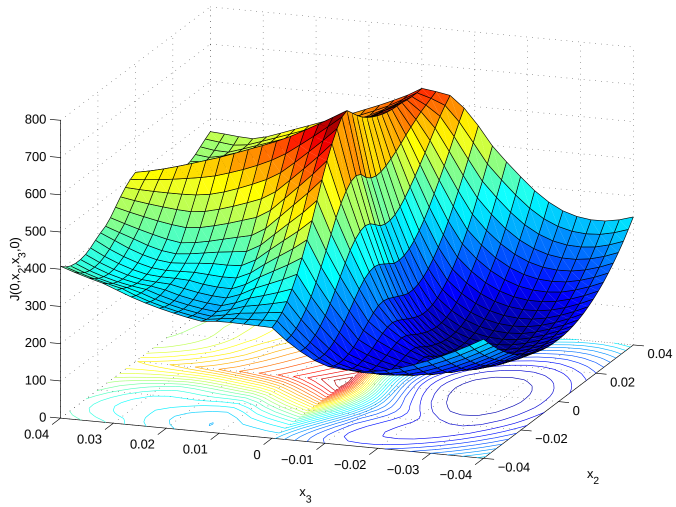


Users' Guide for imfil Version 0.8

C. T. Kelley

Version of November 11, 2009

Contents

Preface

This is the users' guide for the MATLAB version of implicit filtering **imfil.m**. As of November 11, 2009, the code is ready for beta-testing. You will find that parts of the code, especially some of the parts deep inside, are still poorly documented.

I assume that you have a background in optimization at the level of [11, 20]. 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.

A 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, will wind up in [22], a book in preparation. If you want a copy of the draft of the book, let me know. The draft will become a SIAM book sometime in 2010.

Implicit filtering is a projected quasi-Newton method for bound constrained optimization problems. The gradients are computed with a finite difference and the difference increment varies as the optimization progresses. The points on the difference stencil are also used to guide the search.

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

We will draw a distinction between pure sampling methods, such as the Nelder Mead [28] and Hooke-Jeeves [18] algorithms, coordinate search, and the many variations of multidirectional search [1,12,25], and what we will call interpolatory methods, where second derivative information is harvested from the sampling by interpolation [9, 10, 17, 30, 31]. Implicit filtering lies in the middle and seeks to exploit the advantages of pure sampling for discontinuous and highly oscillatory problems, while using first-order interpolation and a quasi-Newton model Hessian to capture the rapid convergence of interpolatory methods for problems which are well-modeled by smooth approximations.

As implicit filtering has evolved since [34], it and several related approaches have moved closer together. In the current version, as reflected in this book, ideas from [1, 10, 19, 27] have found their way into the implementation in **imfil.m**.

imfil.m is a MATLAB implementation of the implicit filtering method. This version differs in significant ways from our older FORTRAN code [7], and we support only the MATLAB version now. This document is a complete reference to Version

0.8 of **imfil.m**, covering installation, testing, and its use in both serial and parallel environments. 0.8 is an update of 0.7, with some new features and the expected bug fixes.

This is a work in progress, as you will see when you run into the **boldface notes to myself** that are scattered all over the place.

C. T. Kelley
Raleigh, North Carolina
September, 2008

Latest changes:

- You can now use an `add_new_directions` option to add directions to the stencil adaptively. This is useful, for example, if you have linear constraints and want to add some tangent directions if a constraint is nearly active.
- **imfil.m** maintains a data structure which contains the history of the iteration. I use this internally to make sure **imfil.m** does not evaluate the function twice at the same point. You can get to it as the final output argument.
- We have made the nonlinear least squares solver better. It's still based on [21], but is better, cleaner, and more robust.
- I continue to mess with the details of the options structure. This will only affect you if you have been **hacking** that structure.

What's next?

- I have the Levenberg-Marquardt iteration working, but its performance has been no better than the damped Gauss-Newton I have in **imfil.m** now. This will get in there, but will be an example of how to put in your own model Hessian and globalization scheme. This is part of . . .
- The final big deal part of **imfil.m** is letting you take over the inner loop. I plan to call this the **executive** option. This will let you do things like (1) replace the model Hessian with anything you like and use a trust region method instead of the line search and/or (2) follow the quasi-Newton iteration by investing some function evaluations in your favorite global optimization method.

How to get the software

The codes live at

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

On that page you will find

- A pdf file of the users' guide [23].
- **imfil.m**
  This is the main implicit filtering code. This manual is current as of November 11, 2009. This is version 0.8. This version is a significant change from 0.7. If you are using any version prior to 0.7, please upgrade to 0.8. Of course, let me know if you have any problems.
- **imfil_optset.m** handles the options.
- The codes which generate Figures 1.1 and 1.2: **pid_example_chapter_1.m**, **pidobj.m**, and **pidlsq.m**.

One can obtain MATLAB from
The MathWorks, Inc.
24 Prime Park Way
Natick, MA 01760,
(508)653-1415
Fax: (508)653-2997
Email: info@mathworks.com
WWW: http://www.mathworks.com

Chapter 1

A Simple Example

In this chapter we give a brief description of what **imfil.m** does, and illustrate its use for a simple problem. We do not describe all the options, the details of the algorithms, or the ways to extend the code on your own. The later chapters will do that.

1.1 Computing Environment

As of November 11, 2009, 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.

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.

1.2 What imfil.m does

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

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

by which me 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 **hyper-rectangle** 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, 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 [14, 16, 29] f is a smooth (*i. e.* twice Lipschitz continuously differentiable) function and Ω can be described by smooth inequality constraints, *i. e.*

$$\Omega_G = \{x \in R^N \,|\, g_i(x) \leq 0, 1 \leq i \leq M\}. \tag{1.3}$$

There are several good gradient-based methods and codes for solving this classical problem [3,4,6,8,15,33]. 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

$$x_c \pm hv_i \; 1 \leq i \leq N,$$

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 sequence of scales is

$$\{2^{-n}\}_{n=1}^{\texttt{scaledepth}}.$$

The algorithmic parameter `scaledepth` can be changed from the default of 7 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 want to query the history of the iteration using the `add_directions` or `consultant` options.

These options are on the way, but not here yet.

1.2.1 Terminating the Iteration

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 must communicate a failure to **imfil.m**. We present an example below.

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.

1.3 Basic Usage

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.

So, the call to f would look like

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

In the most simple case, $icount$ will the the number of calls to f. If some failed points cost more than others, or you want $icount$ to reflect some other measure of cost, such as wall clock time, you may do that as well. The budget will be computed in terms of $icount$.

You should put the bounds in a $N \times 2$ array, with L in the first column and U in the second.

Finally, you must specify 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. § 2.8 and § 2.9 describe other ways to terminate the iteration.

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. Remember that if your objective function is a MATLAB .m file, say `myfun.m`, you'll have to put single quotes around the 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.

The `histout` array is an $IT \times (N+5)$ dimensional array, where IT is the number of quasi-Newton iterations used in the optimization. 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 A Simple Example

We will use an example from [2,20]. In this example $N = 2$. The goal is to identify the damping c and spring constant k of a linear spring by minimizing the difference of 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.

The motion of an unforced harmonic oscillator satisfies the initial value problem

$$u'' + cu' + ku = 0; u(0) = u_0, u'(0) = 0, \tag{1.4}$$

on the interval $[0, T]$. In (1.4) $u' = du/dt$ 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 (1.4). If the displacement is sampled at $\{t_j\}_{j=1}^M$, where $t_j = (j-1)T/(M-1)$, and the observations for u are $\{u_j\}_{j=1}^M$, then the objective function is

$$f(x) = \frac{1}{2}\sum_{j=1}^{M} |u(t_j : x) - u_j|^2. \tag{1.5}$$

This is a **nonlinear least squares** problem, but we will ignore the nonlinear least squares structure for the present. We show how to solve the nonlinear least squares problem in § 1.3.3 and 1.3.5.

We will use MATLAB's `ode15s` [32] to solve (1.4), and use the solution from `ode15s` to compute f. The first step in using `ode15s` is to convert (1.4) 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

$$y' = F(y) = \begin{pmatrix} v \\ -cv - ku \end{pmatrix}, \tag{1.6}$$

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

When one uses an integrator like `ode15s`, one is asked to provide a local truncation error tolerance. This tolerance controls the accuracy of the integration, and thereby the resolution in f.

The examples for this chapter are in a subdirectory of the software collection, and this first example uses the files

- `pid_example_chapter_1.m`, which is the driver for all the examples in this chapter, and
- `pidobj.m`, which calls the least squares formulation `pidlsq.m` to compute f using `ode15s`.

We sample at $M = 101$ equally spaced points $\{i/100\}$ for $0 \le i \le 100$ and configure the problem so that the solution is $(c, k) = (1, 1)$. The objective function is `pidobj.m`, and the first line is

```
function [f,ifail,icount]=pidobj(x)
```

Functions you write must **AT LEAST FOR NOW** conform to this paradigm. Even if your function never fails and always does the same amount of work when called, **imfil.m** needs $ifail$ and $icount$. **This will get fixed soon.**

`pidobj.m` has a failure mode. If either c or k is negative, the spring is not physical. The code traps this and returns without calling the integrator. You could fix this yourself, by making sure that the lower bounds you give to **imfil.m** are all nonnegative. We put this in as an example so you can see how to do it yourself.

Here is a sketch of what happens when `pidobj.m` receives x. You can look at `pidlsq.m` and `pidobj.m` for the details. The first thing to do is test for a negative component of x. The MATLAB looks like

```
ifail=0; icount=1;
%
% Call the integrator only if x is physically reasonable, ie if
% x(1) and x(2) are nonnegative. Otherwise, report a failure.
%
if min(x) < 0
   ifail=1; icount=0; f=NaN;
else
%  call the integrator and do the work
end
```

We omit the details of the call to the integrator, and the way `pidobj.m` uses global variables. The curious reader can look at the source in the software collection.

1.3.2 Calling imfil.m and Looking at Results

We can now show how the simplest call would work. The plots in the upper row of Figure 1.1 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 were

```
bounds=[2 20; 0 5];
x0=[5,5]'; budget= 100;
options=imfil_optset('scaledepth',5,options);
[x,histout]=imfil(x0,'pidobj',budget,bounds,options);
```

As you can see from the plot on the upper left of Figure 1.1, the iteration terminated before the budget had been exhausted. We can return to the default set of scales by reinitializing the `options` structure

```
options=imfil_optset;
```

and calling `imfil.m` again. The picture on the upper right reflects the results of this change. Now the optimization requires almost the entire budget, and the final value of the objective function is much lower.

One can query the `histout` array to see how far the optimization got in the list of scales. All the plots were made with the first two columns of the `histout` array. The plots at the top of Figure 1.1 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 that we have resolved the optimal point as far as the resolution in the integrator will allow.

The plots on the bottom of Figure 1.1 are from an optimization where the global minimum is within the bounds. Here we set

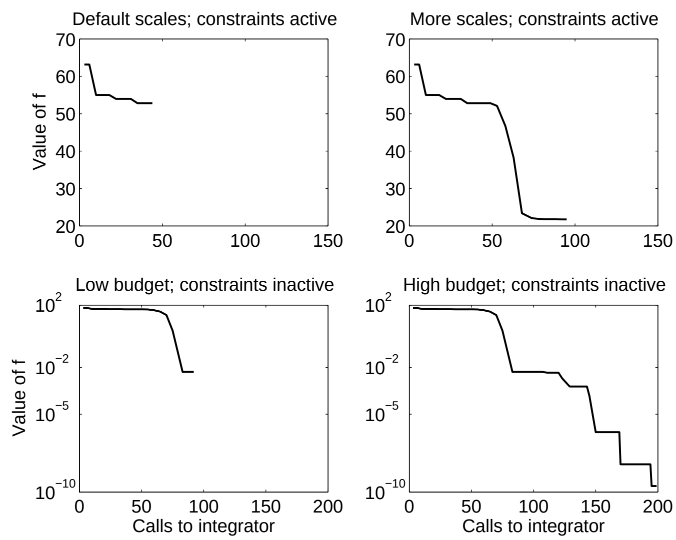
```
bounds=[0 20; 0 5];
```

The lower left plot in Figure 1.1 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 insufficient to fully resolve the optimal value. The lower right plot shows the results with a budget of 200 and 20 scales. The results are very different, which 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? These are open research questions at this time (2008). In § 2.8 § 2.9 we explore the options in **imfil.m** for termination.

1.3.3 Nonlinear Least Squares

Many problems, such as the example in this section, are best formulated as nonlinear least squares problems, where F returns an vector of residuals in R^M and the

Figure 1.1. *Optimization History: Parameter ID*



function to be minimized is

$$f(x) = F(x)^T F(x)/2. \tag{1.7}$$

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,options);
```

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.

1.3.4 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 M 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 § 1.3.5 shows how **imfil.m** responds to

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

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 $N \times P$ 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 § 2.6 for the details). One implication of this is that more function evaluations will be used even if the final results is the same as in the serial case and the total runtime is significantly less. One should interpret graphs like Figure 1.1 with care when one does the function evaluations in parallel. The default is $parallel = 0$.

The latest versions of MATLAB support some parallelism, and there are some very useful resources in the **MATLAB Central File Exchange**. **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 parallel toolbox provides the `parfor` loop. Similarly to MULTICORE the `parfor` loop runs a copy of MATLAB on each core. The software associated with [24] has the `pRUN` program, which allows you to run the same MATLAB code on multiple processors. These approachs 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.

1.3.5 Revisiting the Example

We will close this section by reexamining the results in Figure 1.1 by comparing the serial optimization results in that figure with a serial least squares 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 lower right plot in Figure 1.2, the serial and parallel iteration histories can differ by over 50%, 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 extreme example of the difference between the parallel and serial algorithms. The plot in the upper right, which we explore in more detail below, shows similar performance of the serial and parallel algorithms, but if one keeps in mind that the parallel algorithm is doing at least twice the work at each iteration, the parallel method should be faster.

One can use the optional output argument `complete_history` 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`, `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);
```

The `complete_history` is used inside **imfil.m** to avoid redundant function calls.

In Figure 1.3 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 another view

Figure 1.2. *Optimization History: Parameter ID Revisited*

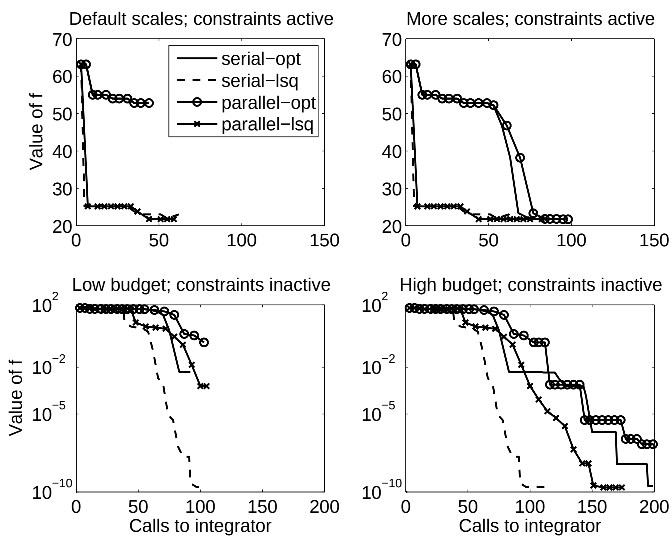
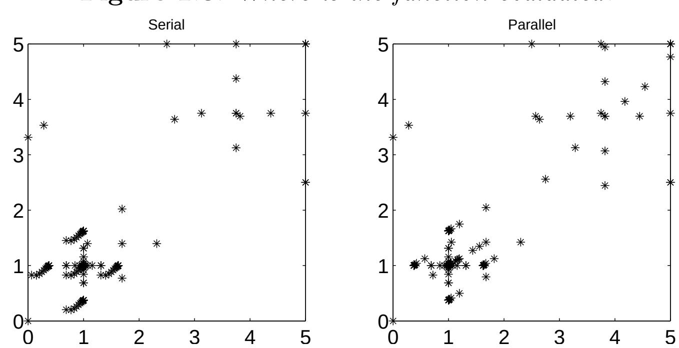


Figure 1.3. *Where is the function evaluated?*



of the example from the lower right corner of Figure 1.2 and shows that the function is evaluated in somewhat different places. The parallel method requires more function evaluations (163) than the serial (137), which is no surprise. Note how the evaluations cluster near the solution in both cases.

1.4 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. You have already seen the `scaledepth` option. We will explain the details for all the options in § 2.3.

Before setting the options, you need to get the default options structure 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 `fscale` to 1.0 and `scaledepth` to 10, you could call `imfil_optset` three times:

```
options=imfil_optset;
options=imfil_optset('fscale',1.0,options);
options=imfil_optset('scaledepth',10,options);
```

prior to the call to **imfil.m**. You can also put all the calls to optset on a single line

```
options=imfil_optset('fscale',1.0,'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, the call would be

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

1.4.1 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. See § 2.4.1 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 to the scale and make a termination decision. The story on that will appear in § 2.8. **It is not in there yet. Stay tuned.**

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 § 2.8.

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 [22] 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 and the list of scales are input arguments to **imfil.m** and this mode of termination usually works well. One can do more, and set several 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 § 2.8 for the details.

1.4.2 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

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

where e_i is the unit vector in the ith coordinate direction, and h, the **scale** varies as the optimization progresses. 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 $scaledepth = 7$.

imfil.m has a few options for messing with the stencil and future versions will let you do all kinds of stuff.

1.4.3 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. 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.

Put an example in here!

Chapter 2

Using imfil.m

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

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

The last input argument `options` is optional. If the `options` argument is omitted, **imfil.m** will use the defaults.

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

2.1 Input

The input data are

- $x0 \in R^N$: the initial iterate,
- f: the function to be minimized,
- *budget*, the maximum number of function evaluations allowed to the optimization,
- the bounds array `bounds`, and
- the `options` structure.

We will discuss all but the `options` argument in this section. We will explain the `options` at length in § 2.3.

2.1.1 The Initial Iterate

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

$$bounds(i,1) \leq x0(j) \leq bounds(i,2),$$

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

2.1.2 The Objective Function f

If the `parallel` option is off $= 0$, the calling sequence for f should be

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

or

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

if your function is **scale-aware**, *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. .

If $f(x)$ successfully returns a value, $fout$ 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 may elect to return a NaN as the value when this happens, but that is not required. **imfil.m** will eliminate failed points from the stencil when computing the stencil gradient.

If the `parallel` option is on ($= 1$), then **imfil.m** will send an array of input arguments to f. These arrays vary in size. **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 three points $\{x + \lambda d\}$ for $\lambda = 1, 1/2, 1/4, 1/8$. Your parallel function must be able to accept an $N \times M$ array of M arguments to f, and return three $M \times 1$ arrays of values for $fout$, $ifail$, and $icount$. It is your job to construct your function to use what parallelism you have efficiently.

2.1.3 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). The optimization is likely to finish over budget because **imfil.m** does not stop the outer (optimization) loop in mid-stream.

2.1.4 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.

2.2 Output and Troubleshooting

The output of **imfil.m** are x an approximation of the solution, `histout`, an iteration history, and (optionally) the `complete_history` structure, which contains every point where **imfil.m** has evaluated f and either the value of f or a failure flag.

2.2.1 The `histout` array

The `histout` array is an $IT \times (N+5)$ dimensional array, where IT is the number of quasi-Newton iterations used in the optimization. For optimization problems

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

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 f, the norm of 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.

When **imfil.m** reduces the scale because of stencil failure, the `histout` array indicates this by setting $iarm = -1$, to indicate that no quasi-Newton work at all was done.

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

2.2.2 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/ParamID` 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**.

2.2.3 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 you got stencil failure with each iteration. That is an indicator that you could terminate the optimization earlier by either changing `scaledepth` (§ 2.5.1) or `target` (§ 2.8.1).

If $iarm = maxitarm$ 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. If this happens 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 (§ 2.5.4). Your function may also be poorly scaled, and changing `fscale` (§ 2.4.1) can help.

2.3 Setting Options

You can change most of **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. So, 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

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

If you wish to use the `options` structure, you add that as a final 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);
```

2.4 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.

2.4.1 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 function by dividing it by the size of a "typical value", which we call $imfil\_scale$.

The default is

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

which is usually fine.

If $imfil\_scale$ 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 $imfil\_scale$ 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

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

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

$$imfil\_scale = fscale.$$

$fscale = 0$ is not a sensible value; if you blunder and set $fscale = 0$, **imfil.m** will restore the default.

2.4.2 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 [20] formulation of the quasi-Newton methods.

2.4.3 Nonlinear Least Squares

imfil.m will also solve nonlinear least squares problems where

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

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 F is returned. **imfil.m** will compute f for you.

As of November 11, 2009 the solver is a projected damped Gauss-Newton iteration [11, 20, 22].

2.4.4 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_{arm}, **imfil.m**will select x_{arm} to be the new point as long as the line search succeeds, *i. e.*

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

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

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

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

2.4.5 Limiting the Quasi-Newton Direction

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` 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 **at least in version 0.85** is 'yes', which is a good choice for noisy problems. For nearly smooth problems, 'no' is better.

2.5 Managing and Using the Scales

2.5.1 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 $1 \leq i \leq m$. 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.

2.5.2 Controlling the Number of Scales

imfil.m has an outer iteration over the scales, an inner iteration in the finite-difference quasi-Newton loop, and an iteration within the line search. All of these iterations have limits you can set. You set the limit on the number of scales with `scalestart` and `scaledepth` (see § 2.5.1).

2.5.3 Custom Scales

The default scales are $\{2^{-n}\}_{n=1}^{7}$. By using `scaledepth` and `scalestart` you can change that to

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

The iteration will terminate when the supply of scales has been exhausted.

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 the internal scaling in **imfil_core.m** would put all points in the stencil outside of the bound constraints. You set this option with

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

2.5.4 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);
```

Put an example of a scale-aware f somewhere.

2.6 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 M arguments to f and `parallel` is set to 1, a call to $f(x)$ will return a 3 $1 \times P$ vectors of values and flags.

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 $N \times P$ array of residual values as well as the row vectors of $iflag$ 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 $parallel = 0$.

Code fragments will go here.

2.7 Stencils

As for November 11, 2009, **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** [25, 26] 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, 1 for the one-sided stencil, and 2 for the positive basis stencil.

2.7.1 Vstencil

If you don't like any of the build-in stencils, you may create your own 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 of avoid stagnation when the stencil directions are insufficient.

2.7.2 Random Vectors

You can augment the stencil with k random vectors by setting the `random_stencil` option to k. The theory from [1, 13] will apply if $k > 1$.

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 make it likely that stencil failure will not happen, and then the iteration spends too much time in the line search.

If you suspect that the optimal point is on a constraint boundary, especially a hidden constraint boundary, use this option.

2.8 Terminating The Outer Iteration

Most problems can be solved with the default termination criteria for the optimization (or outer) iteration. These are exhausting the list of scales or exceeding the budget. Sometimes, however, you may know things that can help **imfil.m** do its job better.

2.8.1 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.

2.8.2 Function_Error

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 `function_error` 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 `function_error`. To turn this optional termination test on set the `function_error` option to your estimate of the error. The default is -1 which means the option is off.

2.9 Terminating the Inner Iteration

The nonlinear (or inner) iteration has its own termination criteria. Changing the termination criteria for the inner iteration requires detailed knowledge of how the quasi-Newton loops are implemented, and we refer the reader to [22] for the details.

You have a good deal of control over the inner iteration, but should take care before messing about with these options.

`maxit` is the upper limit on the number of quasi-Newton (or Gauss-Newton) iterations. The default is 50.

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`.

2.10 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 the options which change the behavior of **imfil_core.m**. One must take care with these options because both the function and the bounds have been scaled within **imfil_core.m**.

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 file you will want **imfil_core.m** to call before computing the stencil derivative.

So if your function is `my_directions.m`, 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_core.m** will call your function to add directions immediately before computing the stencil derivative. **imfil_core.m** will also undo its own internal scaling and give you x and V in your original coordinates. When you return you new directions to **imfil_core.m** they are rescaled and normalized interally.

The example `linear_constraints.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 uses the tangent directions to a linear constraint when it is nearly active, and ignores the linear constraint otherwise. This j avoids stagnation when the stencil directions are insufficient and, unlike using `vstencil` does not add more directions when they are not needed.

3.2 The Executive Function

3.2.1 Testing More Points

Bibliography

[1] C. Audet and J. E. Dennis, *Mesh adaptive direct search algorithms for constrained optimization*, Tech. Rep. TR04-02, Department of Computational and Applied Mathematics, Rice Univeristy, 2004.

[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 P. L. Toint, *Recent progress in unconstrained optimization without derivatives*, Math. Prog. Ser. B, 79 (1997), pp. 397–414.

[10] A. R. Conn, K. Scheinberg, and L. N. Vicente, *Introduction to derivative-free optimization.* unpublished manuscript, 2007.

[11] 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.

[12] J. E. Dennis and V. Torczon, *Direct search methods on parallel machines*, SIAM J. Optim., 1 (1991), pp. 448 – 474.

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

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

[15] 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.

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

[17] H.-M. Gutmann, *A radial basis function method for global optimization*, Journal of Global Optimization, 19 (2001), pp. 201–227.

[18] R. Hooke and T. A. Jeeves, *'Direct search' solution of numerical and statistical problems*, Journal of the Association for Computing Machinery, 8 (1961), pp. 212–229.

[19] 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.

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

[21] ——, *Implicit filtering and nonlinear least squares problems*, in System Modeling and Optimization XX, E. W. Sachs and R. Tichatschke, eds., Dordrecht, 2003, Kluwer Academic Publishers, pp. 71–90.

[22] ——, *Implicit Filtering.* Unfinished manuscript; to be published by SIAM 2009, 2008.

[23] ——, *Users' Guide for imfil Version 0.8.* Software manual, work in progress, 2009.

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

[25] 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.

[26] 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.

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

[28] J. A. Nelder and R. Mead, *A simplex method for function minimization*, Comput. J., 7 (1965), pp. 308–313.

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

[30] M. J. D. Powell, *UOBYQA: unconstrained optimization by quadratic approximation*, Math. Programming, 92 (2002), pp. 555–582.

[31] R. G. Regis and C. A. Shoemaker, *Local function approximation in evolutionary algorithms for the optimization of costly functions*, IEEE Transactions on Evolutionary Computation, 8 (2004), pp. 490–505.

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

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

[34] 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