immoptibox

A MATLAB TOOLBOX FOR
OPTIMIZATION AND DATA FITTING

Hans Bruun Nielsen
DTU Informatics, IMM
Technical University of Denmark

Version 2.2. November 2010

The toolbox contains a number of functions for optimization an data fitting. The selection of algorithms was guided by the DTU course 02610 Optimization and Data Fitting, but the MATLAB functions in the toolbox are expected also to have wider interest.

To get the toolbox, download immoptibox.zip from

http://www2.imm.dtu.dk/~hbn/immoptibox/

to the directory, where you save your MATLAB files, use unzip to unpack it, and update your MATLAB path.

See History about the changes since the previous version of the toolbox.
Contents

1. Unconstrained optimization
   1.1. General optimization 3
       dampnewton 3
       linesearch 4
       ucminf 5
       Example 6
   1.2. Least squares problems 8
       dogleg 8
       marquardt 9
       smarquardt 10
       Example 11

2. Data fitting
   2.1. Cubic splines 13
       splinefit 13
       splineval 14
       splinedif 14
       Example 15
   2.2. Robust estimation 16
       linhuber 16
       nonlinhuber 17
       huberobj 18
       Example 19
   2.3. Multiexponential fitting 21
       mexpfit 21
       Example 22

3. Miscellaneous
   3.1. Nonlinear systems 23
       nonlinsys 23
       Example 24
   3.2. Auxiliary programs 25
       immoptset 25
       checkgrad 26
       Example 26
   3.3. Test problems 27
       uctpget 27
       uctpval 28
       Example 29
       Data sets 29

4. History 31

5. References 32
1. Unconstrained Optimization

1.1. General Optimization

We seek a minimizer \( \hat{x} \) of a function \( f : \mathbb{R}^n \rightarrow \mathbb{R} \). A minimizer satisfies \( \nabla f(\hat{x}) = 0 \), where \( \nabla f \in \mathbb{R}^n \) is the gradient.

The function \( \text{dampnewton} \) calls a user-supplied MATLAB function with a header of the form

\[
\text{function \ [f, g, H] = fun(x,p1,p2,...)}
\]

The function should return \( f(x) \) in \( f \), the gradient \( \nabla f(x) \) in \( g \), and the Hessian \( \nabla^2 f(x) \) in \( H \).

The functions \( \text{linesearch} \) and \( \text{ucminf} \) only need a user-supplied function of the form

\[
\text{function \ [f, g] = fun(x,p1,p2,...)}
\]

that returns the values of the function and its gradient.

The toolbox functions allow a list of parameters \( p1,p2,... \) to be passed to \( \text{fun} \).

The implementation of \( \nabla f(x) \) and \( \nabla^2 f(x) \) can be checked by \( \text{checkgrad} \), Section 3.2.2.

1.1.1. User’s guide to \( \text{dampnewton} \)

This function is based on the algorithm described in [4, Section 3.2]. Given the iterate \( x \), the step \( h \) to the next iterate is found as the solution to the linear system given to the right.

\[
\left( \nabla^2 f(x) + \mu I \right) h = - \nabla f(x)
\]

The damping parameter \( \mu \) is a positive number. For \( \mu \) sufficiently large the matrix is positive definite, and \( h \) is guaranteed to be a descent direction. During iteration \( \mu \) is monitored so that, as \( x \) approaches \( \hat{x} \), the damping tends to zero, so we approach the Newton method with quadratic convergence.

Typical calls are

\[
\begin{align*}
[X, \ info] & = \text{dampnewton}(fun, x0) \\
[X, \ info] & = \text{dampnewton}(fun, x0, opts, p1,p2,...) \\
[X, \ info, perf] & = \text{dampnewton}(.....)
\end{align*}
\]

Input parameters

- \( \text{fun} \) Handle to the function.
- \( x0 \) Starting guess for \( \hat{x} \).
- \( \text{opts} \) Either a struct with fields ‘tau’, ‘tolg’, ‘tolx’ and ‘maxeval’, or a vector with the values of these options, \( \text{opts} = [\text{tau} \ \text{tolg} \ \text{tolx} \ \text{maxeval}] \).
- \( \text{tau} \) is used to get starting value for the damping parameter:

\[
\mu = \text{tau} * \| \nabla^2 f(x0) \|_{\infty}
\]

The other options are used in the stopping criteria (3.1),
\begin{align*}
\|\nabla f(x)\|_\infty & \leq \text{tolg} \quad \text{or} \\
\|\delta x\|_2 & \leq \text{tolx}(\text{tolx} + \|x\|_2) \quad \text{or} \\
\text{no. of function evaluations exceeds maxeval}
\end{align*}

Default: $\tau = 1e^{-3}$, $\text{tolg} = 1e^{-4}$, $\text{tolx} = 1e^{-8}$, $\text{maxeval} = 100$.

If the input $\text{opts}$ has less than 4 elements, it is augmented by the default values.
Also, zeros and negative elements are replaced by the default values.

$p_1, p_2, \ldots$ are passed directly to the function $\text{fun}$.

**Output parameters**

$X$  
If $\text{perf}$ is present, then $X$ is an array, holding the iterates columnwise, with the computed solution in the last column.
Otherwise, $X$ returns the computed solution vector.

$\text{info}$  
Performance information, vector with 7 elements:

- $\text{info}(1:4)$: Final values of $[f(x), \|\nabla f(x)\|_\infty, \|\delta x\|_2, \mu/\max(|(\nabla^2 f(x))_{ii}|)]$.
- $\text{info}(5:6)$: Number of iteration steps and function evaluations.
- $\text{info}(7)$ = 1: Stopped by a small gradient.
  2: Stopped by a small $x$-step,
  3: No. of function evaluations exceeds $\text{opts}(4)$
  -1: $x$ is not a real valued vector.
  -2: $f$ is not a real valued scalar.
  -3: $g$ is not a real valued vector or $H$ is not a real valued matrix.
  -4: Dimension mismatch in $x$, $g$, $H$.
  -5: $H$ is not symmetric.

$\text{perf}$  
Struct with fields $f$: values of $f(x)$, $\text{ng}$: values of $\|\nabla f(x)\|_\infty$, $\text{mu}$: values of damping parameter $\mu$.

**1.1.2. User’s guide to linesearch**

This function is based on the algorithm described in [4, Section 2.3]. Given $x$ and a descent direction $h$, the task is to find an approximate minimizer in that direction, ie a minimizer of the function

$$
\varphi(\alpha) = f(x + \alpha h), \quad \varphi'(\alpha) = h^T \nabla f(x + \alpha h).
$$

$\text{linesearch}$ can be used both for soft and exact line search. In the former case we are satisfied with a point $\hat{\alpha}$ in the acceptable region, sketched in the figure, ie $\varphi(\hat{\alpha}) \leq \varphi(0) + \hat{\alpha}\beta_1\varphi'(0)$ and $\varphi'(\hat{\alpha}) \geq \beta_2\varphi'(0)$. $\beta_1 < \beta_2$ are input.

In case of exact line search we seek $\hat{\alpha}$ such that $|\varphi'(\hat{\alpha})| \leq \delta_1|\varphi'(0)|$ or $|b - a| \leq \delta_2 b$, where $[a,b]$ is the current interval for $\hat{\alpha}$.

Typical calls are

- $[\text{xn}, \text{fn}, \text{gn}, \text{info}] = \text{linesearch}(\text{fun}, x, f, g, h)$
- $[\text{xn}, \text{fn}, \text{gn}, \text{info}] = \text{linesearch}(\text{fun}, x, f, g, h, \text{opts}, p_1, p_2, \ldots)$
- $[\text{xn}, \text{fn}, \text{gn}, \text{info}, \text{perf}] = \text{linesearch}(\ldots)$
1.1. General Optimization

**Input parameters**

- **fun** Handle to the function.
- **x** Current $x$.
- **f,g** $f(x)$ and $\nabla f(x)$.
- **h** Step vector.

**opts** Either a struct with fields 'choice', 'cp1', 'cp2', 'maxeval', 'amax' or a vector with the values of these options, $\text{opts} = [\text{choice} \ \text{cp1} \ \text{cp2} \ \text{maxeval} \ \text{amax}]$.
  - **choice** = $0$: exact line search.
    Otherwise soft line search (Default).
  - **cp1, cp2**: options for stopping criteria.
    - **choice** = $0$: $|\phi'(a)| \leq \text{cp1} * |\phi'(0)|$ or $c - b \leq \text{cp2} * c$, where $a = \hat{\alpha}$ and $[b, c]$ is the current interval for $a$. Default $\text{cp1} = \text{cp2} = 10^{-3}$.
    - Otherwise: $\phi(a) \leq \phi(0) + a * \text{cp1} * \phi'(0)$ and $\phi'(a) \geq \text{cp2} * \phi'(0)$.
    Default $\text{cp1} = 10^{-3}$, $\text{cp2} = 0.99$.
  - **maxeval** Maximum number of function evaluations. Default $\text{maxeval} = 10$.
  - **amax** Maximal allowable $\alpha-$value. Default $\text{amax} = 10$.

If the input $\text{opts}$ has less than 5 elements, it is augmented by the default values. Also, negative elements and (except for $\text{choice}$) zeros are replaced by the default values.

**p1,p2,...** are passed directly to the function $\text{fun}$.

**Output parameters**

- **xn** New iterate, $x + \hat{\alpha}h$.
- **fn,gn** $f(xn)$ and $\nabla f(xn)$.
- **info** Performance information, vector with 3 elements:
  - info(1) > 0: Successful call. Value of $\hat{\alpha}$.
  - info(1) = 0: $h$ is not downhill or it is so large and $\text{maxeval}$ so small, that a better point was not found.
  - info(1) = −1: $x$ is not a real valued vector.
  - info(1) = −2: $f$ is not a real valued scalar.
  - info(1) = −3: $g$ or $h$ is not a real valued vector.
  - info(1) = −4: $g$ or $h$ has different length from $x$.
  - info(2) Slope ratio $\phi'(\hat{\alpha}) / \phi'(0)$.
  - info(3) Number of function evaluations used.
- **perf** Struct with fields
  - **alpha**: values of $\alpha$.
  - **phi**: values of $\phi(\alpha)$.
  - **slope**: values of $\phi'(\alpha)$.

1.1.3. User's guide to ucminf

This function is based on a Quasi-Newton method with BFGS updating of the approximate inverse Hessian, see eg [4, Section 3.5].

**Typical calls are**

- $[X, \text{info}] = \text{ucminf}(\text{fun}, x0)$
- $[X, \text{info}] = \text{ucminf}(\text{fun}, x0, \text{opts})$
- $[X, \text{info}] = \text{ucminf}(\text{fun}, x0, \text{opts}, D0, p1,p2,...)$
- $[X, \text{info}, \text{perf}] = \text{ucminf}(....)$
- $[X, \text{info}, \text{perf}, D] = \text{ucminf}(....)$
Input parameters
fun Handle to the function.
x0 Starting guess for \( \hat{x} \).

opts Either a struct with fields 'Delta', 'tolg', 'tolx' and 'maxeval', or a vector with the values of these options, \( \text{opts} = [\text{Delta} \ \text{tolg} \ \text{tolx} \ \text{maxeval}] \).

 Delta: Expected length of initial step.
The other options are used in the stopping criteria (3.1),
\[
\|\nabla f(x)\|_\infty \leq \text{tolg} \quad \text{or} \quad \|\delta x\|_2 \leq \text{tolx}(\text{tolx} + \|x\|_2) \quad \text{or} \quad \text{no. of function evaluations exceeds maxeval}
\]
Default \( \text{Delta} = 1, \ \text{tolg} = 1e^{-4}, \ \text{tolx} = 1e^{-8}, \ \text{maxeval} = 100 \). If the input \text{opts} has less than 4 elements, it is augmented by the default values. Also, zeros and negative elements are replaced by the default values.

D0 If present, then approximate inverse Hessian at \( x \). Otherwise, \( D0 := I \).
p1,p2,... are passed directly to the function \text{fun}.

Output parameters
X If \text{perf} is present, then \( X \) is an array, holding the iterates columnwise, with the computed solution in the last column.
Otherwise, \( X \) returns the computed solution vector.

info Performance information, vector with 6 elements:
 info(1:3) Final values of \([f(x), \|\nabla f(x)\|_\infty, \|\delta x\|_2]\).
 info(4:5) Number of iterations steps and evaluations of \( f \) and \( \nabla f \).
 info(6) =
 1: Stopped by a small gradient.
 2: Stopped by a small \( x \)-step.
 3: No. of function evaluations exceeds \text{opts}(4).
-1: \( x \) is not a real valued vector.
-2: \( f \) is not a real valued scalar.
-3: \( g \) is not a real valued vector.
-4: Dimension mismatch in \( x, g \).
-6: \( D0 \) is not a symmetric, positive definite \( n \times n \) matrix.

perf Struct with fields

f: values of \( f(x_k) \),
ng: values of \( \|\nabla f(x_k)\|_\infty \).

Delta: values of trust region radius.
am: values of \( \hat{\alpha} \) from line search,
slope: values of \( \varphi'(\hat{\alpha}) \) from line search,
neval: no. of function evaluations in current line search.

D Array holding the approximate inverse Hessian at the computed minimizer.

1.1.4. Example
Consider the function

function [f,g,H] = rosenbrock(x, p1,p2)
f1 = p1*(x(2) - x(1)^2); f2 = 1 - x(1); f = 0.5*(f1^2 + f2^2) + p2;
if nargout > 1
g = [-2*p1*x(1)*f1 - f2; p1*f1];
if nargout > 2

1.1. General Optimization

\begin{verbatim}
p12 = p1^2; h12 = -2*p12*x(1);
H = [1-2*p1*(f1 - 2*p1*x(1)^2) h12
     h12 p12];
end
end
\end{verbatim}

Choosing the parameters \( p_1 = 10 \) and \( p_2 = 0 \) this is an implementation of the famous Rosenbrock function. In the program

\begin{verbatim}
[X1, iii1, pf1] = dampnewton(@rosenbrock,[-1.2 1],opts,10,0);
[X2, ii2] = ucminf(@rosenbrock,[-1.2 1],[],[],10,0)
\end{verbatim}

we use the default values for \texttt{opts} and an empty \texttt{D0} in \texttt{ucminf}, and get the results,

\begin{verbatim}
X1(:,end) = 0.99998987 X2(:,end) = 1.00000032
0.99997934 1.00000063
ii1 = 5.95e-11 7.11e-05 1.38e-03 1.50e-06 21 31 1
ii2 = 5.25e-14 1.42e-06 9.32e-06 34 38 1
\end{verbatim}

In both cases we find a good approximation to the true minimizer, \( \hat{x} = [1 1] \), and \( \text{ii1}(7) = \text{ii2}(6) = 1 \) shows that iteration stops because the infinity norm of the gradient \( \text{ii}x(2) \) is smaller than the default threshold \( \text{tolg} = 1e-4 \). We see that \texttt{ucminf} used 38 evaluations of \( f \) and \( \nabla f \), while \texttt{dampnewton} used 31 evaluations of \( f \), \( \nabla f \) and the Hessian \( \nabla^2 f \).

The adjoining figure shows the two iteration paths. It was obtained by the command

\begin{verbatim}
plot(X1(:,1,:),X1(:,2,:),'-ob', X2(:,1,:),X2(:,2,:),'-xr')
\end{verbatim}

In \texttt{X2} every column 2,3,... is obtained by line search from the previous, and \( \text{ii}2(4:5) \) shows that in almost every iteration step we only need one function evaluation. In the damped Newton case \( \text{info}(6) = \text{info}(5)+1 \) would have indicated that all trial steps were successful. The result above shows that there were 9 uphill trial steps, where the iterand did not change, but the damping was increased. This can be seen from the ratios between dampings,

\[
r = \text{pf1.mu}(2:end) ./ \text{pf1.mu}(1:end-1)
\]

4 of these ratios are larger than 2, indicating at least one uphill trial.

To illustrate \texttt{linesearch} we give a program that uses soft and exact line search in the steepest descent direction from the same starting point as above. Again we use default values for \texttt{opts}.

\begin{verbatim}
x0 = [-1.2 1]; [f0, g0] = rosenbrock(x0,10,0)
optse = immoptset('linesearch', 'choice',0) % For exact line search
[x3, f3, g3, ii3] = linesearch(@rosenbrock,x0,f0,g0,-g0,[],10,0)
[x4, f4, g4, ii4] = linesearch(@rosenbrock,x0,f0,g0,-g0,optse,10,0)
\end{verbatim}

We get the following results,

\begin{verbatim}
x3 = -0.8761675820 x4 = -1.0219187641
1.1321764971 1.0726862187
ii3 = 3.00e-03 -6.11e-01 4 f3 = 8.4033
ii4 = 1.65e-03 -3.92e-02 6 f4 = 2.0843
\end{verbatim}

The step used with exact line search is about half the step used with soft line search, and we get smaller values for both \( f(x) \) and the ratio \( \varphi'(\hat{\alpha})/\varphi'(0) \), but the number of evaluations of \( f(x) \) and \( \nabla f(x) \) grows from 4 to 6.
1.2. Nonlinear Least Squares Problems

We seek a minimizer \( \hat{x} \) of the function

\[
f(x) = \frac{1}{2} \sum_{i=1}^{m} r_i(x)^2 = \frac{1}{2} r(x)^T r(x),
\]

where the \( r_i \) are given functions of \( x \in \mathbb{R}^n \) and \( r(x) \in \mathbb{R}^m \) is the vector function with \( r_i(x) \) as its \( i \)th component. A minimizer satisfies \( \nabla f(x) = 0 \), where the gradient is given by

\[
\nabla f(x) = J(x)^T r(x).
\]

\( J(x) \) is the Jacobian, as defined by

\[
J_{ij} = \frac{\partial r_i}{\partial x_j}.
\]

The toolbox functions \texttt{dogleg} and \texttt{marquardt} assume an analytic expression for the Jacobian. In those cases the user must supply a MATLAB function with a header of the form

\[
\text{function } [r, J] = \text{fun}(x,p1,p2,...)
\]

The function should return \( r(x) \) as a column vector in \( r \) and the \( m \times n \) Jacobian in \( J \). The toolbox functions allow a list of parameters \( p1,p2,... \). The implementation of \( J(x) \) can be checked by \texttt{checkgrad}, Section 3.2.2.

The function \texttt{smarquardt} only needs a user supplied function with the header of the form

\[
\text{function } r = \text{fun}(x,p1,p2,...)
\]

Also this function should return \( r(x) \) as a column vector in \( r \).

1.2.1. User's guide to \texttt{dogleg}

This function is based on Powell's dog-leg algorithm, as described eg in [4, Section 6.3].

Typical calls are

\[
[X,\text{info}] = \text{dogleg}(\text{fun},x0)
\]

\[
[X,\text{info}] = \text{dogleg}(\text{fun},x0,\text{opts},p1,p2,...)
\]

\[
[X,\text{info},\text{perf}] = \text{dogleg}(......)
\]

Input parameters

\texttt{fun} Handle to the function.
\texttt{x0} Starting guess for \( \hat{x} \).
\texttt{opts} Either a struct with fields 'Delta', 'tolg', 'tolx', 'tolr' and 'maxeval', or a vector with the values of these options.

\texttt{ opts = [Delta tolg tolx tolr maxeval]. }

\texttt{Delta} Initial trust region radius \( \Delta \).

The other options are used in an extended version of the stopping criteria (3.1),

\[
\|\nabla f(x)\|_\infty \leq \text{tolg} \quad \text{or}
\|
\delta x\|_2 \leq \text{tolx}(\text{tolx} + \|x\|_2) \quad \text{or}
\|r(x)\|_\infty \leq \text{tolr} \quad \text{or}
\]

no. of function evaluations exceeds \text{maxeval}.

Default \( \Delta = 0.1(1 + \|x_0\|_2) \), \( \text{tolg} = 10^{-4} \), \( \text{tolx} = 10^{-8} \), \( \text{tolr} = 10^{-6} \), \( \text{maxeval} = 100 \). If the input \texttt{opts} has less than 5 elements, it is augmented by the default values. Also, zeros and negative elements are replaced by the default values. \texttt{p1,p2,...} are passed directly to the function \texttt{fun}.
1.2. Nonlinear Least Squares

**Output parameters**

\[ X \]  
If `perf` is present, then \( X \) is an array, holding the iterates columnwise, with the computed solution in the last column. 
Otherwise, \( X \) returns the computed solution vector.

\[ \text{info} \]  
Performance information, vector with 7 elements:

- `info(1:4)`: Final values of \( f(x) \), \( \|\nabla f(x)\|_\infty \), \( \|\delta x\|_2 \), \( \Delta \).
- `info(5:6)`: No. of iteration steps and function evaluations.
- `info(7)`: 
  1: Stopped by a small gradient.
  2: Stopped by a small \( x \)-step,
  3: No. of function evaluations exceeds `maxeval`
  -1: \( x \) is not a real valued vector.
  -2: \( r \) is not a real valued column vector.
  -3: \( J \) is not a real valued matrix.
  -4: Dimension mismatch in \( x \), \( r \) and \( J \).
  -5: Overflow during computation.

\[ \text{perf} \]  
Struct with fields

- `f`: values of \( f(x_k) \),
- `ng`: values of \( \|\nabla f(x_k)\|_\infty \),
- `Delta`: values of trust region radius \( \Delta \).

**1.2.2. User’s guide to marquardt**

This function is based on Levenberg-Marquardt damping of the Gauss-Newton method, as described in [4, Section 6.2]. The updating of the damping parameter is further described in [10].

Typical calls are

\[ [X,\text{info}] = \text{marquardt}(\text{fun},x0) \]
\[ [X,\text{info}] = \text{marquardt}(\text{fun},x0,\text{opts},p1,p2,...) \]
\[ [X,\text{info},\text{perf}] = \text{marquardt}(......) \]

**Input parameters**

- `fun`: Handle to the function.
- `x0`: Starting guess for \( \hat{x} \).
- `opts`: Either a struct with fields `’tau’`, `’tolg’`, `’tolx’` and `’maxeval’`, or a vector with the values of these options,  
  \[ \text{opts} = [\tau \ \text{tolg} \ \text{tolx} \ \text{maxeval}] . \]
- `tau`: used in starting value for Marquardt parameter:
  \[ \mu = \tau \cdot \max\{(J(x_0)^T J(x_0))_{ii}\} . \]
  The other options are used in the stopping criteria (3.1),

\[ \|\nabla f(x)\|_\infty \leq \text{tolg} \quad \text{or} \]
\[ \|\delta x\|_2 \leq \text{tolx} (\text{tolx} + \|x\|_2) \quad \text{or} \]
  no. of function evaluations exceeds `maxeval`.

Default `tau = 10^{-3}`, `tolg = 10^{-4}`, `tolx = 10^{-8}`, `maxeval = 100`.

If the input `opts` has less than 4 elements, it is augmented by the default values.

Also, zeros and negative elements are replaced by the default values.

- `p1`, `p2`, ... are passed directly to the function `fun`.

**Output parameters**

\[ X \]  
If `perf` is present, then \( X \) is an array, holding the iterates columnwise, with the computed solution in the last column.
Otherwise, \( X \) returns the computed solution vector.
Performance information, vector with 7 elements:

\[ \begin{bmatrix} f(x), \| \nabla f(x) \|_\infty, \| \delta x \|_2, \frac{\mu}{\max \{|J(x)^T J(x)|_{ii}\}} \end{bmatrix}. \]

info(5:6) No. of iteration steps and function evaluations.
info(7) = 1: Stopped by a small gradient.
  2: Stopped by a small \( x \)-step,
  3: No. of function evaluations exceeds \texttt{maxeval}
-1: \( x \) is not a real valued vector.
-2: \( r \) is not a real valued column vector.
-3: \( J \) is not a real valued matrix.
-4: Dimension mismatch in \( x, r \) and \( J \).
-5: Overflow during computation.

\textbf{perf} Struct with fields
  \( f \) : values of \( f(x_k) \).
  \( \text{ng} \) : values of \( \| \nabla f(x_k) \|_\infty \).
  \( \mu \) : values of damping parameter \( \mu \).

\section*{1.2.3. User’s guide to \texttt{smarquardt}}

This function is based on the same algorithm as marquardt, but instead of an analytic expression for the Jacobian, this matrix function is approximated by successive updatings. See \cite[Section 6.4]{4}.

The initial approximation is either given as input or it is computed by forward differences,

\[ B_{-,j} = \frac{r(x + h_j e_j) - r(x)}{h_j}, \]

Where \( e_j \) is the unit vector in the \( j \)'th direction, and \( h_j = d^2 \) if \( x_j = 0 \), otherwise \( h_j = d \cdot |x_j| \). \( d \) is an input option, ‘\texttt{relstep}’.

Typical calls are

\[ [X, \text{info}] = \texttt{smarquardt}(\text{fun},x0) \]
\[ [X, \text{info}] = \texttt{smarquardt}(\text{fun},x0,\text{opts}) \]
\[ [X, \text{info}] = \texttt{smarquardt}(\text{fun},x0,\text{opts},B0,p1,p2,\ldots) \]
\[ [X, \text{info}, \text{perf}] = \texttt{smarquardt}(\ldots) \]
\[ [X, \text{info}, \text{perf}, B] = \texttt{smarquardt}(\ldots) \]

\section*{Input parameters}

\texttt{fun} Handle to the function that returns \( r(x) \).
\texttt{x0} Starting guess for \( \hat{x} \).
\texttt{opts} Either a struct with fields ‘\texttt{tau}’, ‘\texttt{tolg}’, ‘\texttt{tolx}’, ‘\texttt{maxeval}’ and ‘\texttt{relstep}’, or a vector with the values of these options,
\[ \text{opts} = [\text{tau} \ \text{tolg} \ \text{tolx} \ \text{maxeval} \ \text{relstep}] . \]
\texttt{tau} used in starting value for Marquardt parameter:
\[ \mu = \tau \cdot \max\{|B_0^T B_0|_{ii}\}, \]
where \( B_0 \) is an approximate Jacobian at \( x_0 \).
\texttt{relstep}, “relative” step length for difference approximations.
The other options are used in the stopping criteria, cf (3.1),
1.2. Nonlinear Least Squares

\[ \| B(x)^T r(x) \|_\infty \leq \text{tolg} \]
\[ \| \delta x \|_2 \leq \text{tolx} (\text{tolx} + \| x \|_2) \]

or

no. of function evaluations exceeds maxeval.

Default \( \tau = 10^{-3}, \quad \text{tolg} = 10^{-4}, \quad \text{tolx} = 10^{-8}, \quad \text{maxeval} = 100 + 10n, \quad \text{relstep} = 10^{-6}. \)

If the input \( \text{opts} \) has less than 5 elements, it is augmented by the default values.

Also, zeros and negative elements are replaced by the default values.

\( B_0 \) (Approximation to) \( J(x_0) \).

If \( B_0 \) is not given or is empty, a forward difference approximation to it is used.

\( p_1, p_2, \ldots \) are passed directly to the function \( \text{fun} \).

Output parameters

\( X \)

If \( \text{perf} \) is present, then \( X \) is an array, holding the iterates columnwise, with the computed solution in the last column. Otherwise, \( X \) returns the computed solution vector.

\( \text{info} \)

Performance information, vector with 7 elements:

**info(1:4)** Final values of

\[ f(x), \quad \| B(x)^T r(x) \|_\infty, \quad \| \delta x \|_2, \quad \frac{\mu}{\max\{(B(x)^T B(x))_{ii}\}} \].

**info(5)** Number of function evaluations.

**info(6)**

1: Stopped by a small gradient.
2: Stopped by a small \( x \)-step.
3: No. of function evaluations exceeds maxeval.
-1: \( x \) is not a real valued vector.
-2: \( r \) is not a real valued column vector.
-4: Dimension mismatch in \( x, r \) and \( J \).
-5: Overflow during computation.
-6: Error in approximate Jacobian.

**info(7)** = No. of iterations.

\( \text{perf} \)

Struct with fields

\( f \): values of \( f(x_k) \),

\( \text{ng} \): values of \( \| B(x)^T r(x) \|_\infty \),

\( \mu \): values of damping parameter \( \mu \).

\( B \)

Computed approximation to the Jacobian at the solution, \( J(\hat{x}) \).

1.2.4. Example

The following function is a “vector version” of the function from Example 1.1.4.

```matlab
function [r, J] = rosenbrockv(x, p1,p2)
if p2 > 0, r = [p1*(x(2) - x(1)^2); 1 - x(1); sqrt(2*p2)];
else, r = [p1*(x(2) - x(1)^2); 1 - x(1)]; end
if nargout > 1 % also the Jacobian
if p2 > 0, J = [-2*p1*x(1) p1; -1 0; 0 0];
else, J = [-2*p1*x(1) p1; -1 0]; end
end
end
```

In the program

```
[ x1 ii1 ] = dogleg(@rosenbrockv, [-1.2 1], [], 10,1e6)
[ x2 ii2 ] = marquardt(@rosenbrockv, [-1.2 1], [], 10,1e6)
[X3 ii3 pf3 B] = smarquardt(@rosenbrockv, [-1.2 1], [], [], 10,1e6);
```
we use the default values for opts and an empty B0 in smarquardt. We get the following results, where \( x_3 = X_3(:,\text{end}) \),
\[
\begin{align*}
x_1 &= 1 & x_2 &= 0.9999992008 & x_3 &= 0.9999998944 \\
& & 1 & 0.9999983957 & 0.999997841 \\
\end{align*}
\]
\
<table>
<thead>
<tr>
<th></th>
<th>1.00e+06</th>
<th>0</th>
<th>6.02e-03</th>
<th>2.16e-01</th>
<th>16</th>
<th>21</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>1.00e+06</td>
<td>5.87e-07</td>
<td>1.17e-04</td>
<td>2.55e-06</td>
<td>13</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>i2</td>
<td>1.00e+06</td>
<td>8.16e-07</td>
<td>2.20e-05</td>
<td>1.79e-06</td>
<td>18</td>
<td>50</td>
<td>1</td>
</tr>
</tbody>
</table>

In all the cases we find a good approximation to the true minimizer, \( \hat{x} = [1 1] \), and iiix(7) = 1 shows that iteration stops because the the infinity norm of the gradient iiix(2) is smaller than the default threshold \( \text{tolg} = 1\text{e}-4 \).
smarquardt uses almost the same number of iterations as marquardt, costing a total of 50 function evaluations, whereas marquardt uses 16 evaluations of \( r(x) \) and the Jacobian \( J(x) \).

The computed approximation to \( J(\hat{x}) \) and the true Jacobian at the minimizer are
\[
\begin{align*}
B &= \begin{bmatrix} -19.969 & 9.985 \\ -1 & -6.53e-17 \\ 0 & 0 \end{bmatrix} & J(xm) &= \begin{bmatrix} -20 & 10 \\ -1 & 0 \\ 0 & 0 \end{bmatrix}
\end{align*}
\]

Thus, also \( B \) is a good approximation to the Jacobian \( J(\hat{x}) \).

Finally,
\[
\begin{align*}
t &= 0 : 18; \\
\text{semilogy}(t, pf3.f - 1e6, 'ok', ... \\
& t, pf3.ng, 'vb', t, pf3.mu, 'pr')
\end{align*}
\]
illustrates the iteration with smarquardt. We subtract the final value in order to be able see the variation in the function values.

Both \( f \) and \( \nabla f \) show superlinear convergence in the final iterations, and \( \mu \) is divided by 3 in each of these steps.
2. Data Fitting

2.1. Cubic Splines

The toolbox contains a number of functions that can be used to determine a cubic spline, which either interpolates given points or is found as a weighted least squares fit to a set of given data points. In both cases it is possible to apply quite general boundary conditions. The theory is described eg in [1, Sections 5.9 – 5.12] and [4, Section 5.7].

The function `splinefit` is used to determine the spline \( s \) so that it fits to (or interpolates) given data points, possibly with constraints of the form

\[
\begin{align*}
    d_{11}s(a) + d_{12}s'(a) + d_{13}s''(a) &= d_{14} \\
    d_{21}s(b) + d_{22}s'(b) + d_{23}s''(b) &= d_{24}
\end{align*}
\]

(2.1)

where \([a, b]\) is the range of the spline knots.

`splineval` and `splinedif` can be used to evaluate the spline and derivatives of it, respectively.

The spline \( s \) is represented by a struct \( S \) with fields:

- `fail` Performance indicator. The other fields are only significant if \( S.fail = 0 \).
  - `fail = 0`: No problems
  - `1`: \( x \) is not a real valued vector of length at least 2.
  - `2`: Knots are not in increasing order.
  - `3`: Some data abscissa is not real or is outside \([x(1), x(end)]\).
  - `4`: No data ordinates are given.
  - `5`: Too few active data points (ie points with strictly positive weights).
  - `6`: The Schoenberg-Whitney conditions are not satisfied.
  - `7`: When given, the boundary condition matrix \( D \) must be a \( 2 \times 4 \) matrix.

- `x` Knots.
- `c` Coefficients in the B-spline representation of \( s \).
- `pp` Piecewise polynomial representation of \( s \).
- `sdy` Estimated standard deviation of data.
- `sdc` Estimated standard deviation of the coefficients \( c \).

### 2.1.1. User’s guide to `splinefit`

This function determines a cubic spline as a weighted least squares fit to given data points, possibly with given boundary conditions. The algorithm is described in [4, Section 5.7]. If the number of data points is equal to the number of degrees of freedom, one gets the interpolating spline.

Typical calls are

\[
S = \\text{splinefit}(tyw, x)\\nS = \\text{splinefit}(tyw, x, D)
\]
Input parameters

\texttt{tyw} Data points and weights. Array with 2 or 3 columns,
\texttt{tyw(:,1)} Data abscissas.
\texttt{tyw(:,2)} Data ordinates.
\texttt{tyw(:,3)} Weights.

If \texttt{tyw} holds less than 3 columns, then all weights are set to 1.

\texttt{x} Knots. Must satisfy \(x(1) < x(2) \leq \ldots \leq x(\text{end}-1) < x(\text{end})\).

\texttt{D} 2x4 array with the \(d_{ij}\) presented in (2.1). \(a = x(1), b = x(\text{end})\).

Output parameters

\texttt{S} Struct representing the spline, as described on page 13.

2.1.2. User’s guide to \texttt{splineval}

This function can be used to evaluate a cubic spline, computed by \texttt{splinefit}. The evaluation is done by means of the piecewise 3rd order polynomial provided in \texttt{S.pp}. For arguments outside the knot range we use the 2nd order polynomial obtained by continuation of the spline beyond its end knots.

Typical calls are
\[
\begin{align*}
  f &= \texttt{splineval}(S, t) \\
  f &= \texttt{splineval}(t, S)
\end{align*}
\]

The alternative formulation is introduced in order that \texttt{splineval} can be used in conjunction with \texttt{fminbnd}, \texttt{fzero}, \texttt{quad}, and other standard MATLAB function functions.

Input parameters

\texttt{S} Struct representing the spline, as described on page 13.
\texttt{t} Vector with arguments for the spline.

Output parameters

\texttt{f} Vector of the same type as \texttt{t}, with \(f_i = s(t_i)\).

2.1.3. User’s guide to \texttt{splinedif}

This function can be used to evaluate derivatives of a cubic spline, computed by \texttt{splinefit}. The evaluation is done by means of a differentiated version of the piecewise 3rd order polynomial provided in \texttt{S.pp}. Arguments outside the knot range are not allowed.

Typical calls are
\[
\begin{align*}
  f &= \texttt{splinedif}(S, t) \\
  f &= \texttt{splinedif}(S, t, d)
\end{align*}
\]

Input parameters

\texttt{S} Struct representing the spline, as described on page 13.
\texttt{t} Vector with arguments for the spline.

If any \texttt{t}(i) is outside the knot range, you get an error return.

\texttt{d} Differentiation order. Default \(d = 1\).

Output parameters

\texttt{f} Vector of the same type as \texttt{t}, with \(f_i = s^{(d)}(t_i)\).
2.1.4. Example

The toolbox data set \texttt{wild.dat} discussed on page 29 was generated by adding "noise" to a function defined on \([-1, 1]\) with slope approximately equal to 0.2 at both end points. This is used at the right hand end in the following MATLAB program, while we use natural spline condition at the other end. The 6 interior knots are distributed so that they are closest where the function varies most, cf [1, Section 5.11].

```matlab
load wild.dat
D = [0 0 1 0; 0 1 0 0.2]; x = [-1 -0.25 0 0.2 0.3 0.4 0.6 1];
S = splinefit(wild, x, D);
t = linspace(-1,1,201);
subplot(211), plot(wild(:,1),wild(:,2),'.b', t,splineval(S,t),'-r')
grid on, legend('Data', 'Fit, n = 7',2)
subplot(223), plot(t, splinedif(S,t),'-r')
grid on, legend('First derivative',3)
subplot(224)
plot(t,splinedif(S,t,2),'-r'), grid on
grid on, legend('Second derivative', 3)
```

The plot is shown below. Note how the position of the knots is seen clearly in the piecewise linear $s''$. 

![Plot](image_url)
2.2. Robust Estimation

The toolbox contains a number of functions that are meant for robust data fitting. They are based on Huber estimation, where "wild point" residuals only contribute with their absolute value, while "good point" residuals contribute as in a least squares fit. The threshold \( \gamma \) is used to distinguish between small and large residuals. The theory is described in [4, Section 7.3], and algorithmic details are given eg in [3], [5], [7], [9].

The figure shows the Huber function (full line) and the scaled least squares function, \( r^2/(2\gamma) \) (dashed line).

The MATLAB function \texttt{linhuber} finds the minimizer of a slightly extended version of the linear Huber estimation problem,

\[
f(x) = \sum_{i=1}^{m} \phi_\gamma(r_i(x)) + \frac{1}{2} \mu \| Lx \|_2^2 + c^T x
\]  

(2.2)

Here, \( r(x) = b - Ax \), where \( A \) is an \( m \times n \) matrix and \( b \) an \( m \)-vector, and the extra terms are optional.

In connection with \texttt{nonlinhuber} and \texttt{huberobj} the user must supply a MATLAB function with a header of the form

\[
\text{function } [r, J] = \text{fun}(x, p1,p2,...)
\]

(2.3)

(cf Sections 1.2.1 and 1.2.2 (\texttt{dogleg} and \texttt{marquardt})). The function should return \( r(x) \) as a column vector in \( r \) and the \( m \times n \) Jacobian in \( J \). We allow a list of parameters \( p1,p2,... \). The implementation of \( J(x) \) can be checked by \texttt{checkgrad}, Section 3.2.2.

2.2.1. User’s guide to \texttt{linhuber}

The function \( f(x) \) is a piecewise quadratic. The algorithm for finding a minimizer for it is outlined in [4, Section 7.3] and more details are given in [9].

Typical calls are

\[
[X, \text{info}] = \text{linhuber}(A,b,c,L,\text{par})
\]

\[
[X, \text{info}] = \text{linhuber}(A,b,c,L,\text{par},\text{init})
\]

\[
[X, \text{info}, \text{perf}] = \text{linhuber}(........)
\]

**Input parameters**

- \( A, b \) \( m \times n \) matrix and \( m \)-vector, respectively.
- \( c \) \( n \)-vector or empty. In the latter case the term \( c^T x \) is omitted.
- \( L \) Matrix with \( n \)-columns or empty. In the case \( \mu = \text{par}(2) > 0 \) an empty \( L \) is treated as \( L = I \), the identity matrix.
- \( \text{par} \) Vector with one, two or three elements.
  - \( \text{par}(1) \) \( \gamma \), Huber threshold.
  - \( \text{par}(2) \) \( \mu \); default: \( \mu = 0 \).
  - \( \text{par}(3) \) Choice of Huber function,
    - 1: one-sided, \( \rho(r) = 0 \) for \( r > 0 \).
    - 2: one-sided, all \( r_i \leq \gamma \) are active.
  - Otherwise: standard Huber (default), ie equations with \( |r_i| \leq \gamma \) are active.
2.2. Robust Estimation

Init Choice of starting point:
- init is a vector: Given $x_0$.
- init is a struct like info below: given active set and factorization.
- init is not present: $x_0$ is the least squares solution to $Ax \simeq b$.

Output parameters

X If perf is present, then X is an array, holding the iterates columnwise, with the computed solution in the last column.
Otherwise, X returns the computed solution vector.

info Struct with information of the performance and computed solution. Fields
- pf: Vector $[ f(\hat{x}), \|\nabla f(\hat{x})\|_\infty, \text{no. iterations, no. factorizations} ]$.
  - pf(1) = $-\infty$ indicates an unbounded problem.
- S: Struct with the Huber active set at the solution. Fields
  - s: Huber sign vector,
  - A: active set (indices of small residuals),
  - N: inactive set (indices of large residuals),
  - L: $[\text{length}(S.A) \text{ length}(S.N)]$.
- R: $n \times n$ matrix with Cholesky factor of active set.
- p: Permutation vector used in the factorization.

perf Iteration history. Struct with fields
- f: values of $f(x)$,
- ng: values of $\|\nabla f(x)\|_\infty$.
- nA: number of active equations.

2.2.2. User’s guide to nonlinhuber

This function is an implementation of [4, Algorithm 7.14]. Basically it is just the function marquardt with the objective function given by (2.2) instead of $f(x) = \frac{1}{2}\|r(x)\|_2^2$. With the default $\gamma$-value all the equations are active.

Typical calls are
- $[X, S, \text{info}] = \text{nonlinhuber}(\text{fun}, x0)$
- $[X, S, \text{info}] = \text{nonlinhuber}(\text{fun}, x0, \text{opts}, p1,p2,...)$
- $[X, S, \text{info}, \text{perf}] = \text{nonlinhuber}(....)$

Input parameters

- fun Handle to the function discussed at (2.3).
- x0 Starting guess for Huber estimator.
- opts Either a struct with fields 'gamma', 'Htype', 'tau', 'tolg', 'tolx' and 'maxeval', or a vector with the values of these options,
  - opts = [gamma Htype tau tolg tolx maxeval].
- gamma Huber threshold $\gamma$.
- Htype Choice of Huber function,
  - Htype = 1: One-sided, $r_i > 0$ are neglected,
  - 2: one-sided, all $r_i \leq \gamma$ are active,
  - otherwise: standard Huber, ie equations with $|r_i| \leq \gamma$ are active.
- tau used in starting value for Marquardt parameter:
  - $\mu = \tau \cdot \max\{(J(x_0)^TJ(x_0))_{ii}\}$.

The other options are used in the stopping criteria (3.1),
- $\|\nabla f(x)\|_\infty \leq \text{tolg}$ or
- $\|\delta x\|_2 \leq \text{tolx}(\text{tolx} + \|x\|_2)$ or
- no. of function evaluations exceeds maxeval.
Default \( \gamma = \|r(x_0)\|_{\infty} \) \( \text{Htype} = 0 \) \( \tau = 10^{-3} \) \( \text{tolg} = 10^{-4} \) \( \text{tolx} = 10^{-8} \) \( \text{maxeval} = 100 \)

If the input \( \text{opts} \) has less than 6 elements, it is augmented by the default values. Also, zeros and negative elements are replaced by the default values.

\( p_1, p_2, \ldots \) are passed directly to the function \( \text{fun} \).

**Output parameters**

\( X \)  
If \( \text{perf} \) is present, then \( X \) is an array, holding the iterates columnwise, with the computed solution in the last column.
Otherwise, \( X \) returns the computed solution vector.

\( S \)  
Struct with the Huber active set for the last col. in \( X \). Fields
\( s \): Huber sign vector,
\( A \): active set (indices of small components in \( r \)),
\( N \): vector \( [\text{length}(S.A), \text{length}(S.N)] \).

\( \text{info} \)  
Performance information, vector with 7 elements:
\( \text{info}(1:4) \) Final values of
\[
\begin{bmatrix}
  f(x), & \|\nabla f(x)\|_{\infty}, & \|\delta x\|_2, & \frac{\mu}{\max\{|J(x)^T J(x)|_{ii}\}}
\end{bmatrix}.
\]
\( \text{info}(5:6) \) No. of iteration steps and function evaluations.
\( \text{info}(7) = 1 \): Stopped by a small gradient.
\( 2 \): Stopped by a small \( x \)-step,
\( 3 \): No. of function evaluations exceeds \( \text{maxeval} \).

\( \text{perf} \)  
Struct with fields
\( f \): values of \( f(x_k) \),
\( \text{ng} \): values of \( \|\nabla f(x_k)\|_{\infty} \),
\( \mu \): values of damping parameter \( \mu \).

**2.2.3. User’s guide to huberobj**

This function can be used to compute values of \( f \) and its gradient, where \( f(x) \) is the simple version of ((2.2)),
\[
f(x) = \sum_{i=1}^{m} \phi(r_i(x)).
\]  
(2.4)

\( \phi \) is the Huber function, and \( r \) is a vector function.

Typical calls are
\( [f, S, r] = \text{huberobj}(\text{fun}, x, \gamma) \)
\( [f, S, r] = \text{huberobj}(\text{fun}, x, \gamma, \text{Htype}, p_1, p_2, \ldots) \)
\( [f, S, r, J, g] = \text{huberobj}(\ldots) \)

**Input parameters**

\( \text{fun} \)  
Handle to the function discussed at (2.3).

\( x \)  
\( n \)-vector. Argument.

\( \gamma \)  
Huber threshold.

\( \text{Htype} \)  
Choice of Huber function,
\( 1 \): one-sided, \( \rho(r_i) = 0 \) for \( r_i > 0 \).
\( 2 \): one-sided, all \( r_i \leq \gamma \) are active.
Otherwise: standard Huber (default), ie equations with \( |r_i| \leq \gamma \) are active.

\( p_1, p_2, \ldots \) are passed directly to the function \( \text{fun} \).
2.2. Robust Estimation

Output parameters

- **f**: Huber objective function.
- **S**: Struct with the Huber active set at the solution. Fields
  - **s**: Huber sign vector,
  - **A**: active set (indices of small elements in \( r \)),
  - **N**: inactive set (indices of large elements in \( r \)),
  - **L**: \([\text{length}(S.A) \text{ length}(S.N)]\).
- **r,J**: Output from the evaluation of \( \text{fun} \).
  - If \( \text{nargout}<4 \), then \( \text{fun} \) only needs to return \( r(x) \), and the gradient is not computed.
- **g**: Gradient, \( g = \nabla f(x) \).

2.2.4. Example

The data set `efit2.dat` provided in the toolbox can be modelled by the function

\[
M(x,t) = x_1 e^{-10t} + x_2 e^{-5t} + x_3 .
\]

The following program computes the Huber estimator for \( \gamma = 0.015 \).

```matlab
ty = load('efit2.dat'); m = size(ty,1);
A = [exp(ty(:,1)*[-10 -5]) ones(m,1)];
[x info] = linhuber(A, ty(:,2), [], [], 0.015)
S = info.S
```

The results are

\[
x = -0.99897 \quad \text{info = pf: } [0.28914 \ 0 \ 2 \ 2] \quad \text{S = s: } [1 \times 40 \ \text{double}]
\]

\[
0.99901 \quad S: [1 \times 1 \ \text{struct}] \quad A: [1 \times 37 \ \text{double}]
\]

\[
0.20001 \quad R: [3 \times 3 \ \text{double}] \quad N: [7 \ 22 \ 37]
\]

\[
p: [3 \ 2 \ 1] \quad L: [37 \ 3]
\]

Thus, the solution is found after two iteration steps and there are three "wild points". They are marked by stars in the figure below, and the dotted lines indicate that their influence on the fit is as if they were on the boundary of the shaded region between \( M(x,\gamma,t) - \gamma \) and \( M(x,\gamma,t) + \gamma \).

![Graph showing the fit with wild points marked]

Next, consider the above fitting model with unknown exponents

\[
M(x,t) = x_3 e^{-x_1t} + x_4 e^{-x_2t} + x_5 .
\]

The corresponding \textit{fun} for use in `nonlinhuber` and `huberobj` may eg have the form
function  [r, J] = myfun2(x, ty)
x = x(:);  t = ty(:,1);  m = length(t);
E = exp(t*(-x(1:2)'));  F = [E  ones(m,1)];
r = ty(:,2) - F*x(3:5);
if  nargout > 1  % Jacobian
    J = [(t * x(3:4)').*E -F];
end

With the data in efit2 we want to find the Huber estimator for $\gamma = 0.015$ and $\gamma = 10^{-5}$:

```matlab
ty = load('efit2.dat');  x0 = [10 5 -1 1 0.2];
[x1 S1 info1] = nonlinhuber(@myfun2, x0, 1.5e-2, ty)
[x2 S2 info2] = nonlinhuber(@myfun2, x0, 1e-5, ty)
```

the results are

```matlab
x1' = 10.15384  4.89230  -0.95188  0.95307  0.19885
x2' = 9.86634  5.06794  -1.01994  1.02529  0.20106
```

S1 = s: [1x40 double]  S2 = s: [1x40 double]
A: [1x37 double]  A: [4 8 11 18 33]
N: [7 22 37]  N: [1x35 double]
L: [37 3]  L: [5 35]

```
info1 = 0.289  1.05e-05  8.70e-04  2.61e-07  9  9  1
info2 = 0.451  2.86e-06  2.45e-05  1.11e-06  12  18  1
```

Notice that we only provide $\gamma$, the first element in opts. The other 5 options are assigned their default values, and in both cases iterations are stopped by a small gradient. The solution with $\gamma = 0.015$ has the same active set as the solution found with linhuber, while the active set with $\gamma = 10^{-5}$ only has 5 elements.

Finally, in an attempt to see what happens as $\gamma \to 0$ we might try $\gamma = 10^{-15}$:

```matlab
[x3 S3 info3] = nonlinhuber(@myfun2, x0, 1e-15, ty)
```

This, however, gives an error return

```matlab
??? Error using ==> huberobj at 46
    gamma must be at least 6.12e-14
```

Error in ==> nonlinhuber at 68
    [f S r J g] = huberobj(fun,x,gamma,Htype,varargin:);  neval = 1;
```

The background for this message is that the effect of rounding errors on the elements in $r(x)$ may be larger than this small value of $\gamma$, with the meaningless effect, eg, that the active set may be considered as empty.
2.3. Multiexponential Fitting

A fitting model of the form

\[ M_0(x, t) = \sum_{j=1}^{p} c_j e^{-z_j t} \quad \text{or} \quad M_1(x, t) = \sum_{j=1}^{p} c_j e^{-z_j t} + c_{p+1}, \quad x = \begin{pmatrix} z \\ c \end{pmatrix}, \quad (2.5) \]

occurs so frequently in practice that it has its own name, multiexponential model, and deserves a special MATLAB function, which exploits that about half the parameters (the coefficients \( c \)) occur linearly. The special-purpose method is known as separable least squares, cf eg [4, Example 6.20], [2], [11], and is implemented in \texttt{mexpfit}.

2.3.1. User’s guide to \texttt{mexpfit}

This function is based on the algorithm described in [11]. With given data \( \{(t_i, y_i)\}_{i=1}^{m} \) the model (2.5) corresponds to

\[ F(z)c(z) \approx y, \]

where \( c(z) \) is the (possibly weighted) least squares solution to this overdetermined system of equations. We use the Levenberg–Marquardt algorithm with the objective function (1.1) changed expressed as

\[ f(z) = \frac{1}{2} \|y - F(z)c(z)\|_2^2. \]

Typical calls are

\[
\begin{align*}
&[Z, c, \text{info}] = \text{mexpfit}(\text{tyw}, z0) \\
&[Z, c, \text{info}] = \text{mexpfit}(\text{tyw}, z0, \text{opts}) \\
&[Z, c, \text{info}, \text{perf}] = \text{mexpfit}(\ldots)
\end{align*}
\]

Input parameters

\begin{itemize}
  \item \texttt{tyw} Data points and weights. Array with 2 or 3 columns, \texttt{tyw(:,1:2)} Abscissas and ordinates of data points. \texttt{tyw(:,3)} Weights. If \texttt{tyw} has less than 3 columns, then all weights are set to 1. \texttt{z0} Starting guess for \( z \).
  \item \texttt{opts} Either a struct with fields \texttt{’const’, ’tau’, ’tolg’, ’tolx’, and ’maxeval’}, or a vector with the values of these options, \texttt{opts} = \[ \texttt{[const tau tolg tolx maxeval]} \]. \texttt{const} If positive then there is a constant term. \texttt{tau} Used in starting value for Marquardt parameter for the optimization, \( \mu = \tau \ast \max\{ |J(z_0)^T J(z_0)|_{ii} \} \). The other options are used in stopping criteria:
    \[ ||\nabla f(z)||_\infty \leq \text{tolg} \]
    \[ ||\delta z||_2 \leq \text{tolx} (\text{tolx} + ||z||_2) \]
    no. of function evaluations exceeds \texttt{maxeval}
  \item Default \texttt{const} = 0, \texttt{tau} = 10^{-3}, \texttt{tolg} = 10^{-6}, \texttt{tolx} = 10^{-10}, \texttt{maxeval} = 100.
\end{itemize}

If the input \texttt{opts} has less than 5 elements, it is augmented by the default values. Also, zeros and negative elements are replaced by the default values.

Output parameters

\begin{itemize}
  \item \texttt{Z} If \texttt{perf} is present, then \texttt{Z} is an array, holding the iterates columnwise, with the computed solution in the last column. Otherwise, \texttt{Z} returns the computed solution vector.
\end{itemize}
If `info.fail = 0`, then `c` holds coefficients `[c_1, ..., c_q]`, where
\[ \text{opts.const} > 0: \quad q = p+1; \quad c_q \text{ is the constant term}, \]
\[ \text{otherwise:} \quad q = p. \]

**info**: Struct with information on performance. Fields
- `fail`: Successful run if `fail = 0`. Otherwise `info.msg` tells what went wrong.
- `msg`: Text message about performance.
- `vals`: Vector with final values of \( f(z) \), \( \| \nabla f(z) \|_\infty \), \( \| \delta z \|_2 \).
- `its`: Vector with number of iterations and function evaluations.

**perf**: Struct with fields
- `f`: values of \( f(z_k) \),
- `ng`: values of \( \| \nabla f(z_k) \|_\infty \),
- `mu`: values of damping parameter \( \mu \).

### 2.3.2. Example

As in Example 2.2.4 we consider the data set `efit2.dat`, for which we know that with a proper choice of \( z_{1:2} \) and \( c_{1:3} \) the model
\[
M(x,t) = c_1 e^{-z_1 t} + c_2 e^{-z_2 t} + c_3
\]
is a good approximation to the background function. In Example 2.2.4 we saw that data point nos. 3, 22 and 37 are “wild”, i.e., they have exceptionally large errors, and we shall neglect them in this example. Finally, we take the poor starting guess \( z_0 = [5 \ 3] \), and except for `const = 1` we use the default `opts`-values.

```matlab
ty = load('efit2.dat');
w = ones(size(ty(:,1))); w([7 22 37]) = 0;
[z c info] = mexpfit([ty w], [5 3], 1)
```

The results are
```
z = 10.013  c = -0.94005
4.8556      0.94466
          0.19990
```
```
info = fail: 0
msg: 'Iterations stopped by a small gradient'
vals: [8.42e-04  9.80e-07  1.28e-01]
its: [5  6]
```

Thus, 5 iterations give us almost the same solution as in Example 2.2.4. We have changed the objective function a little, and cannot expect better agreement.
3. Miscellaneous

3.1. Nonlinear Systems of Equations

We seek a root \( \hat{x} \) of a vector function \( r \), as given by a MATLAB function with a header of the form

\[
\text{function } r = \text{fun}(x,p1,p2,\ldots)
\]

The function should return \( r(x) \) as a column vector in \( r \).

(\( p1,p2,\ldots \) are possible parameters of \( r \)).

\text{nonlinsys} can be used to solve such a problem. If the Jacobian of the system is available, we recommend to use \text{dogleg} instead.

3.1.1. User’s guide to \text{nonlinsys}

This function is based on Powell’s dog-leg algorithm, as described eg in [4, Section 6.5]. A root \( \hat{x} \) of the vector function \( r(x) \) is a minimizer of

\[
\phi(x) = \frac{1}{2}r(x)^T r(x)
\]

The gradient of \( \phi \) and the Newton step for \( r \) are given by respectively

\[
\nabla \phi = Jr \quad \text{and} \quad h_{GN} = -J^{-1}r.
\]

Approximations to both the Jacobian and its inverse are successively updated.

Typical calls are

\[
[X, \text{info}] = \text{nonlinsys}(\text{fun}, x0)
\]

\[
[X, \text{info}] = \text{nonlinsys}(\text{fun}, x0, \text{opts})
\]

\[
[X, \text{info}] = \text{nonlinsys}(\text{fun}, x0, \text{opts}, B0, p1,p2,\ldots)
\]

\[
[X, \text{info}, \text{perf}] = \text{nonlinsys}(\ldots)
\]

Input parameters

\text{fun} Handle to the function.
\text{x0} Starting guess for \( \hat{x} \).
\text{opts} Vector with five elements,

\text{opts}(1) initial trust region radius \( \Delta_0 \).
\text{opts}(2:4) used in stopping criteria:

\[
\|\nabla r(x)\|_\infty \leq \text{opts}(2) \quad \text{or} \quad \|\delta x\|_2 \leq \text{opts}(3) (\text{opts}(3) + \|x\|_2) \quad \text{or}
\]

no. of function evaluations exceeds \text{opts}(4) \( <= \) update!

\text{opts}(5) ”relative” step length for difference approximations.

Default \text{opts} = \([0.1(1 + \|x_0\|), 1e-6, 1e-8, 100, 1e-6]\)
If the input \texttt{opts} has less than 5 elements, it is augmented by the default values.
Also, non-positive elements are replaced by the default values.
Initial approximation to the Jacobian of $J_r(x_0)$. If \texttt{B0} is not given, a forward
difference approximation to $J_r(x_0)$ is used.

\texttt{p1,p2,...} are passed directly to the function \texttt{fun}.

\textbf{Output parameters}

\textbf{X} If \texttt{perf} is present, then \textbf{X} is an array, holding the iterates columnwise, with the
computed solution in the last column.
Otherwise, \textbf{X} returns the computed solution vector.

\textbf{info} Performance information, vector with 6 elements:
\begin{itemize}
  \item \texttt{info(1:3)} Final values of $\|r(x)\|_\infty$, $\|\nabla r(x)\|_\infty$, $\|\delta x\|_2$, $\Delta$.
  \item \texttt{info(4:5)} Number of iteration steps and function evaluations.
  \item \texttt{info(6)} =
    \begin{itemize}
      \item 1: Stopped by small $r$-vector.
      \item 2: Stopped by a small $x$-step,
      \item 3: No. of function evaluations exceeds \texttt{opts(4)} $<=$ update!
      \item -1: $x$ is not a real valued vector.
      \item -2: $r$ is not a real valued column vector.
      \item -3: Dimension mismatch in $x$, $r$, \texttt{B0}.
      \item -4: Maybe started at a saddle point.
      \item -5: Overflow during computation.
    \end{itemize}

\textbf{perf} Array, holding
\begin{itemize}
  \item \texttt{perf(1,:)} values of $\|r(x)\|_\infty$ ,
  \item \texttt{perf(2,:)} values of $\Delta$ .
\end{itemize}

\textbf{3.1.2. Example}

The following \texttt{MATLAB} function implements the example from [1, Section 4.8]. in \texttt{B1}.

\begin{verbatim}
function r = rr(x)
  r = [4*x(1)^2 + 9*x(2)^2 - 36 ; 16*x(1)^2 - 9*x(2)^2 - 36];
\end{verbatim}

In the call

\begin{verbatim}
[x ii] = nonlinsys(@rr, [1 1])
\end{verbatim}

we use the starting point $x_0 = (1, 1)$, default values for \texttt{opts} and an empty \texttt{B0}. We get the
following results,
\begin{verbatim}
x =  1.8974
     1.5492

ii = 3.9703e-07  1.8401e-05  2.2481    7   14   1
\end{verbatim}

Thus, we find an approximate solution (with $\|r(x)\|_\infty = 3.97 \cdot 10^{-7}$) after 7 iteration steps,
involving 14 evaluations of $r(x)$.

For comparison, if we extend \texttt{rr} so that it also returns the Jacobian, and use \texttt{dogleg} with the
same starting point and default \texttt{opts}-values, we get the same solution (to 8 digits accuracy),
but $\|r(x)\|_\infty = 1.14 \cdot 10^{-13}$. This costs 7 evaluations of $r(x)$ \textbf{and} the Jacobian $J(x)$. 
3.2. Auxiliary Programs

The function `immoptset` helps the user setting the options that are required by a number of the `immoptibox` functions, and `checkgrad` can be used to check the user’s implementation of a gradient or Jacobian or Hessian.

3.2.1. User’s guide to `immoptset`

The ideas behind this function are due to PhD student Carsten Völcker, DTU Informatics.

The following table gives the 9 functions that are handled by `immoptset` and the associated option names.

<table>
<thead>
<tr>
<th>Function</th>
<th>Option 1</th>
<th>Option 2</th>
<th>Option 3</th>
<th>Option 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>dampnewton</td>
<td>'tau'</td>
<td>'tolg'</td>
<td>'tolx'</td>
<td>'maxeval'</td>
</tr>
<tr>
<td>linesearch</td>
<td>'choice'</td>
<td>'cp1'</td>
<td>'cp2'</td>
<td>'maxeval'</td>
</tr>
<tr>
<td>ucmindf</td>
<td>'Delta'</td>
<td>'tolg'</td>
<td>'tolx'</td>
<td>'maxeval'</td>
</tr>
<tr>
<td>dogleg</td>
<td>'Delta'</td>
<td>'tolg'</td>
<td>'tolx'</td>
<td>'tolr'</td>
</tr>
<tr>
<td>marquardt</td>
<td>'tau'</td>
<td>'tolg'</td>
<td>'tolx'</td>
<td>'maxeval'</td>
</tr>
<tr>
<td>smarquardt</td>
<td>'tau'</td>
<td>'tolg'</td>
<td>'tolx'</td>
<td>'maxeval'</td>
</tr>
<tr>
<td>nonlinhuber</td>
<td>'gamma'</td>
<td>'Htype'</td>
<td>'tau'</td>
<td>'tolg'</td>
</tr>
<tr>
<td>mexpfit</td>
<td>'const'</td>
<td>'tau'</td>
<td>'tolg'</td>
<td>'tolx'</td>
</tr>
<tr>
<td>nonlinsys</td>
<td>'Delta'</td>
<td>'tolg'</td>
<td>'tolx'</td>
<td>'maxeval'</td>
</tr>
</tbody>
</table>

Most cases include the option names 'tolg', 'tolx' and 'maxeval', which correspond to the stopping criteria

$$
\| \nabla f(x) \|_\infty \leq \text{tolg} \quad \text{or} \quad \| \delta x \|_2 \leq \text{tolx}(\text{tolx} + \| x \|_2) \quad \text{or} \quad \text{no. of function evaluations exceeds maxeval} \tag{3.1}
$$

The other options are explained in connection with the function.

A typical call is

```matlab
opts = immoptset(mlfun, p1,v1, p2,v2, ...)
```

**Input parameters**

- `mlfun` String with the name of the function or a handle to it.
- `p1, p2, ...` Strings with option names
- `v1, v2, ...` Real numbers with the values of the options.

**Output parameters**

- `opts` Struct with the option names and their values. Options that do not appear as input are assigned their default value.
  The values are not checked for feasibility; this is done in the function defined by `mlfun`. 
3.2.2. User’s guide to checkgrad

The user’s implementation of partial derivatives are controlled by approximating forward and backward differences and extrapolations of these, \( D_F, D_B, \) and \( D_E \), respectively. Let \( p \) denote the value of the partial derivative and \( u = p - D \). The implementation is probably correct if

\[
u_B \simeq -\frac{1}{2} u_F
\]

and \( u_E \) is orders of magnitude smaller. For details see [6] or the example in [1, pp 156 – 158].

A typical call is

\[
\text{[maxJ, err, index] = checkgrad(fun, x, h, p1,p2,...)}
\]

Input parameters

- **fun** Handle to the function.
- **x** The point where we wish to check the derivatives.
- **h** Step length used in difference approximations.
  May be a vector with \( h(j) \) used in the \( j \)’th coordinate direction.
  \( h_j \simeq 10^{-6} |x_j| \) is generally a good choice.
- **p1, p2, ...** are passed directly to the function \( \text{fun} \).

Output parameters

- **maxJ** Largest element in the vector (or matrix) of partial derivatives.
- **err** Vector with three elements. The maximal absolute value of \( u_F, u_B \) and \( u_E \), respectively.
- **index** \( 3 \times 2 \) array, with \( \text{index}(k,:) \) giving the position in matrix of partial derivatives, where \( \text{err}(k) \) occurs.

3.2.3. Example

The calls

\[
\text{opts1 = immoptset(@marquardt) and opts = immoptset('marquardt')}
\]

both give the default values

\[
\text{opts = tau: 0.001}
\]

\[
\text{tolg: 0.0001}
\]

\[
\text{tolx: 1e-008}
\]

\[
\text{maxeval: 100}
\]

\[
\text{opts2 = immoptset(@Marquardt, 'Tau',1e-6) gives opts2 = tau: 1e-006}
\]

\[
\text{tolg: 0.0001}
\]

\[
\text{tolx: 1e-008}
\]

\[
\text{maxeval: 100}
\]

Notice that upper/lower case is ignored.

Finally, \( \text{opts3 = immoptset(@marquart, 1e-6,'tau')} \) gives

\[
??? Error using ==> immoptset at 48
\]

\[
\text{IMMOPTSET is not prepared for marquart}
\]

and \( \text{opts4 = immoptset(@marquardt, 1e-6,'tau')} \) gives

\[
??? Error using ==> immoptset at 56
\]

The value 1e-06 is not assigned to an option.
In order to demonstrate `checkgrad` consider the function

```matlab
function [f, g] = myfun(x)
e = exp(-x(1)^2); f = e*cos(x(2));
if nargout > 1
    g = [-2*x(1)*f; -e*sin(x(2))];
end
```

The call

```
[maxJ, err, index] = checkgrad(@myfun, [1 2], 1e-6)
```

Gives

```
maxJ = 0.33451   err = -1.5308e-007   index = 1 1
         7.6485e-008   1 1
         -3.7605e-011  1 1
```

The values in `err` behave as discussed above. The implementation of the gradient seems to be correct.

Now suppose that the gradient had been implemented as

```
g = [-2*x(1)*f; e*sin(x(2))];
```

Then the same call would give

```
maxJ = 0.33451   err = -0.66902   index = 1 2
            -0.66902  1 2
            -0.66902  1 2
```

The three values in `err` are identical. This indicates an error, and the second element in `index` shows that this is in the derivative with respect to \( x_2 \).

### 3.3. Test Problems

This section contains a problem generator `uctpget` which gives the choice between 22 small and medium sized unconstrained optimization problems. The function `uctpval` is constructed so that it can be used as `fun` in the MATLAB functions of sections 1.1 and 1.2.

#### 3.3.1. User’s guide to `uctpget`

This function can be used to define a test problem. There is a choice between 22 problems, as described in [13]. With some of the problems it is possible to vary the size of the problem.
<table>
<thead>
<tr>
<th>pno</th>
<th>Name</th>
<th>( m )</th>
<th>( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Linear function, full rank</td>
<td>( m \geq n )</td>
<td>variable</td>
</tr>
<tr>
<td>2</td>
<td>Linear function, rank 1</td>
<td>( m \geq n )</td>
<td>variable</td>
</tr>
<tr>
<td>3</td>
<td>Linear function, rank 1. Zero columns and rows</td>
<td>( m \geq n )</td>
<td>variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( n \geq 3 )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Rosenbrock</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>Helical Valley</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>Powell singular function</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>Freudenstein and Roth</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>Bard</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>Kowalik and Osborne</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>Meyer</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>Watson</td>
<td>31</td>
<td>( 2 \leq n \leq 31 )</td>
</tr>
<tr>
<td>12</td>
<td>Box 3-dimensional</td>
<td>( m \geq 3 )</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>Jennrich and Sampson</td>
<td>( m \geq 2 )</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>Brown and Dennis</td>
<td>( m \geq 4 )</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>Chebyquad</td>
<td>( m \geq n )</td>
<td>variable</td>
</tr>
<tr>
<td>16</td>
<td>Brown almost linear</td>
<td>( m = n )</td>
<td>variable</td>
</tr>
<tr>
<td>17</td>
<td>Osborne 1</td>
<td>33</td>
<td>5</td>
</tr>
<tr>
<td>18</td>
<td>Exponential fit</td>
<td>45</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>Exponential fit, separated</td>
<td>45</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>Modified Meyer</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>Separated Meyer</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>Exp and squares</td>
<td>1</td>
<td>variable</td>
</tr>
</tbody>
</table>

A typical call is 
\[
[\text{par}, \text{x}_0, \text{tau}_0, \text{delta}_0] = \text{uctpget}(\text{pno}, m, n)
\]

**Input parameters**

- **pno** Problem number. Integer in the range \([1, 22]\).
- **m**, **n** Number of components in the vectors \( r(x) \) when \( \text{pno} \leq 21 \) and \( x \), respectively. Not variable in all problems.

**Output parameters**

- **par** Struct defining the problem.
  - **par.p** Problem number.
  - **par.pt** = 0 signifies a least squares problem, otherwise a general problem.
  - **par.xm** Solution. (NaNs if \( m \) or \( n \) is free).
    For some problems \( \text{par} \) has more fields.
- **x0** Standard starting point.
- **tau0** Standard value for ‘\( \text{tau} \)’ in the Marquardt-type functions of the toolbox.
- **delta0** Standard value for ‘\( \text{Delta} \)’ in the DogLeg-type functions of the toolbox.

### 3.3.2. User’s guide to uctpval

This function can be used to evaluate a test problem, as defined by \( \text{uctpget} \). If the problem is “born” as a least squares problem, and \( \text{par}.pt \) is changed to 1 before the call, then \( \text{uctpval} \) returns the function value \( f(\text{x}) \) and the gradient \( \nabla f(\text{x}) \) as defined by

\[
f(\text{x}) = \frac{1}{2} \sum_{i=1}^{m} r_i(\text{x})^2 = r(\text{x})^T r(\text{x}), \quad \nabla f(\text{x}) = J(\text{x})^T r(\text{x}).
\]
3.3. Test Problems

Typical calls are
\[
\begin{align*}
&f = \text{uctpval}(x, \text{par})
\text{[f, Df]} = \text{uctpval}(x, \text{par})
\text{[f, Df, D2f]} = \text{uctpval}(x, \text{par})
\end{align*}
\]

**Input parameters**

- \(x\): Argument vector \(x\).
- \(\text{par}\): Struct defining the problem, cf uctpget.

**Output parameters**

- \(f\): If \(\text{par.p} \leq 21\) and \(\text{par.pt} = 0\), then \(f\) holds the vector \(r(x)\), otherwise \(f\) holds the scalar \(f(x)\).
- \(\text{Df}\): If \(\text{par.p} \leq 21\) and \(\text{par.pt} = 0\), then \(\text{Df}\) holds the Jacobian \(J(x)\), otherwise \(\text{Df}\) holds the gradient \(\nabla f(x)\).
- \(\text{D2F}\): Presumes \(\text{par.pt} \neq 0\). Hessian matrix \(\nabla^2 f(x)\).

### 3.3.3. Example

Suppose that we want to compare marquardt and \texttt{ucminf} applied to Rosenbrock’s problem with the starting point \(x_0 = (-1.2, 1)\) and demand that the gradient be smaller than \(10^{-6}\). This can be done as follows.

```plaintext
[par x0 tau] = uctpget(4,2,2);  
[xm infom] = marquardt(@uctpval, x0, [tau 1e-6 1e-12 100], par);  
par.pt = 1;  
[xu infou] = ucminf(@uctpval,x0,[1 1e-6 1e-12 100],[],par);
```

Both \(xm\) and \(xu\) are close to the solution \(\hat{x} = (1, 1)\), and

\[
\begin{align*}
\text{infom} &= 4.91\text{e-16} \quad 1.43\text{e-08} \quad 9.47\text{e-06} \quad 1.23\text{e-06} \quad 23 \quad 1 \\
\text{infou} &= 1.45\text{e-17} \quad 6.3\text{e-08} \quad 6.99\text{e-07} \quad 35 \quad 39 \quad 1
\end{align*}
\]

show that in both cases the desired accuracy was obtained after respectively 24 evaluations of \(r\) and \(J\) and 39 evaluations of \(f\) and \(\nabla f\).

### 3.3.4. Data sets

The toolbox contains a number of data sets, which can be used to check a data fitting algorithm.

- **optic.dat**
- **wild.dat**
The command `load optic.dat` gives an $m \times 2$ array `optic`, in which the first column holds values of the independent variable, and the second column holds the corresponding values of the dependent variable. Similar for the other data sets.

The data in `osl.dat` (notice the semilogarithmic scale) are an example of the fitting problem discussed in [4, Examples 5.14 and 6.22].

Also see the standard MATLAB function `titanium`.
4. History
Changes since previous version

Version 2.2. November 2010
• Default $\gamma$ in nonlinhuber has been changed from $\infty$ to $\|r(x_0)\|_\infty$.

Version 2.1. November 2010
• The “homepage” has been changed from html to the present pdf format.
• Notation for objective function, gradient and Hessian has been changed to agree with the textbook K. Madsen & H.B. Nielsen, 2010.
• Options in a number of functions has been changed from vector format to the choice between vector and struct format.
• The function nlshybrid has been removed from the toolbox.
• The functions nonlinhuber, mexpfit and immoptset and the data set osl.dat have been added to the toolbox.

Version 1.7. July 2009
• Corrected some broken links in the references.

• Corrected a bug in ucminf connected with the starting point satisfying the stopping criteria.

Version 1.5. September 2006
• Corrected a bug in linesearch so that it also finds a solution if the starting guess is beyond a local maximizer in the search direction.
• Corrected a bug in ucminf connected with stopping with a zeros step.

Version 1.4. November 2005
• Corrected a bug in ucminf that made it fail if the starting guess was an acceptable approximation to a local minimizer.

Version 1.3. October 2005
• Minor changes in the help parts of linhuber and ucminf.

Version 1.2. September 2004
• Section 3.2. Robust estimation has been added.
• Two data sets for exponential fitting have been added.
• dampnewton has been modified so that it can handle a zero initial Hessian.
• dogleg has been modified so that when perf is present the matrix X returns the trial sites instead of the currently best sites.

Version 1.1. August 2004
5. References


