DTU Informatics
Department of Informatics and Mathematical Modelling

DTU

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

2. Data fitting

3. Miscellaneous

1. Unconstrained Optimization

1.1. General Optimization

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

The function `dampnewton` calls a user-supplied MATLAB function with a header of the form

```
function [f, g, H] = fun(x,p1,p2,...)
```

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

The functions `linesearch` and `ucminf` only need a user-supplied function of the form

```
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 `fun`.

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

1.1.1. User's guide to `dampnewton`

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

$$\left(\boldsymbol{\nabla}^2 f(\boldsymbol{x}) + \mu \boldsymbol{I}\right) \boldsymbol{h} = -\boldsymbol{\nabla} f(\boldsymbol{x})$$

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

Typical calls are

```
[X, info] = dampnewton(fun, x0)
[X, info] = dampnewton(fun, x0, opts, p1,p2,...)
[X, info, perf] = dampnewton(.....)
```

Input parameters

`fun`	Handle to the function.
`x0`	Starting guess for $\hat{\boldsymbol{x}}$.
`opts`	Either a struct with fields `'tau'`, `'tolg'`, `'tolx'` and `'maxeval'`, or a vector with the values of these options, `opts = [tau tolg tolx maxeval]` .
`tau`	is used to get starting value for the damping parameter: $\mu = \texttt{tau} * \Vert\boldsymbol{\nabla}^2 f(\texttt{x0})\Vert_\infty$ The other options are used in the stopping criteria (3.1),

$$\begin{aligned} \|\boldsymbol{\nabla} f(\boldsymbol{x})\|_\infty &\le \texttt{tolg} && \text{or} \\ \|\delta \boldsymbol{x}\|_2 &\le \texttt{tolx}\,(\texttt{tolx} + \|\boldsymbol{x}\|_2) && \text{or} \end{aligned}$$

no. of function evaluations exceeds `maxeval`

Default `tau = 1e-3`, `tolg = 1e-4`, `tolx = 1e-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.

`info` Performance information, vector with 7 elements:

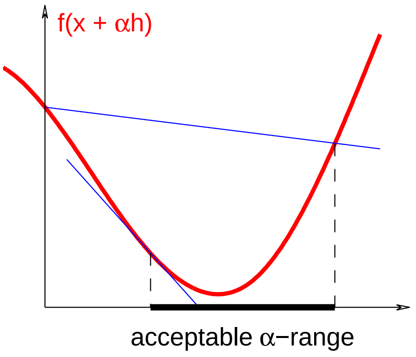
`info(1:4)` Final values of $\left[f(\boldsymbol{x}),\quad \|\boldsymbol{\nabla} f(\boldsymbol{x})\|_\infty,\quad \|\delta\boldsymbol{x}\|_2,\quad \mu/\max|(\boldsymbol{\nabla}^2 f(x))_{ii}|\right]$.

`info(5:6)` Number of iteration steps and function evaluations.

`info(7) =`
1: Stopped by a small gradient.
2: Stopped by a small $\boldsymbol{x}$-step,
3: No. of function evaluations exceeds `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.

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

1.1.2. User's guide to `linesearch`



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

$$\varphi(\alpha) \;=\; f(\boldsymbol{x}+\alpha\boldsymbol{h})\ , \qquad \varphi'(\alpha) \;=\; \boldsymbol{h}^T \boldsymbol{\nabla} f(\boldsymbol{x}+\alpha\boldsymbol{h})\ .$$

`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 $\varphi(\hat{\alpha}) \le \varphi(0) + \hat{\alpha}\beta_1\varphi'(0)$ and $\varphi'(\hat{\alpha}) \ge \beta_2\varphi'(0)$. $\beta_1 < \beta_2$ are input.

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

Typical calls are

```
[xn,fn,gn,info] = linesearch(fun,x,f,g,h)
[xn,fn,gn,info] = linesearch(fun,x,f,g,h,opts,p1,p2,...)
[xn,fn,gn,info,perf] = linesearch(......)
```

Input parameters

`fun` Handle to the function.

`x` Current $\boldsymbol{x}$.

`f,g` $f(\boldsymbol{x})$ and $\boldsymbol{\nabla} f(\boldsymbol{x})$.

`h` Step vector.

`opts` Either a struct with fields `'choice'`, `'cp1'`, `'cp2'`, `'maxeval'`, `'amax'` or a vector with the values of these options, `opts = [choice cp1 cp2 maxeval amax]`.

- `choice` $= 0$: exact line search.
  - Otherwise soft line search (Default).
- `cp1, cp2`: options for stopping criteria.
  - `choice = 0`: $|\varphi'(a)| \le \texttt{cp1} * |\varphi'(0)|$ or $c - b \le \texttt{cp2} * c$, where $a = \hat{\alpha}$ and $[b, c]$ is the current interval for a. Default $\texttt{cp1} = \texttt{cp2} = 10^{-3}$.
  - `otherwise`: $\varphi(a) \le \varphi(0) + a * \texttt{cp1} * \varphi'(0)$ and $\varphi'(a) \ge \texttt{cp2} * \varphi'(0)$. Default $\texttt{cp1} = 10^{-3}$, $\texttt{cp2} = 0.99$.
- `maxeval` Maximum number of function evaluations. Default $\texttt{maxeval} = 10$.
- `amax` Maximal allowable $\alpha-$value. Default $\texttt{amax} = 10$.

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

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

Output parameters

`xn` New iterate, $\boldsymbol{x} + \hat{\alpha}\boldsymbol{h}$.

`fn,gn` $f(\texttt{xn})$ and $\boldsymbol{\nabla} f(\texttt{xn})$

`info` Performance information, vector with 3 elements:

- `info(1)` > 0: Successful call. Value of $\hat{\alpha}$.
  - $= 0$: $\boldsymbol{h}$ is not downhill or it is so large and `maxeval` so small, that a better point was not found.
  - $= -1$: `x` is not a real valued vector.
  - $= -2$: `f` is not a real valued scalar.
  - $= -3$: `g` or `h` is not a real valued vector.
  - $= -4$: `g` or `h` has different length from `x`.
- `info(2)` Slope ratio $\varphi'(\hat{\alpha})/\varphi'(0)$.
- `info(3)` Number of function evaluations used.

`perf` Struct with fields

- `alpha:` values of α ,
- `phi:` values of $\varphi(\alpha)$,
- `slope:` values of $\varphi'(\alpha)$.

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].

BFGS updating

soft line search

Typical calls are

```
[X, info] = ucminf(fun, x0)
[X, info] = ucminf(fun, x0, opts)
[X, info] = ucminf(fun, x0, opts, D0, p1,p2,...)
[X, info, perf] = ucminf(.....)
[X, info, perf, D] = ucminf(.....)
```

Input parameters

`fun` Handle to the function.

`x0` Starting guess for $\hat{\boldsymbol{x}}$.

`opts` Either a struct with fields `'Delta'`, `'tolg'`, `'tolx'` and `'maxeval'`, or a vector with the values of these options, `opts = [Delta tolg tolx maxeval]` .

`Delta`: Expected length of initial step.

The other options are used in the stopping criteria (3.1),

$$\begin{aligned} \|\nabla f(\boldsymbol{x})\|_\infty &\le \texttt{tolg} && \text{or} \\ \|\delta\boldsymbol{x}\|_2 &\le \texttt{tolx}\,(\texttt{tolx} + \|\boldsymbol{x}\|_2) && \text{or} \end{aligned}$$

no. of function evaluations exceeds `maxeval`

Default `Delta = 1`, `tolg = 1e-4`, `tolx = 1e-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.

`D0` If present, then approximate inverse Hessian at `x`. Otherwise, `D0` $:= \boldsymbol{I}$.

`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.

`info` Performance information, vector with 6 elements:

`info(1:3)`	Final values of $[f(\boldsymbol{x}),\ \ \|\nabla f(\boldsymbol{x})\|_\infty,\ \ \|\delta\boldsymbol{x}\|_2]$.
`info(4:5)`	Number of iterations steps and evaluations of f and ∇f.
`info(6) =`	1: Stopped by a small gradient. 2: Stopped by a small $\boldsymbol{x}$-step. 3: No. of function evaluations exceeds `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 $f(\boldsymbol{x}_k)$,
`ng`:	values of $\|\nabla f(\boldsymbol{x}_k)\|_\infty$,
`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
```

```
    p12 = p1^2;  h12 = -2*p12*x(1);
    H = [1-2*p1*(f1 - 2*p1*x(1)^2) h12
      h12  p12];
  end
end
```

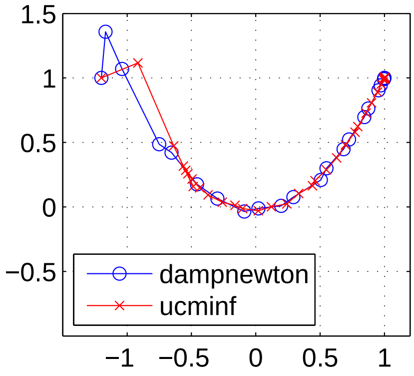
Choosing the parameters $p_1 = 10$ and $p_2 = 0$ this is an implementation of the famous Rosenbrock function. In the program

```
[X1, ii1, pf1] = dampnewton(@rosenbrock,[-1.2 1],opts,10,0);
[X2, ii2] = ucminf(@rosenbrock,[-1.2 1],[],[],10,0)
```

we use the default values for `opts` and an empty `D0` in `ucminf`, and get the results,

```
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
```

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



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

```
plot(X1(1,:),X1(2,:),'-ob', X2(1,:),X2(2,:),'-xr')
```

In `X2` every column 2,3,... is obtained by line search from the previous, and `ii2(4:5)` shows that in almost every iteration step we onle need one function evaluation. In the damped Newton case `info(6)` $=$ `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 = pf1.mu(2:end) ./ pf1.mu(1:end-1)
```

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

To illustrate `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 `opts`.

```
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)
```

We get the following results,

```
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
```

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

1.2. Nonlinear Least Squares Problems

We seek a minimizer $\hat{\boldsymbol{x}}$ of the function

$$f(\boldsymbol{x}) \;=\; \tfrac{1}{2}\sum_{i=1}^{m} r_i(\boldsymbol{x})^2 \;=\; \tfrac{1}{2}\boldsymbol{r}(\boldsymbol{x})^T\boldsymbol{r}(\boldsymbol{x}) \;, \tag{1.1}$$

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

$$\boldsymbol{\nabla} f(\boldsymbol{x}) \;=\; J(\boldsymbol{x})^T\boldsymbol{r}(\boldsymbol{x}) \;.$$

$J(\boldsymbol{x})$ is the Jacobian, as defined by

$$J_{ij} \;=\; \frac{\partial r_i}{\partial x_j}$$

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

```
function [r, J] = fun(x,p1,p2,...)
```

The function should return $\boldsymbol{r}(\boldsymbol{x})$ 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 $J(\boldsymbol{x})$ can be checked by `checkgrad`, Section 3.2.2.

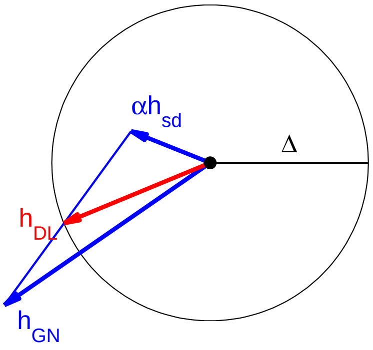
The function `smarquardt` only needs a user supplied function with the header of the form

```
function r = fun(x,p1,p2,...)
```

Also this function should return $\boldsymbol{r}(\boldsymbol{x})$ as a column vector in `r`.

1.2.1. User's guide to dogleg

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



Typical calls are

```
[X,info] = dogleg(fun,x0)
[X,info] = dogleg(fun,x0,opts,p1,p2,...)
[X,info,perf] = dogleg(......)
```

Input parameters

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

`opts = [Delta tolg tolx tolr maxeval]` .

`Delta` Initial trust region radius Δ.

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

$$\begin{aligned} \|\boldsymbol{\nabla} f(\boldsymbol{x})\|_\infty &\le \texttt{tolg} && \text{or}\\ \|\delta\boldsymbol{x}\|_2 &\le \texttt{tolx}\,(\texttt{tolx} + \|\boldsymbol{x}\|_2) && \text{or}\\ \|\boldsymbol{r}(\boldsymbol{x})\|_\infty &\le \texttt{tolr} && \text{or} \end{aligned}$$

no. of function evaluations exceeds `maxeval`

Default `Delta` $= 0.1(1 + \|\boldsymbol{x}_0\|_2)$, `tolg` $= 10^{-4}$, `tolx` $= 10^{-8}$, `tolr` $= 10^{-6}$, `maxeval` $= 100$. If the input `opts` has less than 5 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.
`info`	Performance information, vector with 7 elements:

`info(1:4)`	Final values of $[f(\boldsymbol{x}),\quad \|\nabla f(\boldsymbol{x})\|_\infty,\quad \|\delta\boldsymbol{x}\|_2,\quad \Delta]$.
`info(5:6)`	No. of iteration steps and function evaluations.
`info(7) =`	1: Stopped by a small gradient. 2: Stopped by a small $\boldsymbol{x}$-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.

`perf` Struct with fields

`f:`	values of $f(\boldsymbol{x}_k)$,
`ng:`	values of $\|\nabla f(\boldsymbol{x}_k)\|_\infty$,
`Delta:`	values of trust region radius Δ.

1.2.2. User's guide to `marquardt`

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

$$\left(\boldsymbol{J}(\boldsymbol{x})^T\boldsymbol{J}(\boldsymbol{x}) + \mu\boldsymbol{I}\right)\boldsymbol{h} \\ = -\nabla f(\boldsymbol{x})$$

Typical calls are

```
[X,info] = marquardt(fun,x0)
[X,info] = marquardt(fun,x0,opts,p1,p2,...)
[X,info,perf] = marquardt(......)
```

Input parameters

`fun`	Handle to the function.
`x0`	Starting guess for $\hat{\boldsymbol{x}}$.
`opts`	Either a struct with fields `'tau'`, `'tolg'`, `'tolx'` and `'maxeval'`, or a vector with the values of these options, `opts = [tau tolg tolx maxeval]` . `tau` used in starting value for Marquardt parameter: $\mu = \tau \cdot \max\{\left(\boldsymbol{J}(\boldsymbol{x}_0)^T\boldsymbol{J}(\boldsymbol{x}_0)\right)_{ii}\}$. The other options are used in the stopping criteria (3.1), $\|\nabla f(\boldsymbol{x})\|_\infty \le \texttt{tolg}$ or $\|\delta\boldsymbol{x}\|_2 \le \texttt{tolx}\,(\texttt{tolx} + \|\boldsymbol{x}\|_2)$ or no. of function evaluations exceeds `maxeval`. Default $\texttt{tau} = 10^{-3}$, $\texttt{tolg} = 10^{-4}$, $\texttt{tolx} = 10^{-8}$, $\texttt{maxeval} = 100$. 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.

`info` Performance information, vector with 7 elements:

`info(1:4)` Final values of

$$\left[f(\boldsymbol{x}), \quad \|\boldsymbol{\nabla} f(\boldsymbol{x})\|_\infty, \quad \|\delta\boldsymbol{x}\|_2, \quad \frac{\mu}{max\{(\boldsymbol{J}(\boldsymbol{x})^T\boldsymbol{J}(\boldsymbol{x}))_{ii}\}} \right] .$$

`info(5:6)` No. of iteration steps and function evaluations.

`info(7) =` 1: Stopped by a small gradient.
2: Stopped by a small $\boldsymbol{x}$-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.

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

1.2.3. User's guide to `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 [4, Section 6.4].

$$\boldsymbol{B} := \boldsymbol{B} + \frac{(\boldsymbol{r}_{new} - \boldsymbol{r} - \boldsymbol{B}\boldsymbol{h})\,\boldsymbol{h}^T}{\boldsymbol{h}^T\boldsymbol{h}}$$

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

$$B_{:,j} = \frac{\boldsymbol{r}(\boldsymbol{x} + h_j\boldsymbol{e}_j) - \boldsymbol{r}(\boldsymbol{x})}{h_j} ,$$

Where $\boldsymbol{e}_j$ 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, `'relstep'`.

Typical calls are

```
[X, info] = smarquardt(fun,x0)
[X, info] = smarquardt(fun,x0,opts)
[X, info] = smarquardt(fun,x0,opts,B0,p1,p2,...)
[X, info, perf] = smarquardt(......)
[X, info, perf, B] = smarquardt(......)
```

Input parameters

`fun` Handle to the function that returns $\boldsymbol{r}(\boldsymbol{x})$.

`x0` Starting guess for $\hat{\boldsymbol{x}}$.

`opts` Either a struct with fields `'tau'`, `'tolg'`, `'tolx'`, `'maxeval'` and `'relstep'`, or a vector with the values of these options,

`opts = [tau tolg tolx maxeval relstep]` .

`tau` used in starting value for Marquardt parameter:

$$\mu = \tau \cdot \max\{\left(\boldsymbol{B}_0^T\boldsymbol{B}_0\right)_{ii}\} ,$$

where $\boldsymbol{B}_0$ is an approximate Jacobian at $\boldsymbol{x}_0$.

`relstep` d, “relative” step length for difference approximations.

The other options are used in the stopping criteria, cf (3.1),

$\|\boldsymbol{B}(\boldsymbol{x})^T \boldsymbol{r}(\boldsymbol{x})\|_\infty \le \texttt{tolg}$ or

$\|\delta\boldsymbol{x}\|_2 \le \texttt{tolx}\,(\texttt{tolx} + \|\boldsymbol{x}\|_2)$ or

no. of function evaluations exceeds `maxeval`.

Default $\texttt{tau} = 10^{-3}$, $\texttt{tolg} = 10^{-4}$, $\texttt{tolx} = 10^{-8}$,
$\texttt{maxeval} = 100 + 10n$, $\texttt{relstep} = 10^{-6}$.

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

`B0` (Approximation to) $\boldsymbol{J}(\boldsymbol{x}_0)$.
If `B0` is not given or is empty, a forward difference approximation to it is used.

`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.

`info` Performance information, vector with 7 elements:

`info(1:4)` Final values of

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

`info(5)` Number of function evaluations.

`info(6) =`
1: Stopped by a small gradient.
2: Stopped by a small $\boldsymbol{x}$-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.

`perf` Struct with fields
`f:` values of $f(\boldsymbol{x}_k)$,
`ng:` values of $\|\boldsymbol{B}(\boldsymbol{x})^T \boldsymbol{r}(\boldsymbol{x})\|_\infty$,
`mu:` values of damping parameter μ.

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

1.2.4. Example

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

```
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
```

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 `x3 = X3(:,end)`,

```
x1 = 1      x2 = 0.9999992008      x3 =   0.9999998944
     1           0.9999983957             0.9999997841


ii1 = 1.00e+06      0        6.02e-03   2.16e-01    16    21     1
ii2 = 1.00e+06   5.87e-07    1.17e-04   2.55e-06    13    16     1
ii3 = 1.00e+06   8.16e-07    2.20e-05   1.79e-06    18    50     1
```

In all the cases we find a good approximation to the true minimizer, $\hat{\boldsymbol{x}} = [1\ 1]$, and `iix(7) = 1` shows that iteration stops because the the infinity norm of the gradient `iix(2)` is smaller than the default threshold `tolg = 1e-4`.

`smarquardt` uses almost the same number of iterations as `marquardt`, costing a total of 50 function evaluations, whereas `marquardt` uses 16 evaluations of $\boldsymbol{r}(\boldsymbol{x})$ *and* the Jacobian $\boldsymbol{J}(\boldsymbol{x})$. The computed approximation to $\boldsymbol{J}(\hat{\boldsymbol{x}})$ and the true Jacobian at the minimizer are

```
B = -19.969       9.985          J(xm) = -20     10
        -1      -6.53e-17                  -1     0
         0          0                       0     0
```

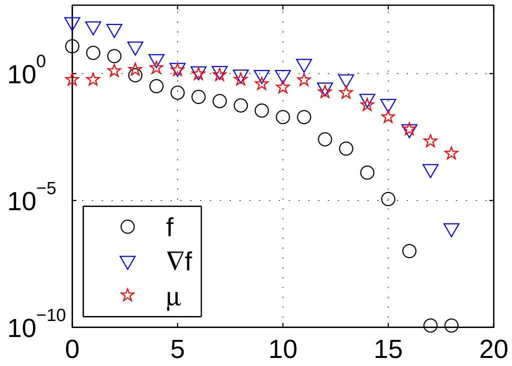
Thus, also $\boldsymbol{B}$ is a good approximation to the Jacobian $\boldsymbol{J}(\hat{\boldsymbol{x}})$.

Finally,

```
t = 0 :  18;
semilogy(t,pf3.f-1e6,'ok', ...
    t,pf3.ng,'vb', t,pf3.mu,'pr')
```

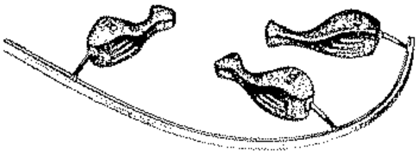
illustrates the iteration with `smarquardt`. We subtract the final value in order to be able see the variation in the function values.

Both f and $\boldsymbol{\nabla} f$ show superlinear convergence in the final iterations, and μ 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

$$\begin{aligned} d_{11}s(a) + d_{12}s'(a) + d_{13}s''(a) &= d_{14} \\ d_{21}s(b) + d_{22}s'(b) + d_{23}s''(b) &= d_{24} \end{aligned} \tag{2.1}$$

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 2×4 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 = splfit(tyw, x)
S = splfit(tyw, x, D)
```

Input parameters

`tyw`	Data points and weights. Array with 2 or 3 columns,	
	`tyw(:,1)`	Data abscissas.
	`tyw(:,2)`	Data ordinates.
	`tyw(:,3)`	Weights. If `tyw` holds less than 3 columns, then all weights are set to 1.
`x`	Knots. Must satisfy `x(1) < x(2) <= ... <= x(end-1) < x(end)` .	
`D`	2×4 array with the d_{ij} presented in (2.1). `a = x(1)`, `b = x(end)`.	

Output parameters

`S`	Struct representing the spline, as described on page 13.

2.1.2. User's guide to `splineval`

This function can be used to evaluate a cubic spline, computed by splinefit. The evaluation is done by means of the piecewise 3rd order polynomial provided in `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

```
f = splval(S, t)
f = splval(t, S)
```

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

Input parameters

`S`	Struct representing the spline, as described on page 13.
`t`	Vector with arguments for the spline.

Output parameters

`f`	Vector of the same type as `t`, with $f_i = s(t_i)$.

2.1.3. User's guide to `splinedif`

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

Typical calls are

```
f = splinedif(S, t)
f = splinedif(S, t, d)
```

Input parameters

`S`	Struct representing the spline, as described on page 13.
`t`	Vector with arguments for the spline. If any `t(i)` is outside the knot range, you get at error return.
`d`	Differentation order. Default $\mathtt{d} = 1$.

Output parameters

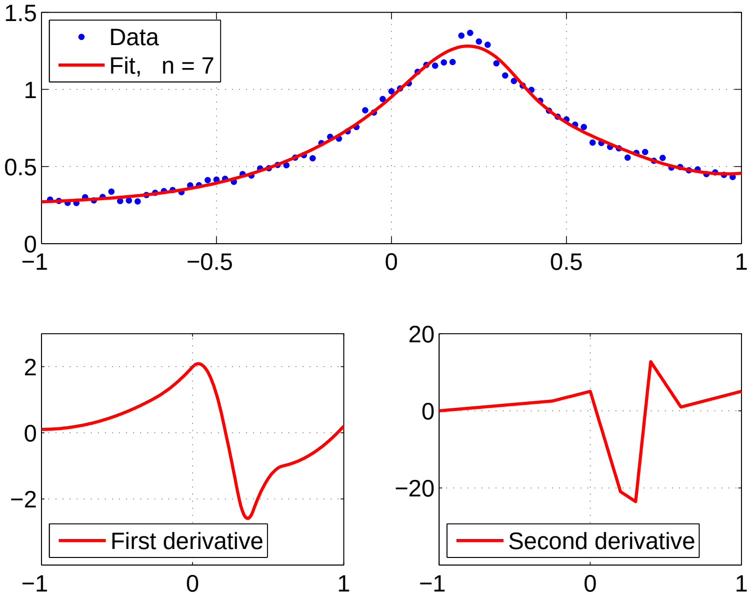
`f`	Vector of the same type as `t`, with $f_i = s^{(d)}(t_i)$.

2.1.4. Example

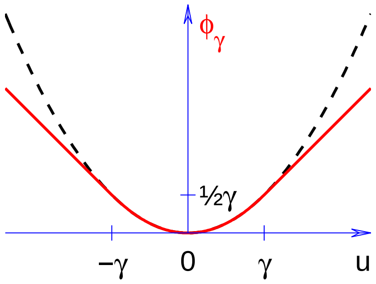
The toolbox data set `wild.dat` discussed on page 29 was generated by adding ”noise” to a function defined on `[-1, 1]` with slope approximately equal to `0.2` at both end points. This is used at the right hand end in the following Matlab program, while we use natural spline condition at the other end.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

```
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''.



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 γ 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 `linhuber` finds the minimizer of a slightly extended version of the linear Huber estimation problem,

$$f(\boldsymbol{x}) \;=\; \sum_{i=1}^{m} \phi_\gamma(r_i(\boldsymbol{x})) + \tfrac{1}{2}\mu\|L\boldsymbol{x}\|_2^2 + \boldsymbol{c}^T\boldsymbol{x} \tag{2.2}$$

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

In connection with `nonlinhuber` and `huberobj` the user must supply a MATLAB function with a header of the form

```
function [r, J] = fun(x, p1,p2,...)
```
(2.3)

(cf Sections 1.2.1 and 1.2.2 (`dogleg` and `marquardt`)). The function should return $\boldsymbol{r}(\boldsymbol{x})$ 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 $J(\boldsymbol{x})$ can be checked by `checkgrad`, Section 3.2.2.

2.2.1. User's guide to `linhuber`

The function $f(\boldsymbol{x})$ 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, info] = linhuber(A,b,c,L,par)
[X, info] = linhuber(A,b,c,L,par,init)
[X, info, perf] = linhuber(......)
```

Input parameters

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

`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 $\boldsymbol{Ax} \simeq \boldsymbol{b}$.

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 [$f(\hat{\boldsymbol{x}})$, $\lVert\boldsymbol{\nabla} f(\hat{\boldsymbol{x}})\rVert_\infty$, 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`: [`length(S.A) 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 $f(\boldsymbol{x})$, `ng`: values of $\lVert\boldsymbol{\nabla} f(\boldsymbol{x})\rVert_\infty$. `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 $f(\boldsymbol{x}) = \frac{1}{2}\lVert\boldsymbol{r}(\boldsymbol{x})\rVert_2^2$. With the default γ-value all the equations are active.

Typical calls are

```
[X, S, info] = nonlinhuber(fun, x0)
[X, S, info] = nonlinhuber(fun, x0, opts, p1,p2,...)
[X, S, info, perf] = 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 γ . `Htype` Choice of Huber function, `Htype` = 1: One-sided, $r_i > 0$ are neglected, 2: one-sided, all $r_i \le \gamma$ are active, otherwise: standard Huber, ie equations with $\lvert r_i\rvert \le \gamma$ are active. `tau` used in starting value for Marquardt parameter: $\mu = \tau \cdot \max\{\big(\boldsymbol{J}(\boldsymbol{x}_0)^T\boldsymbol{J}(\boldsymbol{x}_0)\big)_{ii}\}$. The other options are used in the stopping criteria (3.1), $\lVert\boldsymbol{\nabla} f(\boldsymbol{x})\rVert_\infty \le \texttt{tolg}$ or $\lVert\delta\boldsymbol{x}\rVert_2 \le \texttt{tolx}\,(\texttt{tolx} + \lVert\boldsymbol{x}\rVert_2)$ or no. of function evaluations exceeds `maxeval`.

Default `gamma` $= \Vert \boldsymbol{r}(\boldsymbol{x}_0)\Vert_\infty$ `Htype` $= 0$ `tau` $= 10^{-3}$
`tolg` $= 10^{-4}$ `tolx` $= 10^{-8}$ `maxeval` $= 100$

If the input `opts` has less than 6 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.
`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 [`length(S.A)`,`length(S.N)`] .
`info`	Performance information, vector with 7 elements: `info(1:4)` Final values of $\left[f(\boldsymbol{x}),\quad \Vert \boldsymbol{\nabla} f(\boldsymbol{x})\Vert_\infty,\quad \Vert \delta\boldsymbol{x}\Vert_2,\quad \frac{\mu}{max\{(\boldsymbol{J}(\boldsymbol{x})^T\boldsymbol{J}(\boldsymbol{x}))_{ii}\}} \right]$. `info(5:6)` No. of iteration steps and function evaluations. `info(7) =` 1: Stopped by a small gradient. 2: Stopped by a small $\boldsymbol{x}$-step, 3: No. of function evaluations exceeds `maxeval`
`perf`	Struct with fields `f`: values of $f(\boldsymbol{x}_k)$, `ng`: values of $\Vert \boldsymbol{\nabla} f(\boldsymbol{x}_k)\Vert_\infty$, `mu`: values of damping parameter μ.

2.2.3. User's guide to `huberobj`

This function can be used to compute values of f and its gradient, where $f(\boldsymbol{x})$ is the simple version of ((2.2)),

$$f(\boldsymbol{x}) = \sum_{i=1}^{m} \phi(r_i(\boldsymbol{x})) \ . \tag{2.4}$$

ϕ is the Huber function, and $\boldsymbol{r}$ is a vector function.

Typical calls are

```
[f, S, r] = huberobj(fun, x, gamma)
[f, S, r] = huberobj(fun, x, gamma, Htype, p1,p2,...)
[f, S, r, J, g] = huberobj(.....)
```

Input parameters

`fun`	Handle to the function discussed at (2.3).
`x`	n-vector. Argument.
`gamma`	Huber threshold.
`Htype`	Choice of Huber function, 1 : one-sided, $\rho(r_i) = 0$ for $r_i > 0$. 2 : one-sided, all $r_i \le \gamma$ are active. Otherwise: standard Huber (default), ie equations with $\lvert r_i\rvert \le \gamma$ are active.
`p1,p2,...`	are passed directly to the function `fun`.

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 $\boldsymbol{r}$), `N`: inactive set (indices of large elements in $\boldsymbol{r}$), `L`: [`length(S.A) length(S.N)`].
`r,J`	Output from the evaluation of `fun`. If `nargout<4`, then `fun` only needs to return $\boldsymbol{r}(\boldsymbol{x})$, and the gradient is not computed.
`g`	Gradient, $\texttt{g} = \boldsymbol{\nabla} f(\boldsymbol{x})$.

2.2.4. Example

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

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

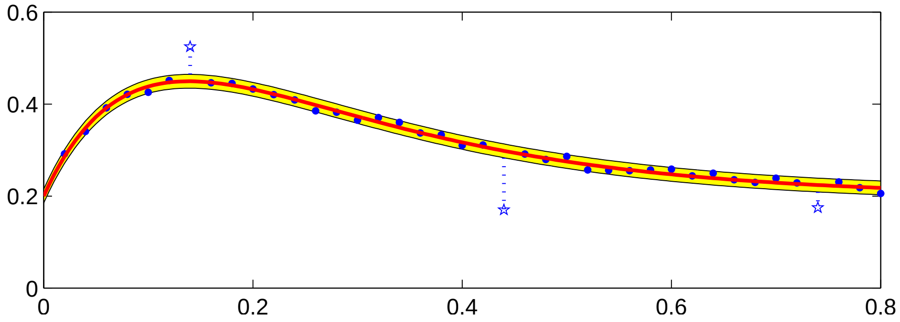
The following program computes the Huber estimator for $\gamma = 0.015$.

```
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     info =  pf: [0.28914 0 2 2]     S = s: [1x40 double]
      0.99901              S: [1x1 struct]            A: [1x37 double]
      0.20001              R: [3x3 double]            N: [7 22 37]
                           p: [3 2 1]                 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 $M(\boldsymbol{x}_\gamma, t) - \gamma$ and $M(\boldsymbol{x}_\gamma, t) + \gamma$.



Next, consider the above fitting model with unknown exponents

$$M(\boldsymbol{x}, t) \;=\; x_3 e^{-x_1 t} + x_4 e^{-x_2 t} + x_5 \;.$$

The corresponding `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 $\gamma = 10^{-5}$:

```
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

```
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 γ, 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 $\gamma \rightarrow 0$ we might try $\gamma = 10^{-15}$:

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

This, however, gives an error return

```
???  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 $\boldsymbol{r}(\boldsymbol{x})$ may be larger than this small value of γ, with the meaningless effect, eg, that the active set may be considered as empty.

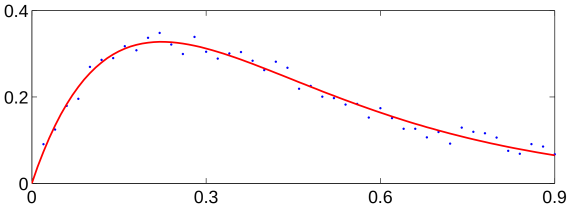
2.3. Multiexponential Fitting

A fitting model of the form

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

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 $\boldsymbol{c}$) occur linearly. The special-purpose method is known as *separable least squares*, cf eg [4, Example 6.20], [2], [11], and is implemented in `mexpfit`.

2.3.1. User's guide to `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

$$\boldsymbol{F}(\boldsymbol{z})\boldsymbol{c}(\boldsymbol{z}) \;\simeq\; \boldsymbol{y}\;,$$

where $\boldsymbol{c}(\boldsymbol{z})$ 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 $f(\boldsymbol{z}) = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{F}(\boldsymbol{z})\boldsymbol{c}(\boldsymbol{z})\|_2^2$.

Typical calls are

```
[Z, c, info] = mexpfit(tyw, z0)
[Z, c, info] = mexpfit(tyw, z0, opts)
[Z, c, info, perf] = mexpfit(...)
```

Input parameters

`tyw` Data points and weights. Array with 2 or 3 columns,

`tyw(:,1:2)` Abscissas and ordinates of data points.

`tyw(:,3)` Weights. If `tyw` has less than 3 columns, then all weights are set to 1.

`z0` Starting guess for $\boldsymbol{z}$.

`opts` Either a struct with fields `'const'`, `'tau'`, `'tolg'`, `'tolx'`, and `'maxeval'`, or a vector with the values of these options, `opts = [const tau tolg tolx maxeval]`.

`const` If positive then there is a constant term.

`tau` Used in starting value for Marquardt parameter for the optimization, $\mu = \texttt{tau} * \max\{\ [\boldsymbol{J}(\boldsymbol{z}_0)^T \boldsymbol{J}(\boldsymbol{z}_0)]_{ii}\}$.

The other options are used in stopping criteria:

$$\|\boldsymbol{\nabla} f(\boldsymbol{z})\|_\infty \;\le\; \texttt{tolg} \qquad \text{or}$$

$$\|\delta\boldsymbol{z}\|_2 \;\le\; \texttt{tolx}\,(\texttt{tolx} + \|\boldsymbol{z}\|_2) \qquad \text{or}$$

no. of function evaluations exceeds `maxeval`

Default $\texttt{const} = 0$, $\texttt{tau} = 10^{-3}$, $\texttt{tolg} = 10^{-6}$, $\texttt{tolx} = 10^{-10}$, $\texttt{maxeval} = 100$.
If the input `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

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

`c`	If `info.fail = 0`, then `c` holds coefficients $[c_1, \ldots, c_q]$, where `opts.const` > 0 : $q = p{+}1$; c_q is the constant term, otherwise : $q = p$.
`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 $f(\boldsymbol{z})$, $\|\boldsymbol{\nabla} f(\boldsymbol{z})\|_\infty$, $\|\delta\boldsymbol{z}\|_2$. `its:` Vector with number of iterations and function evaluations.
`perf`	Struct with fields `f:` values of $f(\boldsymbol{z}_k)$, `ng:` values of $\|\boldsymbol{\nabla} f(\boldsymbol{z}_k)\|_\infty$, `mu:` values of damping parameter μ.

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

$$M(\boldsymbol{x}, t) \;=\; c_1 e^{-z_1 t} + c_2 e^{-z_2 t} + c_3$$

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”, ie they have exceptionally large errors, and we shall neglect them in this example. Finally, we take the poor starting guess $\boldsymbol{z}_0 = [5\ 3]$, and except for `const` $= 1$ we use the default `opts`-values.

```
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 $\hat{\boldsymbol{x}}$ of a vector function $\boldsymbol{r}$, as given by a MATLAB function with a header of the form

```
function r = fun(x,p1,p2,...)
```

The function should return $\boldsymbol{r}(\boldsymbol{x})$ as a column vector in `r`.
(`p1,p2,...` are possible parameters of $\boldsymbol{r}$).

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

3.1.1. User's guide to `nonlinsys`

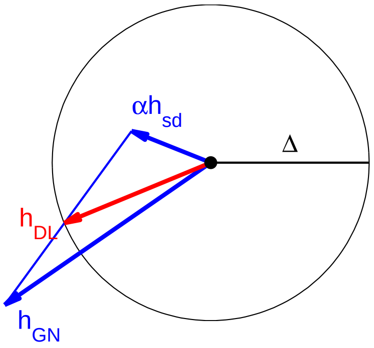
This function is based on Powell's dog-leg algorithm, as described eg in [4, Section 6.5]. A root $\hat{\boldsymbol{x}}$ of the vector function $\boldsymbol{r}(\boldsymbol{x})$ is a minimizer of

$$\phi(\boldsymbol{x}) \;=\; \tfrac{1}{2}\boldsymbol{r}(\boldsymbol{x})^T\boldsymbol{r}(\boldsymbol{x})$$

The gradient of ϕ and the Newton step for r are given by respectively

$$\boldsymbol{\nabla}\phi \;=\; \boldsymbol{J}\boldsymbol{r} \quad \text{and} \quad \boldsymbol{h}_{\text{GN}} \;=\; -\boldsymbol{J}^{-1}\boldsymbol{r}\ .$$

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



Typical calls are

```
[X, info] = nonlinsys(fun, x0)
[X, info] = nonlinsys(fun, x0, opts)
[X, info] = nonlinsys(fun, x0, opts, B0, p1,p2,...)
[X, info, perf] = nonlinsys(.....)
```

Input parameters

`fun`	Handle to the function.	
`x0`	Starting guess for $\hat{\boldsymbol{x}}$.	
`opts`	Vector with five elements,	
	`opts(1)`	initial trust region radius Δ_0.
	`opts(2:4)`	used in stopping criteria: $\Vert\boldsymbol{\nabla}r(\boldsymbol{x})\Vert_\infty \le$ `opts(2)` or $\Vert\delta\boldsymbol{x}\Vert_2 \le$ `opts(3)` (`opts(3)` $+ \Vert\boldsymbol{x}\Vert_2$) or no. of function evaluations exceeds `opts(4)` <= update!
	`opts(5)`	"relative" step length for difference approximations.
	Default `opts` = $[0.1(1+\Vert\boldsymbol{x}_0\Vert)$ `1e-6 1e-8 100 1e-6`$]$	

If the input `opts` has less than 5 elements, it is augmented by the default values. Also, non-positive elements are replaced by the default values.

`B0`	Initial approximation to the Jacobian of $\boldsymbol{J}_r(\boldsymbol{x}_0)$. If `B0` is not given, a forward difference approximation to $\boldsymbol{J}_r(\boldsymbol{x}_0)$ is used.
`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.

`info` Performance information, vector with 6 elements:

`info(1:3)`	Final values of $\|\boldsymbol{r}(\boldsymbol{x})\|_\infty$, $\|\boldsymbol{\nabla} r(\boldsymbol{x})\|_\infty$, $\|\delta\boldsymbol{x}\|_2$, Δ .
`info(4:5)`	Number of iteration steps and function evaluations.
`info(6) =`	

1:	Stopped by small $\boldsymbol{r}$-vector.
2:	Stopped by a small $\boldsymbol{x}$-step,
3:	No. of function evaluations exceeds `opts(4)` <= update!
-1:	`x` is not a real valued vector.
-2:	`r` is not a real valued column vector.
-3:	Dimension mismatch in `x`, `r`, `B0`.
-4:	Maybe started at a saddle point.
-5:	Overflow during computation.

`perf` Array, holding

`perf(1,:)`	values of $\|r(\boldsymbol{x})\|_\infty$,
`perf(2,:)`	values of Δ .

3.1.2. Example

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

```
function  r = rr(x)
r = [4*x(1)^2 + 9*x(2)^2 - 36 ; 16*x(1)^2 - 9*x(2)^2 - 36];
```

In the call

```
[x ii] = nonlinsys(@rr, [1 1])
```

we use the starting point $\boldsymbol{x}_0 = (1,\ 1)$, default values for `opts` and an empty `B0`. We get the following results,

```
x =  1.8974
     1.5492


ii = 3.9703e-07  1.8401e-05   2.2481    7   14    1
```

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

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

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.

`dampnewton`	`'tau'`	`'tolg'`	`'tolx'`	`'maxeval'`		
`linesearch`	`'choice'`	`'cp1'`	`'cp2'`	`'maxeval'`	`'amax'`	
`ucminf`	`'Delta'`	`'tolg'`	`'tolx'`	`'maxeval'`		
`dogleg`	`'Delta'`	`'tolg'`	`'tolx'`	`'tolr'`	`'maxeval'`	
`marquardt`	`'tau'`	`'tolg'`	`'tolx'`	`'maxeval'`		
`smarquardt`	`'tau'`	`'tolg'`	`'tolx'`	`'maxeval'`	`'relstep'`	
`nonlinhuber`	`'gamma'`	`'Htype'`	`'tau'`	`'tolg'`	`'tolx'`	`'maxeval'`
`mexpfit`	`'const'`	`'tau'`	`'tolg'`	`'tolx'`	`'maxeval'`	
`nonlinsys`	`'Delta'`	`'tolg'`	`'tolx'`	`'maxeval'`	`'relstep'`	

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

$$\begin{aligned} \|\boldsymbol{\nabla} f(\boldsymbol{x})\|_\infty &\le \texttt{tolg} && \text{or} \\ \|\delta\boldsymbol{x}\|_2 &\le \texttt{tolx}\,(\texttt{tolx} + \|\boldsymbol{x}\|_2) && \text{or} \qquad (3.1) \\ &\text{no. of function evaluations exceeds } \texttt{maxeval} \end{aligned}$$

The other options are explained in connection with the function.

A typical call is `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

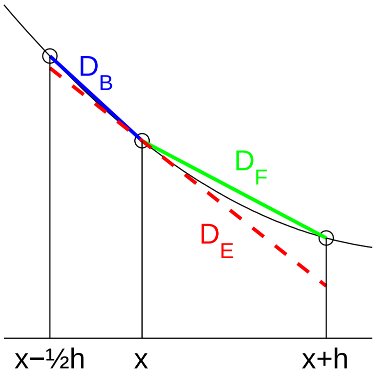
`opts`	Struct with the option names and their values. Options that do not appear as input are assigned their default value. The values are **not** checked for feasibility; this is done in the function defined by `mlfun`.

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

$$u_B \simeq -\tfrac{1}{2}u_F$$

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



A typical call is `[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. $h_j \simeq 10^{-6}\lvert x_j\rvert$ is generally a good choice.
`p1,p2,...`	are passed directly to the function `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`	3×2 array, with `index(k,:)` giving the position in matrix of partial derivatives, where `err(k)` occurs.

3.2.3. Example

The calls `opts1 = immoptset(@marquardt)` and `opts = immoptset('marquardt')` both give the default values

```
opts =      tau: 0.001
           tolg: 0.0001
           tolx: 1e-008
        maxeval: 100
```

`opts2 = immoptset(@Marquardt, 'Tau',1e-6)` gives

```
opts2 =      tau: 1e-006
            tolg: 0.0001
            tolx: 1e-008
         maxeval: 100
```

Notice that upper/lower case is ignored.

Finally, `opts3 = immoptset(@marquart, 1e-6,'tau')` gives

```
???  Error using ==> immoptset at 48
IMMOPTSET is not prepared for marquart
```

and `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

```
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
```

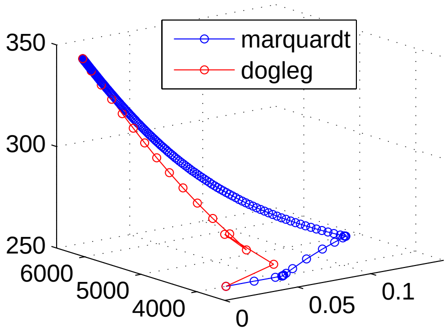
The three values in `err` are identical. This indicates an error, and the second element in index shows that this is in the derivative with respect to x_2.

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.



Problem no. 10

pno	Name	m	n
1	Linear function, full rank	$m \geq n$	variable
2	Linear function, rank 1	$m \geq n$	variable
3	Linear function, rank 1. Zero columns and rows	$m \geq n$	variable $n \geq 3$
4	Rosenbrock	2	2
5	Helical Valley	3	3
6	Powell singular function	4	4
7	Freudenstein and Roth	2	2
8	Bard	15	3
9	Kowalik and Osborne	11	4
10	Meyer	16	3
11	Watson	31	$2 \leq n \leq 31$
12	Box 3-dimensional	$m \geq 3$	3
13	Jennrich and Sampson	$m \geq 2$	2
14	Brown and Dennis	$m \geq 4$	4
15	Chebyquad	$m \geq n$	variable
16	Brown almost linear	$m = n$	variable
17	Osborne 1	33	5
18	Exponential fit	45	4
19	Exponential fit, separated	45	2
20	Modified Meyer	16	3
21	Separated Meyer	16	2
22	Exp and squares	1	variable

A typical call is `[par, x0, tau0, delta0] = uctpget(pno, m, n)`

Input parameters

`pno`	Problem number. Integer in the range $[1, 22]$.
`m,n`	Number of components in the vectors $\boldsymbol{r}(\boldsymbol{x})$ when `pno` ≤ 21 and $\boldsymbol{x}$, 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. (`NaN`s if m or n is free). For some problems `par` has more fields.
`x0`	Standard starting point.
`tau0`	Standard value for `'tau'` in the Marquardt-type functions of the toolbox.
`delta0`	Standard value for `'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 `uctpget`. If the problem is "born" as a least squares problem, and `par.pt` is changed to 1 before the call, then `uctpval` returns the function value $f(\boldsymbol{x})$ and the gradient $\boldsymbol{\nabla} f(\boldsymbol{x})$ as defined by

$$f(\boldsymbol{x}) \;=\; \tfrac{1}{2}\sum_{i=1}^{m} r_i(\boldsymbol{x})^2 \;=\; \boldsymbol{r}(\boldsymbol{x})^T\boldsymbol{r}(\boldsymbol{x}) \;, \qquad \boldsymbol{\nabla} f(\boldsymbol{x}) \;=\; \boldsymbol{J}(\boldsymbol{x})^T\boldsymbol{r}(\boldsymbol{x}) \;.$$

Typical calls are

```
f = uctpval(x, par)
[f, Df] = uctpval(x, par)
[f, Df, D2f] = uctpval(x, par)
```

Input parameters

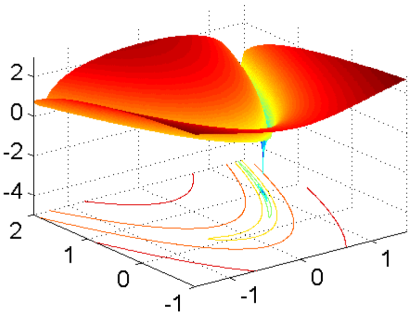
`x`	Argument vector $\boldsymbol{x}$.
`par`	Struct defining the problem, cf uctpget.

Output parameters

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

3.3.3. Example

Suppose that we want to compare marquardt and ucminf applied to Rosenbrock's problem with the starting point $\boldsymbol{x}_0 = (-1.2,\ 1)$ and demand that the gradient be smaller than 10^{-6}. This can be done as follows.



```
[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);
```

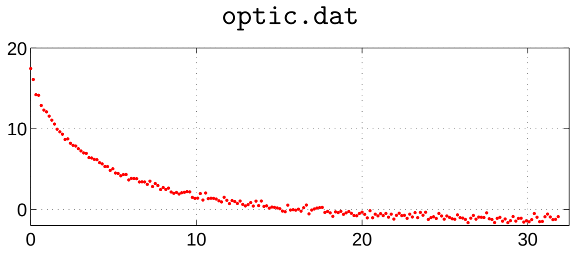
Both `xm` and `xu` are close to the solution $\hat{\boldsymbol{x}} = (1,\ 1)$, and

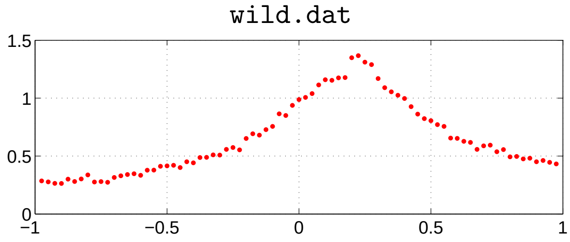
```
infom = 4.91e-16  1.43e-08  9.47e-06  1.23e-06  23  1
infou = 1.45e-17  6.38e-08  6.99e-07  35  39  1
```

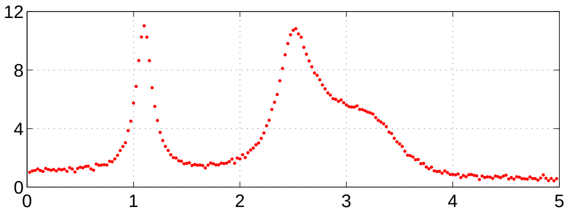
show that in both cases the desired accuracy was obtained after respectively 24 evaluations of $\boldsymbol{r}$ and $\boldsymbol{J}$ and 39 evaluations of f and $\boldsymbol{\nabla} f$.

3.3.4. Data sets

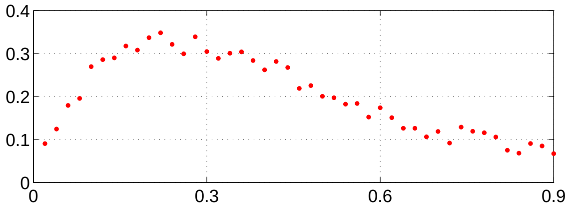
The toolbox contains a number of data sets, which can be used to check a data fitting algorithm.



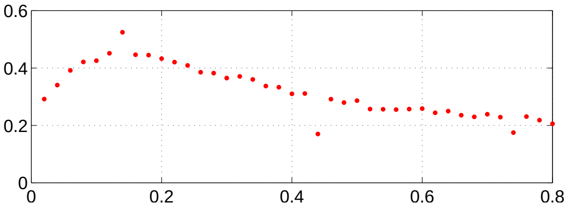




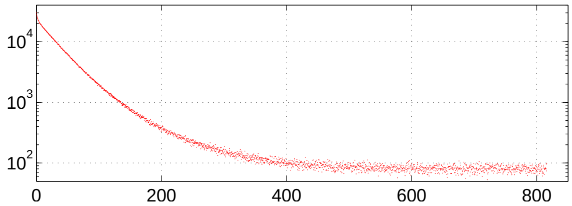
peaks.dat



efit1.dat



efit2.dat



osl.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 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 γ in nonlinhuber has been changed from ∞ to $\|\boldsymbol{r}(\boldsymbol{x}_0)\|_\infty$.

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.

Version 1.6. November 2006

- 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

[1] L. Eldén, L. Wittmemeyer-Koch, H.B. Nielsen, *Introduction to Numerical Computation.* Studentlitteratur (2004).

[2] G. Golub, V. Pereyra, *Separable nonlinear least squares: the variable projection method and its applications.* Inverse Problems **19**, pp R1–R26, (2003).

[3] K. Madsen, H.B. Nielsen, *Finite Algorithms for Robust Linear Regression*, BIT 30 (1990), 682–699.

[4] K. Madsen, H.B. Nielsen, *Introduction to Optimization and Data Fitting.* DTU Informatics (2010). Available from http://www.imm.dtu.dk/pubdb/p.php?5938

[5] K. Madsen, H.B. Nielsen and M.C. Pınar, *Bound Constrained Quadratic Programming via Piecewise Quadratic Functions*, Math. Programming (1999).

[6] H.B. Nielsen, *Checking Gradients.* IMM (2000), 12 pages. Available from http://www.imm.dtu.dk/∼hbn/Software/checkgrad.ps

[7] H.B. Nielsen, *Implementation of a Finite Algorithm for Linear L1 Estimation.* Report NI 91-01, Institute for Numerical Analysis, DTU. (1991). 49 pages.

[8] H.B. Nielsen, *SPLPAK – Pascal and Fortran77 Subprograms for Cubic Splines.* Report NI 91-06, Institute for Numerical Analysis, DTU. (1991). 28 pages.

[9] H. B. Nielsen, *Computing a Minimizer of a Piecewise Quadratic – Implementation.* Report IMM-REP-1998-14, IMM, DTU. (1998), 31 pages. Available from http://www.imm.dtu.dk/∼hbn/publ/TR9814.ps

[10] H. B. Nielsen, *Damping Parameter in Marquardt's Method.* Report IMM-REP-1999-05, IMM, DTU. (1999), 31 pages. Available from http://www.imm.dtu.dk/ hbn/publ/TR9905.ps

[11] H. B. Nielsen, *Separable NonLinear Least Squares.* Report IMM-REP-2000-01, IMM, DTU. (2000), 16 pages. Available from http://www.imm.dtu.dk/ hbn/publ/TR0001.ps

[12] H. B. Nielsen, *Multi-Exponential Fitting of Low-Field* ^1H *NMR Data.* Report IMM-REP-2000-03, IMM, DTU. (2000), 30 pages. Available from http://www.imm.dtu.dk/ hbn/publ/TR0003.ps

[13] H.B. Nielsen, *UCTP – Test Problems for Unconstrained Optimization.* Report IMM-REP-2000-17, IMM, DTU. (2000), 19 pages. Available from http://www.imm.dtu.dk/ hbn/publ/TR0017.ps

[14] H.B. Nielsen, *UCMINF – an Algorithm for Unconstrained, Nonlinear Optimization.* Report IMM-REP-2000-19, IMM, DTU. (2000), 24 pages. Available from http://www.imm.dtu.dk/ hbn/publ/TR0019.ps