MATLAB®

Optimization Toolbox

Computation

Visualization

Programming

User's Guide

Version 5

How to Contact The MathWorks:

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc.
24 Prime Park Way
Natick, MA 01760-1500 Mail

http://www.mathworks.com Web
ftp.mathworks.com Anonymous FTP server
comp.soft-sys.matlab Newsgroup

support@mathworks.com Technical support
suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
subscribe@mathworks.com Subscribing user registration
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information

Optimization Toolbox User's Guide

Printing History: November 1990 First printing
December 1996 Second printing (for MATLAB 5)
May 1997 (online version)

Contents

1 Tutorial

2

3 Reference

Tutorial

Introduction

Optimization concerns the minimization or maximization of functions. The Optimization Toolbox consists of functions that perform minimization (or maximization) on general nonlinear functions. Functions for nonlinear equation solving and least-squares (data-fitting) problems are also provided.

The functions available for minimization are

Table 1-1: Minimization

Type	Notation	Syntax
Scalar Minimization	$\min_a f(a)$ such that $a_1 < a < a_2$	a = fmin('f', a1, a2)
Unconstrained Minimization	$\min_x f(x)$	x = fminu('f', x0) x = fmins('f', x0)
Linear Programming	$\min_x c^T x$ such that $Ax \le b$	x = lp(c, A, b)
Quadratic Programming	$\min_x \frac{1}{2}x^T H x + c^T x$ such that $Ax \le b$	x = qp(H, c, A, b)
Constrained Minimization	$\min_x f(x)$ such that $G(x) \le 0$	x = constr('fG', x0)
Goal Attainment	$\min_{x,\gamma} \gamma$ such that $F(x) - w\gamma \le$ goal	x = attgoal('F', x, goal, w)
Minimax	$\min_x \max_{\{F_i\}} \{F_i(x)\}$ such that $G(x) \le 0$	x = minimax('FG', x0)
Semi-infinite Minimization	$\min_x f(x)$ such that $Gx \le 0$, $K(x, w) \le 0$ for all w	x = seminf('fGK', n, x0)

The functions available for equation solving are

Table 1-2: Equation Solving

Type	Notation	Syntax
Linear Equations	$Ax = b$, n equations, n variables	`x = A\b;`
Nonlinear Equation of One Variable	$f(a) = 0$	`a = fzero('f',a0)`
Nonlinear Equations	$F(x) = 0$, n equations, n variables	`x = fsolve('F',x0)`

The functions available for solving least-squares or data-fitting problems are

Table 1-3: Least-Squares (Curve Fitting)

Type	Notation	Syntax
Linear Least Squares	$\min_x \|Ax-b\|_2^2$, m equations, n variables	`x = A\b;`
Nonnegative Linear Least Squares	$\min_x \|Ax-b\|_2^2$ such that $x \geq 0$	`x = nnls(A,b)`
Constrained Linear Least Squares	$\min_x \|Ax-b\|_2^2$ such that $Cx \leq d$	`x = conls(A,b,C,d)`
Nonlinear Least Squares	$\min_x \frac{1}{2}\|F(x)\|_2^2 = \frac{1}{2}\sum_i F_i(x)^2$	`x = leastsq('F',x0)`
Nonlinear Curve Fitting	$\min_x \frac{1}{2}\|F(x,xdata)-ydata\|_2^2$	`x = curvefit ...` `('F',x0,xdata,ydata)`

Notation

Upper-case letters such as A are used to denote matrices. Lower-case letters such as x are used to denote vectors, except where noted that it is a scalar (in the table above, we use a to denote a scalar in the description of fmin).

For functions, the notation differs slightly to follow the usual conventions in optimization. For vector functions, we use an upper-case letter such as F in F(x). A function that returns a scalar value is denoted with a lower-case letter such as f in f(x).

Most of these routines require the definition of an M-file containing the function to be minimized. Alternatively, a string variable containing a MATLAB expression, with x representing the independent variables, can be used. Maximization is achieved by supplying the routines with –f, where f is the function being optimized.

Optional arguments to the routines place bounds on the variables and change optimization parameters. Default optimization parameters are used extensively but can be changed through an additional argument, options.

Gradients are calculated using an adaptive finite difference method unless they are supplied in a function. Parameters can be passed directly to functions, avoiding the need for global variables.

The Optimization Toolbox routines offer a choice of algorithms and line search strategies. The principal algorithms for unconstrained minimization are the Nelder-Mead simplex search method and the BFGS quasi-Newton method. For constrained minimization, minimax, goal attainment, and semi-infinite optimization, variations of Sequential Quadratic Programming are used. Nonlinear least squares problems use the Gauss-Newton and Levenberg-Marquardt methods.

A choice of line search strategy is given for unconstrained minimization and nonlinear least squares problems. The line search strategies use safeguarded cubic and quadratic interpolation and extrapolation methods.

Installation

Instructions for installing toolboxes are found in the section entitled "Installing Toolboxes" in the computer-specific section of Using MATLAB. On some systems, the Optimization Toolbox may be installed already. It should be located in the directory named `optim` in the toolbox directory.

Examples

The Optimization Toolbox is presented through a tutorial that closely follows the first demonstration in the M-file `optdemo`. The functions `fminu` and `constr` are discussed in detail. The other optimization routines `attgoal`, `minimax`, `leastsq`, `fsolve`, and `seminf` are used in a nearly identical manner, with differences only in the problem formulation and the termination criteria.

Unconstrained Example

Consider the problem of finding a set of values $[x_1, x_2]$ that solves

$$\underset{x}{\text{minimize}} \quad f(x) = e^{x_1}(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1) \qquad (1\text{-}1)$$

To solve this two-dimensional problem, write an M-file that returns the function value. Then, invoke the unconstrained minimization routine `fminu`.

Step 1: Write an M-file fun.m:

```
function f = fun(x)
f = exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
```

Step 2: Invoke optimization routine:

```
x0 = [-1,1];        % Starting guess
x = fminu('fun',x0)
```

After 36 function evaluations, this produces the solution:

```
x =
    0.5000   -1.0000
```

You can evaluate the function at the solution `x`:

```
fun(x)
ans =
    1.3029e-10
```

When there exists more than one local minimum, the initial guess for the vector $[x_1, x_2]$ affects both the number of function evaluations and the value of the solution point. In the example above, `x0` is initialized to `[-1,1]`.

The variable options can be passed to fminu to change characteristics of the optimization solution procedure, as in

```
x = fminu('fun',x0,options);
```

options is a vector that contains values for termination tolerances and algorithm choices. The first element of options controls the amount of output displayed during the optimization cycle for most of the optimization functions. Setting this element to 1 causes a tabular display of the function values and convergence information. The second and third elements of options establish termination criteria. Other elements in options set finite difference perturbation levels, select algorithms, and set the maximum number of function evaluations. This and other calling syntaxes are discussed more fully in later sections of this tutorial and in the References chapter.

Constrained Example

If inequality constraints are added to Eq. 1-1, the resulting problem may be solved by the constr function. For example, if you want to find x that solves

$$\underset{x}{\text{minimize}} \quad f(x) = e^{x_1}(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1)$$

subject to the constraints:

$$\begin{aligned} x_1x_2 - x_1 - x_2 &\le -1.5 \\ x_1x_2 &\ge -10 \end{aligned} \qquad (1\text{-}2)$$

The original M-file is modified to return both the objective function and the constraints. The constrained optimizer, constr, is then invoked. Because constr expects the constraints to be written in the form $G(x) \le 0$, you must rewrite your constraints in the form

$$\begin{aligned} x_1x_2 - x_1 - x_2 + 1.5 &\le 0 \\ -x_1x_2 - 10 &\le 0 \end{aligned} \qquad (1\text{-}3)$$

Step 1: Write an M-file fun.m for the objective function and constraints:

```
function [f,g] = fun(x)
f = exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
g(1,1) = 1.5 + x(1)*x(2) - x(1) - x(2);        % Constraints
g(2,1) = -x(1)*x(2) - 10;
```

Step 2: Invoke constrained optimization routine:

```
x0 = [-1,1];                % Make a starting guess at the solution
x = constr('fun',x0)
```

After 29 function calls, the solution produced is

```
x =
   -9.5474    1.0474
```

We can evaluate the functions and constraints at the solution

```
[f,g] = fun(x)
f =
    0.0236
g =
    1.0e-15 *
   -0.8882
         0
```

Note that both constraint values are less than or equal to zero, that is, x satisfies $G(x) \leq 0$.

Constrained Example with Bounds

The variables in x can be restricted to certain limits by specifying simple bound constraints to the constrained optimizer function. For `constr`, the command

```
x = constr('fun',x0,options,vlb,vub);
```

limits x to be within the range `vlb <=x <= vub`.

To restrict x in Eq. 1-2 to be greater than zero (i.e., $x_1 \geq 0$, $x_2 \geq 0$), use the commands:

```
x0 = [–1, 1];          % Make a starting guess at the solution
options = [ ];         % Use default options
vlb = [0, 0];          % Set lower bounds
vub = [ ];             % No upper bounds
x = constr('fun',x0,options,vlb,vub)
```

Note that to pass in the lower bounds as the fourth argument to `constr`, you must specify a value for the third argument `options`. In this example, we specified `[]` to use the default values for `options`.

After 10 function evaluations, the solution produced is

```
x =
         0    1.5000
[f,g] = fun(x)
f =
         8.5000
g =
         0
       –10
```

In the above example, there were no upper bounds on x. Therefore, `vub` was set to an empty matrix. Alternatively, the upper-bound argument could have been omitted by using the command

```
x = constr('fun',x0,options,vlb)
```

When `vlb` or `vub` contains fewer elements than x, only the first corresponding elements in x are bounded. Alternatively, bounds can be expressed using linear inequality constraints. This alternative may be more appropriate when there are only a few bounded variables, for example,

Upper Bound: $x_i \leq U_B$ is written as: $x_i - U_B \leq 0$

Lower Bound: $x_i \geq L_B$ is written as: $-x_i + L_B \leq 0$

Note that the number of function evaluations to find the solution is reduced since we further restricted the search space. Fewer function evaluations are usually taken when a problem has more constraints and bound limitations because the optimization makes better decisions regarding step-size and regions of feasibility than in the unconstrained case. It is, therefore, good prac-

tice to bound and constrain problems, where possible, to promote fast convergence to a solution.

Constrained Example with Gradients

Ordinarily the minimization routines use numerical gradients calculated by finite difference approximation. This procedure systematically perturbs each of the variables in order to calculate function and constraint partial derivatives. Alternatively, you can provide a function to compute partial derivatives analytically. Typically, the problem is solved more accurately and efficiently if such a function is provided.

To solve the Eq. 1-2 using analytically determined gradients, do the following:

Step 1: Write an M-file for objective function and constraints:

```
function [f,g] = fun(x)
f = exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
g(1) = 1.5 + x(1) * x(2) – x(1) – x(2);     %Constraints
g(2) = –x(1) * x(2)–10;
```

Step 2: Write an M-file for the gradients of the objective function and constraints:

```
function [df,dg] = grad(x)
% Gradient of the objective function
t = exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
df = [ t + exp(x(1)) * (8*x(1) + 4*x(2)),
     exp(x(1))*(4*x(1)+4*x(2)+2)];
% Gradient of the constraints
dg = [x(2)–1, –x(2);
       x(1)–1, –x(1)];
```

Step 3: Invoke constrained optimization routine:

```
x0 = [–1,1];                % Starting guess
options = [ ];              % Use default options
vlb = [ ]; vub = [ ];       % No upper or lower bounds
x = constr('fun',x0,options,vlb,vub,'grad')
```

df contains the partial derivatives of the objective function, f returned by fun(x), with respect to each of the elements in x:

$$\frac{\partial f}{\partial x} = \begin{bmatrix} e^{x_1}(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1) + e^{x_1}(8x_1 + 4x_2) \\ \\ e^{x_1}(4x_1 + 4x_2 + 2) \end{bmatrix} \tag{1-4}$$

The columns of dg contain the partial derivatives for each respective constraint (i.e., the i th column of dg is the partial derivative of the i th constraint with respect to x). So in the above example, dg is

$$\begin{bmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_2}{\partial x_1} \\ \\ \frac{\partial g_1}{\partial x_2} & \frac{\partial g_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} x_2 - 1 & -x_2 \\ \\ x_1 - 1 & -x_1 \end{bmatrix} \tag{1-5}$$

The arguments vlb and vub place lower and upper bounds on the independent variables in x. In this example they are only used for syntactic purposes to give the correct number of right-hand arguments to specify the gradient function name.

After 11 function and gradient evaluations, the solution produced is

```
x =
    -9.5474    1.0474
[f,g] = fun(x)
f =
     0.0236
g =
     1.0e-14 *
     0.1110
    -0.1776
```

Gradient Check: Analytic Versus Numeric

When analytically determined gradients are provided, you can compare the supplied gradients with a set calculated by finite difference evaluation. This is particularly useful for detecting mistakes in either the objective function or the gradient function formulation.

If such gradient checks are desired, initialize `options(9)` to the value 1. The first cycle of the optimization checks the analytically determined gradients. If they do not match within a given tolerance, a warning message indicates the discrepancy and gives an option to abort the optimization or to continue.

Maximization

The optimization functions `fmin`, `fmins`, `fminu`, `constr`, `attgoal`, `minimax`, and `leastsq` all perform minimization of the objective function, $f(x)$. Maximization is achieved by supplying the routines with $-f(x)$. Similarly, to achieve maximization for `qp` supply $-H$ and $-c$, and for `lp` supply $-c$.

Greater than Zero Constraints

The Optimization Toolbox assumes constraints are of the form $G_i(x) \leq 0$. Greater than zero constraints are expressed as less than zero constraints by multiplying them by –1. For example, a constraint of the form $G_i(x) \geq 0$ is equivalent to the constraint $-G_i(x) \leq 0$; a constraint of the form $G_i(x) \geq b$ is equivalent to the constraint $-G_i(x) + b \leq 0$.

Equality Constrained Example

For routines that permit equality constraints, these equality constraints must be expressed in the first elements of the vector of constraint values `g`. Also, `options(13)` must be initialized with the number of equality constraints. For example, to add the constraint $x_1 + x_2 = 1$ to Eq. 1-2, rewrite it as $x_1 + x_2 - 1 = 0$ and then,

Step 1: Write an M-file fun.m:

```
function [f,g] = fun(x)
f = exp(x(1)) * (4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
g(1) = x(1) + x(2) –1;                    %Equality constraints first
g(2) = 1.5 + x(1) * x(2) – x(1) – x(2);   %Inequality
g(3) = –x(1) * x(2) – 10;                 %  constraints
```

Step 2: Invoke constrained optimization routine:

```
x0 = [–1, 1];              % Make a starting guess at the solution
options(13) = 1;           % Specify one equality constraint
x = constr('fun', x0, options)
```

After 22 function evaluations, the solution produced is

```
x =
      –2.7016      3.7016
[f, g] = fun(x)
f =
      1.6775
g =
      –0.0000      –9.5000      0.0000
```

Note that `g(1)` is equal to 0 within the default tolerance and that `g(2)` and `g(3)` are less than or equal to zero as desired.

Additional Arguments: Avoiding Global Variables

Parameters that would otherwise have to be declared as global can be passed directly to M-file functions using additional arguments at the end of the calling sequence.

For example, entering a number of variables at the end of the call to `fsolve`

```
fsolve('fun', x0, options, 'grad', p1, p2, ... )
```

passes the arguments directly to the functions `fun` and `grad` when they are called,

```
f = fun(x, p1, p2, ... )
df = grad(x, p1, p2 ... )
```

Consider, for example, finding zeros of the function `ellipj(u, m)`. The function needs parameter `m` as well as input `u`. To look for a zero near `u = 3`, for `m = 0.5`

```
m = 0.5;
x = fsolve('ellipj', 3, [ ], [ ], m)
```

returns

```
x =
    3.0781
```

Then, solve for the function `ellipj`.

```
f = ellipj(x,m)
f =
    1.158e-11;
```

The empty matrices in the call to `fsolve` imply that default options are used and that analytic gradients are not provided.

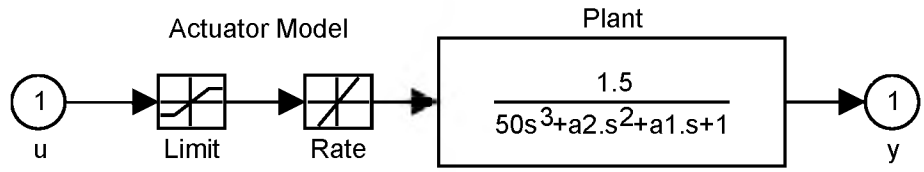
Multiobjective Examples

The previous examples involved problems with a single objective function. This section demonstrates solving problems with multiobjective functions using `lsqnonlin`, `minimax` and `attgoal`. Included is an example of how to optimize parameters in a SIMULINK model.

SIMULINK Example

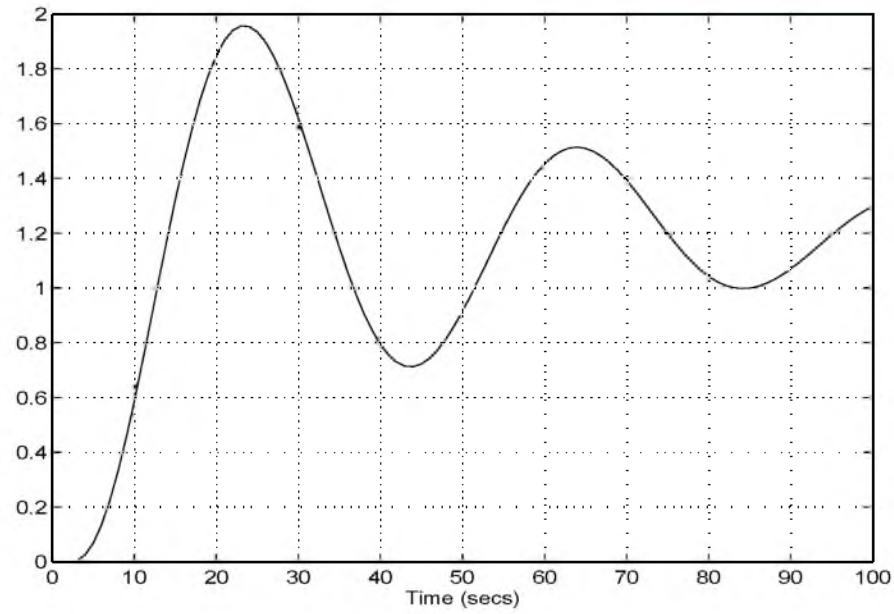
Say you want to optimize the control parameters in the SIMULINK model `optsim.mdl`. (This model can be found in the Optimization Toolbox directory. Note that SIMULINK must be installed on your system to load this model.) The model includes a nonlinear process plant modeled as a SIMULINK block diagram shown in Fig. 1-1.

Figure 1-1: Plant with Actuator Saturation



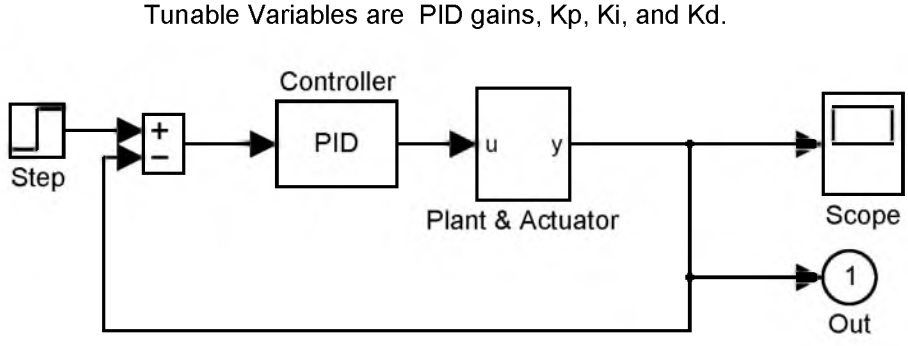
The plant is an under-damped third-order model with actuator limits. The actuator limits are a saturation limit and a slew rate limit. The actuator saturation limit cuts off input values greater than 2 units or less than –2 units. The slew rate limit of the actuator is 0.8 units/sec. The open-loop response of the system to a step input is shown in Fig. 1-2.

Figure 1-2: Open-Loop Response



The problem is to design a feedback control law that tracks a unit step input to the system. The closed-loop plant is entered in terms of the blocks where the plant and actuator have been placed in a hierarchical Subsystem block. A Scope block displays output trajectories during the design process. See Fig. 1-3.

Figure 1-3: Closed-Loop Model



One way to solve this problem is to minimize the error between the output and the input signal. The variables are the parameters of the PID controller. If you only need to minimize the error at one time unit, it would be a single objective function. But the goal is to minimize the error for all time steps from 0 to 100, thus producing a multiobjective function (one function for each time step).

The routine `lsqnonlin` is used to perform a least squares fit on the tracking of the output. This is defined via a MATLAB function in the file `trackls.m` shown below that defines the error signal. The error signal is `yout`, the output computed by calling `sim` minus the input signal 1.

The function `trackls` must run the simulation. When the simulation begins, SIMULINK assumes the values it needs are, by default, in the base workspace. Use the `assignin` command to get your input values, the variables you are optimizing, from the calling workspace to the base workspace.

After choosing a solver using the `simset` function, the simulation is run using `sim`. The simulation is performed using a fixed-step fifth-order method to 100 seconds. When the simulation completes, the variables `tout`, `xout`, and `yout` are now in the calling workspace (that is, the workspace of `trackls`). The Outport block is needed in the block diagram model for `yout` to be nonempty after the simulation.

Step 1: Write an M-file tracklsq.m:

```
function f = tracklsq(pid)
assignin('base','Kp', pid(1)); % Move variables to base workspace
assignin('base','Ki', pid(2));
assignin('base','Kd', pid(3));
opt = simset('solver','ode5'); % Choose solver
[tout,xout,yout] = sim('optsim',[0 100],opt);
f = yout-1;                    % Compute error signal
```

Step 2: Invoke constrained optimization routine:

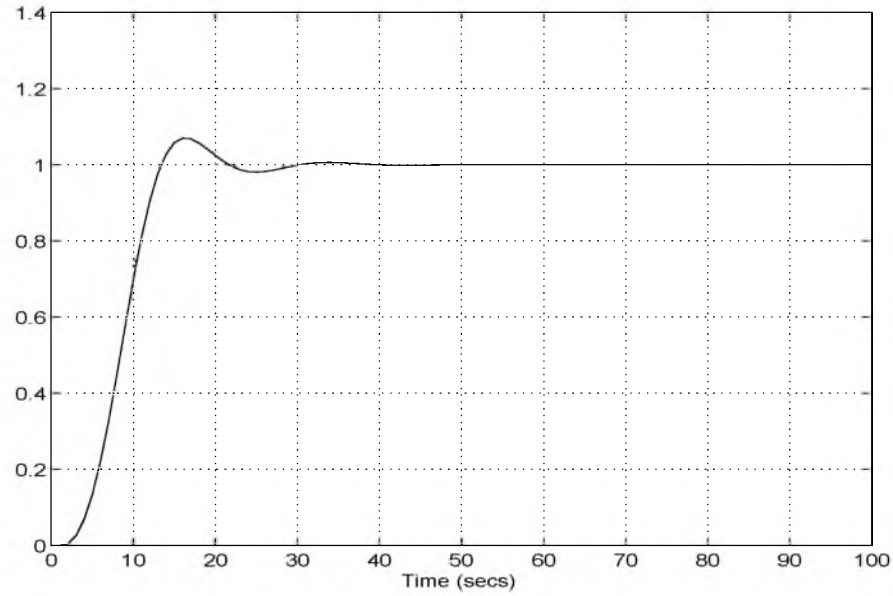
```
pid0 = [0.63 0.0504 1.9688] % Set initial values
options = foptions;
options = [1,0.1,0.1];
pid = leastsq('tracklsq',pid0,options)
```

The vector `options` passed to `leastsq` defines the criteria and display characteristics. In this case you ask for output and give termination tolerances for the step and objective function on the order of 0.1. The optimization gives the solution for the Proportional, Integral, and Derivative (Kp, Ki, Kd) gains of the controller after 47 function evaluations

```
f-COUNT      RESID     STEP-SIZE      GRAD/SD        LAMBDA
      4    16.8341              1        -8.94
     11    13.5356           1.44         3.83       12.5097
     17    12.7714          0.861      -0.0311       12.8168
     27    8.63314            147      0.00927       6.40838
     34    7.53869           1.87       -0.115       2.10188
     41    7.29827           1.24     -0.00355        2.2295
     47    7.25813          0.825     0.000338       2.24491
Optimization Terminated Successfully
pid =
    2.1220    0.2614    9.4222
```

The resulting closed-loop step response is shown in Fig. 1-4.

Figure 1-4: Closed-Loop Response



NOTE The call to sim results in a call to one of the SIMULINK ODE solvers. A choice must be made about the type of solver to use. From the optimization point of view, a fixed-step solver is the best choice if that is sufficient to solve the ODE. In the case of a stiff system, a variable-step method may be required. The numerical solution produced by a variable-step solver, however, is not a smooth function of parameters because of step-size control mechanisms. This lack of smoothness may prevent the optimization routine from converging. This error is not introduced when a fixed-step solver is used. (For a further explanation, see Solving Ordinary Differential Equations I -- Non-stiff Problems, by E.Hairer, S.P. Norsett, G.Wanner, Springer-Verlag, pages 183-184.) The NCD Toolbox is recommended for solving multiobjective optimization problems in conjunction with variable-step solvers in SIMULINK; it provides a special numeric gradient computation that works with SIMULINK and avoids introducing this error.

Another solution approach is to use the `minimax` function. In this case, rather than minimizing the error between the output and the input signal, you minimize the maximum value of the output at any time `t` between 0 and 100. Then in the function `trackmm` the objective function is simply the output returned by the `sim` command. But minimizing the maximum output at all time steps may force the output far below unity for some time steps. To keep the output above 0.95 after the first 20 seconds, add a constraint `yout >= 0.95` from `t=20` to `t=100`. Since constraints must be in the form `g <=0`, the constraint in the function is `g = -yout(20:100)+.95`.

Step 1: Write an M-file trackmm.m:

```
function [f,g] = trackmm(pid)
assignin('base','Kp', pid(1));
assignin('base','Ki', pid(2));
assignin('base','Kd', pid(3));
% Compute function value
opt = simset('solver','ode5');
[tout,xout,yout] = sim('optsim2',[0 100],opt);
f = yout;
g = -yout(20:100)+.95;  % Compute constraints
```

Step 2: Invoke constrained optimization routine:

```
pid0 = [0.63 0.0504 1.9688] % Set initial values
options = foptions;
options = [1,0.1,0.1];
pid = minimax('trackmm',pid0,options)
```

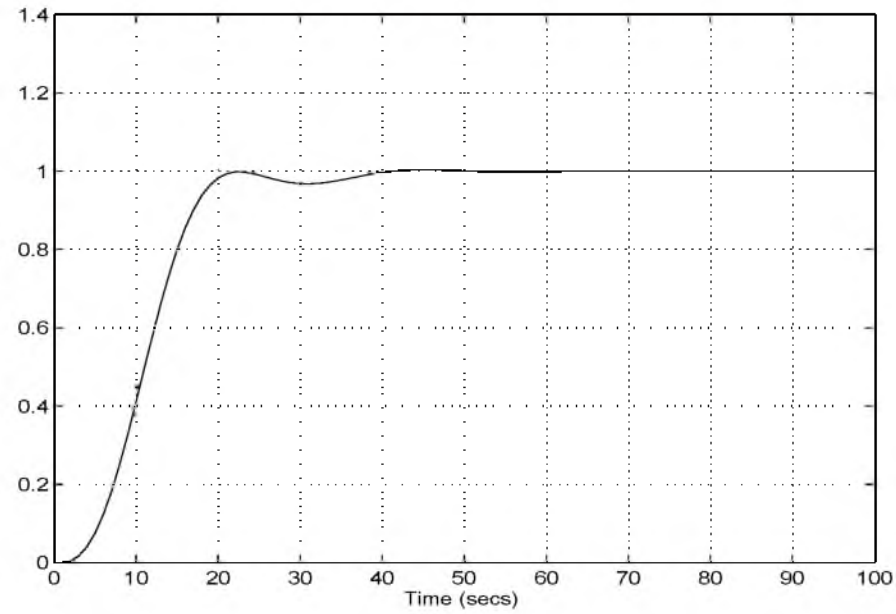
resulting in

```
f-COUNT      MAX{g}          STEP   Procedures
     5      0.984114            1
    10       1.52067            1   Hessian modified twice;
infeasible
    15       1.70044            1
    20       1.27149            1   Hessian modified
    25       1.12734            1
    30       1.03251            1   Hessian modified
    31       1.00352            1   Hessian modified
Optimization Converged Successfully
Active Constraints:
    126
    127
pid =
     1.3415      0.1756      6.9744
```

The last value shown in the MAX{g} column of the output shows the maximum value for all the time steps is 1.00352 (the initial value in this column is smaller, but the g constraints are not satisfied at the initial point). The closed loop response with this result is shown in Fig. 1-5.

This solution differs from the lsqnonlin solution as you are solving different problems.

Figure 1-5: Closed-Loop Response



Signal Processing Example

Consider designing a linear-phase FIR (Finite Impulse Response) filter. The problem is to design a lowpass filter with magnitude one at all frequencies between 0 and 0.1 Hz and magnitude zero between 0.15 and 0.5 Hz.

The frequency response H(f) for such a filter is defined by

$$\begin{aligned} H(f) &= \sum_{n=0}^{M-1} h(n) e^{-j2\pi fn} \\ &= A(f) e^{-j2\pi fM} \\ A(f) &= \sum_{n=0} a(n) \cos(2\pi fn) \end{aligned} \tag{1-6}$$

where A(f) is the magnitude of the frequency response. One solution is to apply a goal attainment method to the magnitude of the frequency response. Given a function that computes the magnitude, the function `attgoal` will attempt to

vary the magnitude coefficients a(n) until the magnitude response matches the desired response within some tolerance. The function that computes the magnitude response is given in `filtmin.m`. This function takes `a`, the magnitude function coefficients, and `w`, the discretization of the frequency domain we are interested in.

To set up a goal attainment problem, you must specify the `goal` and `weights` for the problem. For frequencies between 0 and 0.1, the goal is one. For frequencies between 0.15 and 0.5, the goal is zero. Frequencies between 0.1 and 0.15 are not specified so no goals or weights are needed in this range.

This information is stored in the variable `goal` passed to `attgoal`. The length of `goal` is the same as the length returned by the function `filtmin`. So that the goals are equally satisfied, usually `weight` would be set to `abs(goal)`. However, since some of the goals are zero, the effect of using `weight = abs(goal)` will force the objectives with `weight` 0 to be satisfied as hard constraints, and the objectives with weight 1 possibly to be underattained (see "The Goal Attainment Method" section of the Introduction to Algorithms chapter). Because all the goals are close in magnitude, using a weight of unity for all goals will give them equal priority. (Using `abs(goal)` for the weights is more important when the magnitude of `goal` differs more significantly.) Also, setting `options(15)=length(goal)` specifies that each objective should be as near as possible to its goal value (neither greater nor less than).

Step 1: Write an M-file filtmin.m:

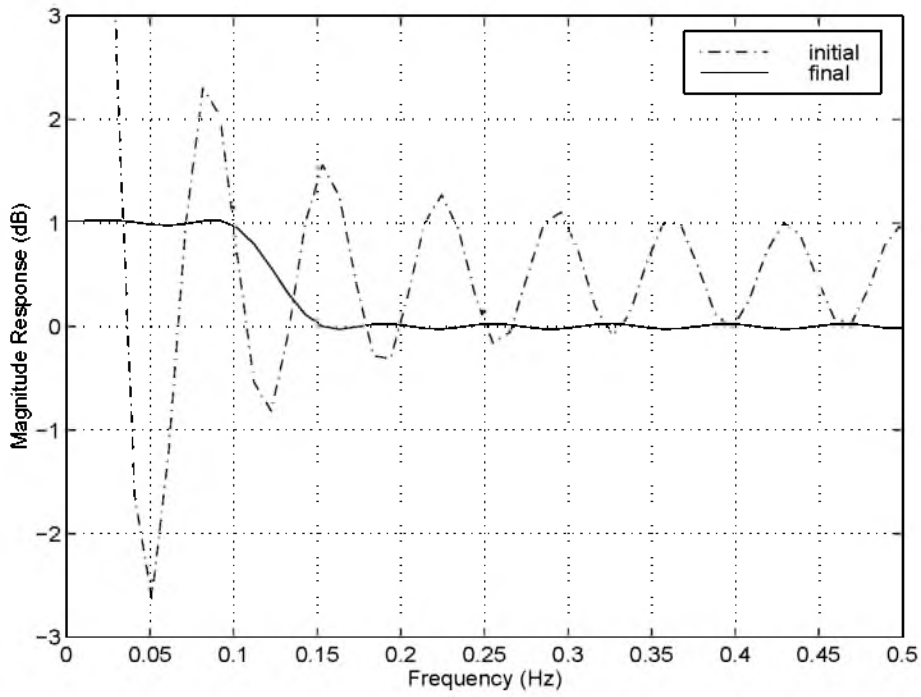
```
function [y, g] = filtmin(a, w)
g = [];          % other constraints
n = length(a);
y = cos(w'*(0:n-1)*2*pi)*a;
```

Step 2: Invoke constrained optimization routine:

```
% Plot with initial coefficients
a0 = ones(15,1);
incr = 50;
w = linspace(0,0.5,incr);

y0 = filtmin(a0,w);
clf, plot(w,y0,'--');
drawnow;

% Set up the goal attainment problem
w1 = linspace(0,0.1,incr) ;
w2 = linspace(0.15,0.5,incr);
w0 = [w1 w2];
goal = [1.0*ones(1,length(w1)) zeros(1,length(w2))];
weight = ones(size(goal));

% Call attgoal
options = foptions;
options(15) = length(goal);
a = attgoal('filtmin',a0,goal,weight,options,[],[],[],w0);

% Plot with the optimized (final) coefficients
y = filtmin(a,w);
hold on, plot(w,y,'r')
axis([0 0.5 -3 3])
```

Compare the magnitude response computed with the initial coefficients and the final coefficients (Fig. 1-6). Note that the remez function in the Signal Processing Toolbox could have been used to design this filter.

Figure 1-6: Magnitude Response with Initial and Final Magnitude Coefficients



Default Parameter Settings

The options vector contains parameters used in the optimization routines. If, on the first call to an optimization routine, the options vector is empty, a set of default parameters is generated. If any of the elements of options are zero, those elements are assigned default values. If options is present and has fewer than 18 elements, the remaining elements assume their default values.

Some of the default options parameters are calculated using factors based on problem size, such as options(14). Other options are used only to return information, such as the value of the function at the last evaluated point in options(8). Options that are used only to return information do not have default values and so N/A is shown as the default value (Not Applicable). Some parameters are dependent on the specific optimization routine and are documented in Chapter 3, Reference. The parameters in the options vector are shown in this table.

Table 1-4: Option Parameters

No.	Function	Default	Description
1	Display	0	Controls amount of output during the optimization cycle. 0 displays no output; 1 displays tabular results; –1 suppresses warning messages.
2	Termination for x	1e–4	Termination criterion that is a measure of the worst case precision required of the independent variables, x. The optimization does not terminate until all termination criteria have been met.
3	Termination for f	1e–4	Termination criterion that is a measure of the precision required of the objective function, f, at the solution.
4	Termination for g	1e–7	Termination criterion used by attgoal, constr, minimax, and seminf that is a measure of the worst case constraint violation that is acceptable.
5	Main Algorithm	0	Main optimization algorithm selection.
6	SD Algorithm	0	Search direction algorithm selection.

Table 1-4: Option Parameters (Continued)

No.	Function	Default	Description
7	Search Algorithm	0	Line search algorithm selection.
8	Function	N/A	Value of the function at the last evaluated point. For `attgoal` and `minimax`, it contains an attainment factor.
9	Gradient Check	0	When set to 1, the analytically supplied gradients are compared with those obtained from a finite difference calculation during the first few iterations. The gradient function must exist when this element is set to 1.
10	Function Count	N/A	Function evaluation counter.
11	Gradient Count	N/A	Number of function gradient evaluations or finite difference gradient calculations.
12	Constraint Count	N/A	Total number of constraint gradient calculations or finite difference gradient calculations.
13	Equality Constraints	0	Number of equality constraints. Equality constraints are placed in the first elements of the variable `g`.
14	Maximum Function Evaluations	100n	Maximum number of function evaluations. This value is set to 100 times `n`, where `n` is the number of independent variables. In `fminsearch` the default is 200 times `n`. In `fminbnd` the default is 500 times `n`.
15	Objectives Used	0	Number of objectives to be as near as possible to the goals. Used by `attgoal`.
16	Minimum Perturbation	1e–8	Minimum change in variables for finite difference gradient calculation. The actual perturbation used is adaptive to increase accuracy of the gradient calculation. It varies between the minimum and maximum perturbation.
17	Maximum Perturbation	0.1	Maximum change in variables for finite difference gradient calculation.

Table 1-4: Option Parameters (Continued)

No.	Function	Default	Description
18	Step-size	N/A	Step-size parameter. Generally on the first iteration this is set conservatively to a value of 1 or less, depending on the derivatives.

As an example, commands that change the first two termination criteria in Eq. 1-2 to 1e–8 are shown below.

Changing the Default Settings Example

```
x0 = [–1, 1];       % Make a starting guess at the solution
options(1) = 1;           % Display intermediate results
options(2) = 1e–8;        % Termination criterion on x
options(3) = 1e–8;        % Termination criterion on fun(x)
x = fminu('fun', x0, options)
```

This yields a solution after 63 function evaluations.

```
x =
     0.5000   –1.0000
fun(x)
ans =
     3.5145e–14
```

Online Help for `options` is available by typing the command `help foptions`. The command `foptions`, when called without arguments, returns the set of default parameters. If `foptions` is given an input vector, it returns the set of default parameters except where the input vector has nonzero values.

Returning the Default Settings

```
opti ons  = f opti ons
opti ons  =
    Col umns  1  t hr ough  7
      0      0. 0001      0. 0001      0. 0000      0      0      0
  Col umns  8  t hr ough  14
      0      0            0            0            0      0      0
   Col umns  15  t hr ough  18
      0      0. 0000     0. 1000      0

opti ons  = f opti ons( [ 0  1e- 2] )
opti ons  =
Col umns  1  t hr ough  7
      0      0. 0100      0. 0001      0. 0000      0      0      0
  Col umns  8  t hr ough  14
      0      0            0            0            0      0      0
   Col umns  15  t hr ough  18
      0      0. 0000     0. 1000      0
```

Output Headings

When opti ons(1) =1 for att goal , const r , cur vef i t , f mi nu, f sol ve, l east sq, mi ni max and semi nf , output is produced in column format. For f mi nu, the column headings are

```
f - COUNT     FUNCTI ON      STEP- SI ZE         GRAD/ SD
```

where

- f - COUNT is the number of function evaluations
- FUNCTI ON is the function value
- STEP- SI ZE is the step size in the search direction
- GRAD/ SD is the gradient of the function along the search direction.

For f sol ve, l east sq and cur vef i t the headings are

```
f - COUNT         RESI D      STEP- SI ZE         GRAD/ SD          LAMBDA
```

where f-COUNT STEP-SIZE and GRAD/SD are the same as for fminu, and

- RESID is the residual (sum-of-squares) of the function
- LAMBDA is the λ_k value defined in the "Least Squares Optimization" section of the Introduction to Algorithms chapter. This value is printed only when the Levenberg-Marquardt method is used.

For constr and seminf the headings are

```
 f-COUNT    FUNCTION          MAX{g}          STEP    Procedures
```

where

- f-COUNT is the number of function evaluations
- FUNCTION is the function value
- MAX{g} is the maximum constraint violation
- STEP is the step size in the search direction
- Procedures are messages about the Hessian update and QP subproblem.

The Procedures messages are discussed in the "Updating the Hessian Matrix" section of the Introduction to Algorithms chapter.

For attgoal and minimax, the headings are the same as for constr except FUNCTION is omitted because MAX{g} gives the maximum goal violation for attgoal and the maximum function value for minimax.

Optimization of String Expressions Instead of M-Files

The routines in the Optimization Toolbox also perform optimization on expressions, avoiding the need to write M-files that define functions. Expressions are placed directly into strings without providing a function as an argument. If the function variable to be evaluated (e.g., fun) contains nonalphanumeric characters (e.g., *,–,+,[), it is evaluated as an expression rather than a function name.

When writing such expressions, the independent variable must always be a lower-case x. An example of using an expression in place of a function argument is

```
x = fminu('sin(x)',1)      % Minimize sin(x) starting at 1
```

Note that this is also equivalent to fminu('sin',1).

A simple squared problem is expressed as

```
x = fmi nu('x(1)^2+x(2)^2',[1;1])
```

which can also be solved with partial derivatives

```
x = fmi nu('x(1)^2+x(2)^2',[1;1],[ ], '[2*x(1); 2*x(2)]')
```

Other examples using this technique follow.

A matrix equation

```
x = fsol ve('x*x*x–[1,2;3,4]', ones(2,2))
```

A least squares problem

```
x = l eastsq('x*x–[3 5;9 10]', eye(2,2))
```

Function parameters have the variable names P1,P2,P3,. . . , which can be used in the expression, for example,

```
x = att goal ('sor t(ei g(P1+P2*x*P3))', zer os(2,2), . . .
[–5,–3,–1],[5, 3, 1], [ ],–4*ones(2),4*ones(2),[ ], A, B, C);
```

solves the problem detailed in Chapter 3, Reference for att goal . Here the function parameters P1, P2, and P3 are set equal to the variables A, B, and C. You cannot enter the variable names directly in the expression using the names A, B, and C because the expression is not evaluated in the base workspace.

When using the routines mi ni max and const r, the objective function and constraints must be named f and g, respectively. For example,

```
x = mi ni max('f = x*x–[1 2;3 4]; g = [ ];', 100*ones(2));
x = const r('f = x(1)^2+x(2)^2; g = x+[1;–2];', [1;1], [1;1]);
```

There are no constraints in the examples above for mi ni max and const r. Therefore, the constraint matrix g is set to the empty matrix.

Partial derivatives are supplied to const r and mi ni max in a similar way by using the variable gf for the function's partial derivatives and gg for the constraint's partial derivatives. An example is in the M-file t ut demo. m.

Practicalities

Optimization problems can take many iterations to converge and can be sensitive to numerical problems such as truncation and round-off error in the calculation of finite difference gradients. Most optimization problems benefit from good starting guesses. This improves the execution efficiency and can help locate the global minimum instead of a local minimum.

Complex problems are best solved by an evolutionary approach whereby a problem with a smaller number of independent variables is solved first. Solutions from lower order problems can generally be used as starting points for higher order problems by using an appropriate mapping.

The use of simpler cost functions and less stringent termination criteria in the early stages of an optimization problem can also reduce computation time. Such an approach often produces superior results by avoiding local minima.

The Optimization Toolbox functions can be applied to a large variety of problems. Used with a little "conventional wisdom," many of the limitations associated with optimization techniques can be overcome. Additionally, problems that are not typically in the standard form can be handled by using an appropriate transformation. Below is a list of typical problems and recommendations for dealing with them:

Problem: The solution does not appear to be a global minimum.

Recommendation: There is no guarantee that you have a global minimum unless your problem is continuous and has only one minimum. Starting the optimization from a number of different starting points may help to locate the global minimum or verify that there is only one minimum. Use different methods, where possible, to verify results.

Problem: The `fmin` function produces warning messages and seems to exhibit slow convergence near the solution.

Recommendation: If you are not supplying analytically determined gradients and the termination criteria are stringent, `fmin` often exhibits slow convergence near the solution due to truncation error in the gradient calculation. Relaxing the termination criteria produces faster, although less accurate, solutions. Changing the finite difference perturbation levels, `options(16:17)`, may increase the accuracy of gradient calculations.

Problem: Sometimes an optimization problem has values of x for which it is impossible to evaluate f and g.

Recommendation: Place bounds on the independent variables or make a penalty function to give a large positive value to f and g when infeasibility is encountered. For gradient calculation the penalty function should be smooth and continuous.

Problem: The function that is being minimized has discontinuities.

Recommendation: The derivation of the underlying method is based upon functions with continuous first and second derivatives. Some success may be achieved for some classes of discontinuities when they do not occur near solution points, or if the finite difference parameters are adjusted in order to jump over small discontinuities. The variables options(16) and options(17) control the perturbation levels for x used in the calculation of finite difference gradients. The perturbation, Δx, is always in the range

options(16) < Dx < options(17)

Another option is to smooth the function. For example, the objective function might include a call to an interpolation function to do the smoothing.

Problem: Warning messages are displayed.

Recommendation: This sometimes occurs when termination criteria are overly stringent, or when the problem is particularly sensitive to changes in the independent variables. This usually indicates truncation or round-off errors in the finite difference gradient calculation, or problems in the polynomial interpolation routines. These warnings can usually be ignored because the routines continue to make steps toward the solution point; however, they are often an indication that convergence will take longer than normal. Scaling can sometimes improve the sensitivity of a problem.

Problem: The independent variables, x, only can take on discrete values, for example, integers.

Recommendation: This type of problem occurs commonly when, for example, the variables are the coefficients of a filter that are realized using finite

precision arithmetic or when the independent variables represent materials that are manufactured only in standard amounts.

Although the Optimization Toolbox functions are not explicitly set up to solve discrete problems, the problem can often be solved by first solving an equivalent continuous problem. Discrete variables can be progressively eliminated from the independent variables, which are free to vary.

Eliminate a discrete variable by rounding it up or down to the nearest best discrete value. After eliminating a discrete variable, solve a reduced order problem for the remaining free variables. Having found the solution to the reduced order problem, eliminate another discrete variable and repeat the cycle until all the discrete variables have been eliminated.

`dfildemo` is a demonstration routine that shows how filters with fixed precision coefficients can be designed using this technique.

Problem: The minimization routine appears to enter an infinite loop or returns a solution that does not satisfy the problem constraints.

Recommendation: Your objective, constraint or gradient functions may be returning Inf, NaN, or complex values. The minimization routines expect only real numbers to be returned. Any other values may cause unexpected results. Insert some checking code into the user-supplied functions to verify that only real numbers are returned (use the function `isfinite`).

Problem: You do not get the convergence you expect from the `leastsq` routine.

Recommendation: You may be forming the sum of squares explicitly and returning a scalar value. `leastsq` expects a vector (or matrix) of function values that are squared and summed internally.

Introduction to Algorithms

Parametric Optimization

This chapter provides an introduction to the different optimization problem formulations used in the Optimization Toolbox and describes the algorithms.

Parametric optimization is used to find a set of design parameters, $x = \{x_1, x_2, \ldots, x_n\}$, that can in some way be defined as optimal. In a simple case this may be the minimization or maximization of some system characteristic that is dependent on x. In a more advanced formulation the objective function, f(x), to be minimized or maximized, may be subject to constraints in the form of equality constraints, $G_i(x) = 0 \ (i = 1, \ldots, m_e)$, inequality constraints, $G_i(x) \leq 0 \ (i = m_e + 1, \ldots, m)$, and/or parameter bounds, x_l, x_u.

A General Problem (GP) description is stated as

$$\begin{aligned} &\underset{x \in \Re^n}{\text{minimize}} \quad f(x) \\ &\text{subject to:} \quad G_i x = 0, \qquad i = 1, \ldots, m_e \\ &\qquad\qquad\quad G_i(x) \leq 0, \qquad i = m_e + 1, \ldots, m \\ &\qquad\qquad\quad x_l \leq x \leq x_u \end{aligned} \tag{2-1}$$

where x is the vector of design parameters, ($x \in \Re^n$), f(x) is the objective function that returns a scalar value ($f(x): \Re^n \rightarrow \Re$), and the vector function G(x) returns the values of the equality and inequality constraints evaluated at x ($G(x): \Re^n \rightarrow \Re^m$).

An efficient and accurate solution to this problem is not only dependent on the size of the problem in terms of the number of constraints and design variables but also on characteristics of the objective function and constraints. When both the objective function and the constraints are linear functions of the design variable, the problem is known as a Linear Programming problem (LP). Quadratic Programming (QP) concerns the minimization or maximization of a quadratic objective function that is linearly constrained. For both the LP and QP problems, reliable solution procedures are readily available. More difficult to solve is the Nonlinear Programming (NP) problem in which the objective function and constraints may be nonlinear functions of the design variables. A solution of the NP problem generally requires an iterative procedure to establish a direction of search at each major iteration. This is usually achieved by the solution of an LP, a QP, or an unconstrained sub-problem.

Unconstrained Optimization

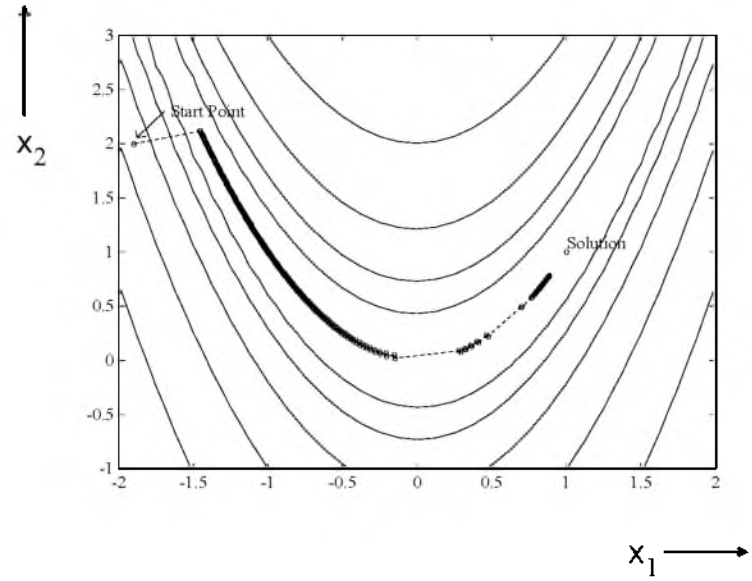
Although a wide spectrum of methods exists for unconstrained optimization, methods can be broadly categorized in terms of the derivative information that is, or is not, used. Search methods that use only function evaluations (e.g., the simplex search of Nelder and Mead [33]) are most suitable for problems that are very nonlinear or have a number of discontinuities. Gradient methods are generally more efficient when the function to be minimized is continuous in its first derivative. Higher order methods, such as Newton's method, are only really suitable when the second order information is readily and easily calculated since calculation of second order information, using numerical differentiation, is computationally expensive.

Gradient methods use information about the slope of the function to dictate a direction of search where the minimum is thought to lie. The simplest of these is the method of steepest descent in which a search is performed in a direction, $-\nabla f(x)$, (where $\nabla f(x)$ is the gradient of the objective function). This method is very inefficient when the function to be minimized has long narrow valleys as, for example, is the case for Rosenbrock's function

$$f(x) = 100(x_1 - x_2^2)^2 + (1 - x_1)^2 \tag{2-2}$$

The minimum of this function is at $x = [1,1]$ where $f(x) = 0$. A contour map of this function is shown in Fig. 2-1, along with the solution path to the minimum for a steepest descent implementation starting at the point [–1.9,2]. The optimization was terminated after 1000 iterations, still a considerable distance from the minimum. The black areas are where the method is continually zig-zagging from one side of the valley to another. Note that towards the center of the plot, a number of larger steps are taken when a point lands exactly at the center of the valley.

Figure 2-1: Steepest Descent Method on Rosenbrock's Function (Eq. 2-2)



This type of function (Eq. 2-2), also known as the banana function, is notorious in unconstrained examples because of the way the curvature bends around the origin. Eq. 2-2 is used throughout this section to illustrate the use of a variety of optimization techniques. The contours have been plotted in exponential increments due to the steepness of the slope surrounding the U-shaped valley.

Quasi-Newton Methods

Of the methods that use gradient information, the most favored are the quasi-Newton methods. These methods build up curvature information at each iteration to formulate a quadratic model problem of the form

$$\min_x \frac{1}{2}x^T Hx + c^T x + b \tag{2-3}$$

where the Hessian matrix, H, is a positive definite symmetric matrix, c is a constant vector, and b is a constant. The optimal solution for this problem occurs when the partial derivatives of x go to zero, i.e.,

$$\nabla f(x^*) = Hx^* + c = 0 \tag{2-4}$$

The optimal solution point, x^*, can be written as

$$x^* = -H^{-1}c \tag{2-5}$$

Newton-type methods (as opposed to quasi-Newton methods) calculate H directly and proceed in a direction of descent using a line search method to locate the minimum after a number of iterations. Calculating H numerically involves a large amount of computation. Quasi-Newton methods avoid this by using the observed behavior of f(x) and $\nabla f(x)$ to build up curvature information to make an approximation to H using an appropriate updating technique.

A large number of Hessian updating methods have been developed. Generally, the formula of Broyden [3], Fletcher [4], Goldfarb [5], and Shanno [6] (BFGS) is thought to be the most effective for use in a general purpose method.

The formula is given by

BFGS

$$H_{k+1} = H_k + \frac{q_k q_k^T}{q_k^T s_k} - \frac{H_k^T s_k^T s_k H_k}{s_k^T H_k s_k} \tag{2-6}$$

where $s_k = x_{k+1} - x_k$

$q_k = \nabla f(x_{k+1}) - \nabla f(x_k)$

As a starting point, H_0 can be set to any symmetric positive definite matrix, for example, the identity matrix I. To avoid the inversion of the Hessian H, you can derive an updating method in which the direct inversion of H is avoided by using a formula that makes an approximation of the inverse Hessian H^{-1} at each update. A well known procedure is the DFP formula of Davidon [7], Fletcher, and Powell [8]. This uses the same formula as the above BFGS method (Eq. 2-6) except that q_k is substituted for s_k.

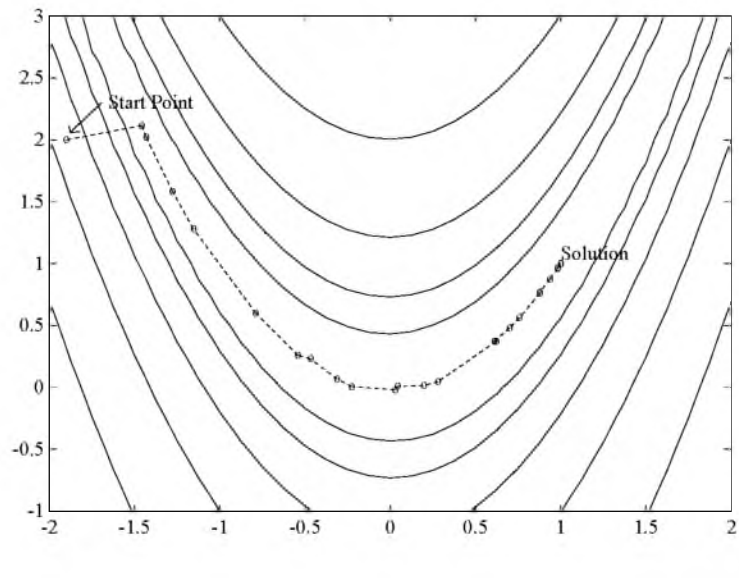
The gradient information is either supplied through analytically calculated gradients, or derived by partial derivatives using a numerical differentiation method via finite differences. This involves perturbing each of the design variables, x, in turn and calculating the rate of change in the objective function.

At each major iteration, k, a line search is performed in the direction

$$d = -H_k^{-1} \cdot \nabla f(x_k) \tag{2-7}$$

The quasi-Newton method is illustrated by the solution path on Rosenbrock's function (Eq. 2-2) in Fig. 2-2. The method is able to follow the shape of the valley and converges to the minimum after 140 function evaluations using only finite difference gradients.

Figure 2-2: BFGS Method on Rosenbrock's Function



Line Search

Most unconstrained and constrained methods use the solution of a sub-problem to yield a search direction in which the solution is estimated to lie. The

minimum along the line formed from this search direction is generally approximated using a search procedure (e.g., Fibonacci, Golden Section) or by a polynomial method involving interpolation or extrapolation (e.g., quadratic, cubic). Polynomial methods approximate a number of points with a univariate polynomial whose minimum can be calculated easily. Interpolation refers to the condition that the minimum is bracketed (i.e., the minimum lies in the area spanned by the available points), whereas extrapolation refers to a minimum located outside the range spanned by the available points. Extrapolation methods are generally considered unreliable for estimating minima for nonlinear functions. However, they are useful for estimating step length when trying to bracket the minimum as shown in the "Line Search Procedures" section. Polynomial interpolation methods are generally the most effective in terms of efficiency when the function to be minimized is continuous. The problem is to find a new iterate x_{k+1} of the form

$$x_{k+1} = x_k + \alpha^* d \tag{2-8}$$

where x_k denotes the current iterate, d the search direction obtained by an appropriate method, and α^* is a scalar step length parameter that is the distance to the minimum.

Quadratic Interpolation

Quadratic interpolation involves a data fit to a univariate function of the form

$$m_q(\alpha) = a\alpha^2 + b\alpha + c \tag{2-9}$$

where an extremum occurs at a step length of

$$\alpha^* = \frac{-b}{2a} \tag{2-10}$$

This point may be a minimum or a maximum. It is a minimum when interpolation is performed (i.e., using a bracketed minimum) or when a is positive. Determination of coefficients, a and b, can be found using any combination of three gradient or function evaluations. It may also be carried out with just two gradient evaluations. The coefficients are determined through the formulation and solution of a linear set of simultaneous equations. Various simplifications in the solution of these equations can be achieved when particular characteristics of the points are used. For example, the first point can generally be taken

as $\alpha = 0$. Other simplifications can be achieved when the points are evenly spaced. A general problem formula is as follows:

Given three unevenly spaced points $\{x_1, x_2, x_3\}$ and their associated function values $\{f(x_1), f(x_2), f(x_3)\}$ the minimum resulting from a second-order fit is given by

Quadratic Interpolation

$$x_{k+1} = \frac{1}{2} \frac{\beta_{23} f(x_1) + \beta_{31} f(x_2) + \beta_{12} f(x_3)}{\gamma_{23} f(x_1) + \gamma_{31} f(x_2) + \gamma_{12} f(x_3)}$$

where

$$\beta_{ij} = x_i^2 - x_j^2$$
$$\gamma_{ij} = x_i - x_j \qquad (2\text{-}11)$$

For interpolation to be performed, as opposed to extrapolation, the minimum must be bracketed so that the points can be arranged to give

$$f(x_2) < f(x_1) \qquad \text{and} \qquad f(x_2) < f(x_3)$$

Cubic Interpolation

Cubic interpolation is useful when gradient information is readily available or when more than three function evaluations have been calculated. It involves a data fit to the univariate function

$$m_c(\alpha) = a\alpha^3 + b\alpha^2 + c\alpha + d \qquad (2\text{-}12)$$

where the local extrema are roots of the quadratic equation

$$3a\alpha^2 + 2b\alpha + c = 0$$

To find the minimum extremum, take the root that gives $6a\alpha + 2b$ as positive. Coefficients a and b can be determined using any combination of four gradient or function evaluations, or alternatively, with just three gradient evaluations.

The coefficients are calculated by the formulation and solution of a linear set of simultaneous equations. A general formula, given two points, $\{x_1, x_2\}$, their corresponding gradients with respect to x, $\{\nabla f(x_1), \nabla f(x_2)\}$, and associated function values, $\{f(x_1), f(x_2)\}$ is

$$x_{k+1} = x_2 - (x_2 - x_1)\frac{\nabla f(x_2) + \beta_2 - \beta_1}{\nabla f(x_2) - \nabla f(x_1) + 2\beta_2}$$

where

$$\beta_1 = \nabla f(x_1) + \nabla f(x_2) - 3\frac{f(x_1) - f(x_2)}{x_1 - x_2} \qquad (2\text{-}13)$$

$$\beta_2 = (\beta_1^2 - \nabla f x_1 \nabla f(x_2))^{1/2}.$$

Quasi-Newton Implementation

A quasi-Newton algorithm is used in `fminu`. The algorithm consists of two phases.

- Determination of a direction of search
- Line search procedure

Implementation details of the two phases are discussed below.

Hessian Update

The direction of search is determined by a choice of either the BFGS (Eq. 2-6) or the DFP method given in the "Quasi-Newton Methods" section (set `options(6) = 1` to select the DFP method.). The Hessian, H, is always maintained to be positive definite so that the direction of search, d, is always in a descent direction. This means that for some arbitrarily small step, α, in the direction, d, the objective function decreases in magnitude. Positive definiteness of H is achieved by ensuring that H is initialized to be positive definite and thereafter $q_k^T s_k$ (from Eq. 2-6) is always positive. The term $q_k^T s_k$ is a product of the line search step length parameter, α_k and a combination of the search direction, d, with past and present gradient evaluations,

$$q_k^T s_k = \alpha_k(\nabla f(x_{k+1})^T d - \nabla f(x_k)^T d) \qquad (2\text{-}14)$$

The condition that $q_k^T s_k$ is positive is always achieved by ensuring that a sufficiently accurate line search is performed. This is because the search direction, d, is a descent direction so that α_k and $-\nabla f(x_k)^T d$ are always positive. Thus, the possible negative term $\nabla f(x_{k+1})^T d$ can be made as small in magnitude as required by increasing the accuracy of the line search.

Line Search Procedures

Two line search strategies are used depending on whether gradient information is readily available or whether it must be calculated using a finite difference method. When gradient information is available, the default is to use a cubic polynomial method. When gradient information is not available, the default is to use a mixed quadratic and cubic polynomial method.

Cubic Polynomial Method

In the proposed cubic polynomial method, a gradient and a function evaluation is made at every iteration, k. At each iteration an update is performed when a new point is found, x_{k+1}, which satisfies the condition that

$$f(x_{k+1}) < f(x_k) \tag{2-15}$$

At each iteration a step, α_k, is attempted to form a new iterate of the form

$$x_{k+1} = x_k + \alpha_k d \tag{2-16}$$

If this step does not satisfy the condition (Eq. 2-15) then α_k is reduced to form a new step, α_{k+1}. The usual method for this reduction is to use bisection (i.e., to continually halve the step length until a reduction is achieved in f(x). However, this procedure is slow when compared to an approach that involves using gradient and function evaluations together with cubic interpolation/extrapolation methods to identify estimates of step length.

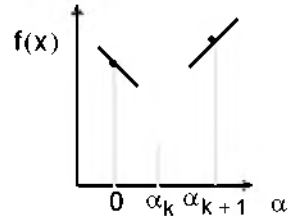
When a point is found that satisfies the condition (Eq. 2-15), an update is performed if $q_k^T s_k$ is positive. If it is not, then further cubic interpolations are performed until the univariate gradient term $\nabla f(x_{k+1})^T d$ is sufficiently small so that $q_k^T s_k$ is positive.

It is usual practice to reset α_k to unity after every iteration. However, note that the quadratic model (Eq. 2-3) is generally only a good one near to the solution point. Therefore, α_k, is modified at each major iteration to compensate for the case when the approximation to the Hessian is monotonically increasing or decreasing. To ensure that, as x_k approaches the solution point, the procedure reverts to a value of α_k close to unity, the values of $q_k^T s_k - \nabla f(x_k)^T d$ and α_{k+1} are used to estimate the closeness to the solution point and thus to control the variation in α_k.

After each update procedure, a step length α_k is attempted, following which a number of scenarios are possible. Consideration of all the possible cases is quite complicated and so they are represented pictorially in Fig. 2-3, where the left-hand point on the graphs represents the point x_k. The slope of the line bisecting each point represents the slope of the univariate gradient, $\nabla f(x_k)^T d$, which is always negative for the left-hand point. The right-hand point is the point x_{k+1} after a step of α_k is taken in the direction d.

Figure 2-3: Cubic Polynomial Line Search Procedures

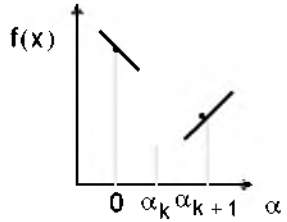
Case 1: $f(x_{k+1}) > f(x_k), \nabla f(x_{k+1})^T d > 0$



Reduce step length.

$$\alpha_{k+1} = \begin{cases} \alpha_c/2 & \text{if } \alpha_k < 0.1 \\ \alpha_c & \text{otherwise} \end{cases}$$

Case 2: $f(x_{k+1}) \le f(x_k), \nabla f(x_{k+1})^T d \ge 0$



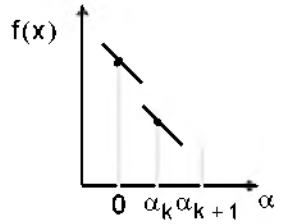
≥ 0 $q_k^T s_k$ < 0

Update H.
Reset d.

Reduce step length.

$\alpha_{k+1} = \min\{1, \alpha_c\}$ $\alpha_{k+1} = 0.9\alpha_c$

Case 3: $f(x_{k+1}) < f(x_k), \nabla f(x_{k+1})^T d < 0$



≥ 0 $q_k^T s_k$ < 0

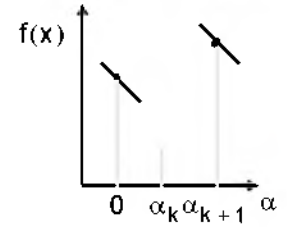
Update H.
Reset d.

Change to steepest descent method temporarily.

$\alpha_{k+1} = \min\{2, p, 1.2\alpha_c\}$ $\alpha_{k+1} = \min\{2, 1.5, \alpha_k\alpha_c\}$

where $p = 1 + q_k^T s_k - \nabla f x_{k+1}{}^T d + \min\{0, \alpha_{k+1}\}$

Case 4: $f(x_{k+1}) \ge f(x_k), \nabla f(x_{k+1})^T d \le 0$



Reduce step length.

$\alpha_{k+1} = \min\{\alpha_c, \alpha_k/2\}$

Cases 1 and 2 show the procedures performed when the value $\nabla f(x_{k+1})^T d$ is positive. Cases 3 and 4 show the procedures performed when the value $\nabla f(x_{k+1})^T d$ is negative. The notation $\min\{a, b, c\}$ refers to the smallest value of the set $\{a, b, c\}$.

At each iteration a cubicly interpolated step length α_c is calculated and then used to adjust the step length parameter α_{k+1}. Occasionally, for very non-linear functions α_c may be negative, in which case α_c is given a value of $2\alpha_k$. The methods for changing the step length have been refined over a period of time by considering a large number of test problems.

Certain robustness measures have also been included so that, even in the case when false gradient information is supplied, a reduction in f(x) can be achieved by taking a negative step. This is done by setting $\alpha_{k+1} = -\alpha_k/2$ when α_k falls below a certain threshold value (e.g., 1e–8). This is important when extremely high precision is required if only finite difference gradients are available.

Mixed Cubic/ Quadratic Polynomial Method

The cubic interpolation/extrapolation method has proved successful for a large number of optimization problems. However, when analytic derivatives are not available, the evaluating finite difference gradients is computationally expensive. Therefore, another interpolation/extrapolation method is implemented so that gradients are not needed at every iteration. The approach in these circumstances, when gradients are not readily available, is to use a quadratic interpolation method. The minimum is generally bracketed using some form of bisection method. This method, however, has the disadvantage that all the available information about the function is not used. For instance, a gradient calculation is always performed at each major iteration for the Hessian update. Therefore, given three points that bracket the minimum, it is possible to use cubic interpolation, which is likely to be more accurate than using quadratic interpolation. Further efficiencies are possible if, instead of using bisection to bracket the minimum, extrapolation methods similar to those used in the cubic polynomial method are used.

Hence, the method that is used in `fminu`, `lsqnonlin`, and `fsolve` is to find three points that bracket the minimum and to use cubic interpolation to estimate the minimum at each line search. The estimation of step length, at each minor iteration, j, is shown in Fig. 2-4 for a number of point combinations. The left-hand point in each graph represents the function value $f(x_1)$ and univariate gra-

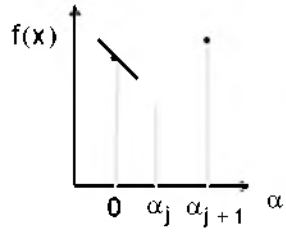
dient $\nabla f(x_k)$ obtained at the last update. The right-hand points represent the points accumulated in the minor iterations of the line search procedure.

The terms α_q and α_c refer to the minimum obtained from a respective quadratic and cubic interpolation or extrapolation. For highly nonlinear functions, α_c and α_q may be negative, in which case they are set to a value of $2\alpha_k$ so that they are always maintained to be positive. Cases 1 and 2 use quadratic interpolation with two points and one gradient to estimate a third point that brackets the minimum. If this fails, cases 3 and 4 represent the possibilities for changing the step length when at least three points are available.

When the minimum is finally bracketed, cubic interpolation is achieved using one gradient and three function evaluations. If the interpolated point is greater than any of the three used for the interpolation, then it is replaced with the point with the smallest function value. Following the line search procedure the Hessian update procedure is performed as for the cubic polynomial line search method.

Figure 2-4: Line Search Procedures with Only Gradient for the First Point.

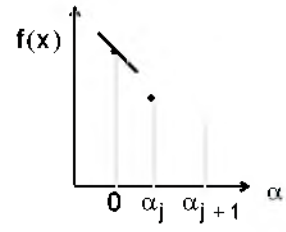
Case 1: $f(x_j) \geq f(x_k)$



Reduce step length.

$\alpha_{j+1} = \alpha_q$

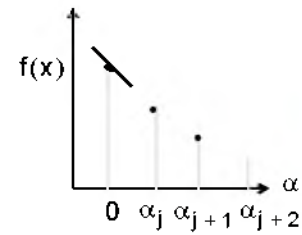
Case 2: $f(x_j) < f(x_k)$



Increase step length.

$\alpha_{j+1} = 1.2\alpha_q$

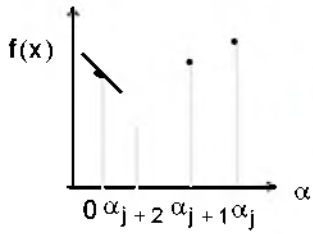
Case 3: $f(x_{j+1}) < f(x_k)$



Increase step length.

$\alpha_{j+2} = \max\{1.2\alpha_q, 2\alpha_{j+1}\}$

Case 4: $f(x_{j+1}) > f(x_k)$



Reduce step length.

$\alpha_{j+2} = \alpha_c$

Least Squares Optimization

The line search procedures used in conjunction with a quasi-Newton method are used in the function fminu. They are also used as part of a nonlinear least squares (LS) optimization routine, lsqnonlin. In the least squares problem a function, f(x) is minimized that is a sum of squares.

LS

$$\min_{x \in \Re^n} f(x) = \frac{1}{2}\| F(x) \|_2^2 = \frac{1}{2}\sum_i F_i(x)^2 \tag{2-17}$$

Problems of this type occur in a large number of practical applications especially when fitting model functions to data, i.e., nonlinear parameter estimation. They are also prevalent in control where you want the output, $y(x,t)$ to follow some continuous model trajectory, $\phi(t)$, for vector x and scalar t. This problem can be expressed as

$$\min_{x \in \Re^n} \int_{t_2}^{t_1} (y(x,t) - \phi(t))^2 dt \tag{2-18}$$

where $y(x,t)$ and $\phi(t)$ are scalar functions.

When the integral is discretized using a suitable quadrature formula, Eq. 2-18 can be formulated as a least squares problem

$$\min_{x \in \Re^n} f(x) = \sum_{i=1}^{m} (\bar{y}(x,t_i) - \bar{\phi}(t_i))^2 \tag{2-19}$$

where $\bar{y}$ and $\bar{\phi}$ include the weights of the quadrature scheme. Note that in this problem the vector F(x) is

$$F(x) = \begin{bmatrix} \bar{y}(x,t_1) - \bar{\phi}(t_1) \\ \bar{y}(x,t_2) - \bar{\phi}(t_2) \\ \cdots \\ \bar{y}(x,t_m) - \bar{\phi}(t_m) \end{bmatrix}$$

In problems of this kind the residual $\| F(x) \|$ is likely to be small at the optimum since it is general practice to set realistically achievable target

trajectories. Although the function in LS (Eq. 2-18) can be minimized using a general unconstrained minimization technique as described in the "Unconstrained Optimization" section, certain characteristics of the problem can often be exploited to improve the iterative efficiency of the solution procedure. The gradient and Hessian matrix of LS (Eq. 2-18) have a special structure.

Denoting the $m \times n$ Jacobian matrix of F(x) as J(x), the gradient vector of f(x) as $G(x)$, the Hessian matrix of f(x) as $H(x)$, and the Hessian matrix of each $F_i(x)$ as $H_i(x)$, we have

$$G(x) = 2J(x)^T F(x)$$
$$H(x) = 2J(x)^T J(x) + 2Q(x)$$

where

$$Q(x) = \sum_{i=1}^{m} F_i(x) \cdot H_i(x) \qquad (2\text{-}20)$$

The matrix Q(x) has the property that when the residual $\| F(x) \|$ tends to zero as x_k approaches the solution, then Q(x) also tends to zero. Thus when $\| F(x) \|$ is small at the solution, a very effective method is to use the Gauss-Newton direction as a basis for an optimization procedure.

Gauss-Newton Method

In the Gauss-Newton method, a search direction, d_k, is obtained at each major iteration, k, that is a solution of the linear least-squares problem

Gauss-Newton

$$\min_{x \in \Re^n} \| J(x_k)d_k - F(x_k) \|_2^2 \qquad (2\text{-}21)$$

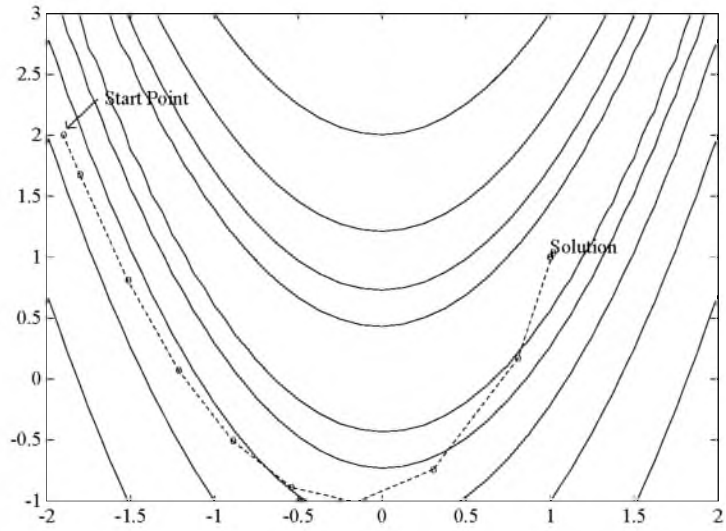
The direction derived from this method is equivalent to the Newton direction when the terms of Q(x) can be ignored. The search direction d_k can be used as part of a line search strategy to ensure that at each iteration the function f(x) decreases.

To consider the efficiencies that are possible with the Gauss-Newton method, Fig. 2-5 shows the path to the minimum on Rosenbrock's function (Eq. 2-2) when posed as a least squares problem. The Gauss-Newton method converges

after only 48 function evaluations using finite difference gradients compared to 140 iterations using an unconstrained BFGS method.

The Gauss-Newton method often encounters problems when the second order term Q(x) in Eq. 2-20 is significant. A method that overcomes this problem is the Levenberg-Marquardt method.

Figure 2-5: Gauss-Newton Method on Rosenbrock's Function



Levenberg-Marquardt Method

The Levenberg-Marquardt [18,19] method uses a search direction that is a solution of the linear set of equations

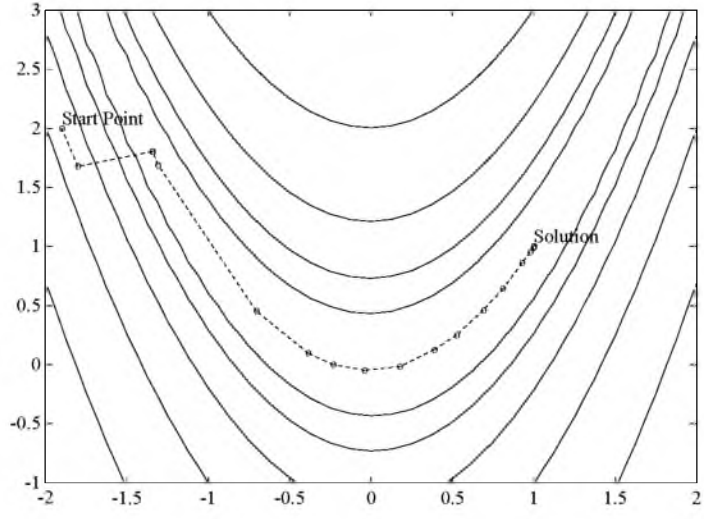
$$(J(x_k)^T J(x) + \lambda_k I)d_k = -J(x_k)F(x_k) \tag{2-22}$$

where the scalar λ_k controls both the magnitude and direction of d_k. When λ_k is zero, the direction d_k is identical to that of the Gauss-Newton method. As λ_k tends to infinity, d_k tends towards a vector of zeros and a steepest descent direction. This implies that for some sufficiently large λ_k, the term

$F(x_k + d_k) < F(x_k)$ holds true. The term λ_k can therefore be controlled to ensure descent even when second order terms, which restrict the efficiency of the Gauss-Newton method, are encountered.

The Levenberg-Marquardt method therefore uses a search direction that is a cross between the Gauss-Newton direction and the steepest descent. This is illustrated in Fig. 2-6 below. The solution for Rosenbrock's function (Eq. 2-2) converges after 90 function evaluations compared to 48 for the Gauss-Newton method. The poorer efficiency is partly because the Gauss-Newton method is generally more effective when the residual is zero at the solution. However, such information is not always available beforehand, and occasional poorer efficiency of the Levenberg-Marquardt method is compensated for by its increased robustness.

Figure 2-6: Levenberg-Marquardt Method on Rosenbrock's Function



Nonlinear Least Squares Implementation

For a general survey of nonlinear least squares methods see Dennis [21]. Specific details on the Levenberg-Marquardt method can be found in Moré [20]. Both the Gauss-Newton method and the Levenberg-Marquardt method are implemented in the Optimization Toolbox. Details of the implementations are discussed below.

Gauss-Newton Implementation

The Gauss-Newton method is implemented using similar polynomial line search strategies discussed for unconstrained optimization. In solving the linear least squares problem (Prob. 2.18), exacerbation of the conditioning of the equations is avoided by using the QR decomposition of $J(x_k)$ and applying the decomposition to $F(x_k)$ (using the MATLAB \ operator). This is in contrast to inverting the explicit matrix, $J(x_k)^T J(x_k)$, which can cause unnecessary errors to occur.

Robustness measures are included in the method. These measures consist of changing the algorithm to the Levenberg-Marquardt method when either the step length goes below a threshold value (in this implementation 1e–15) or when the condition number of $J(x_k)$ is below 1e–10. The condition number is a ratio of the largest singular value to the smallest.

Levenberg-Marquardt Implementation

The main difficulty in the implementation of the Levenberg-Marquardt method is an effective strategy for controlling the size of λ_k at each iteration so that it is efficient for a broad spectrum of problems. The method used in this implementation is to estimate the relative nonlinearity of f(x) using a linear predicted sum of squares $f_p(x_k)$ and a cubicly interpolated estimate of the minimum $f_k(x_*)$. In this way the size of λ_k is determined at each iteration.

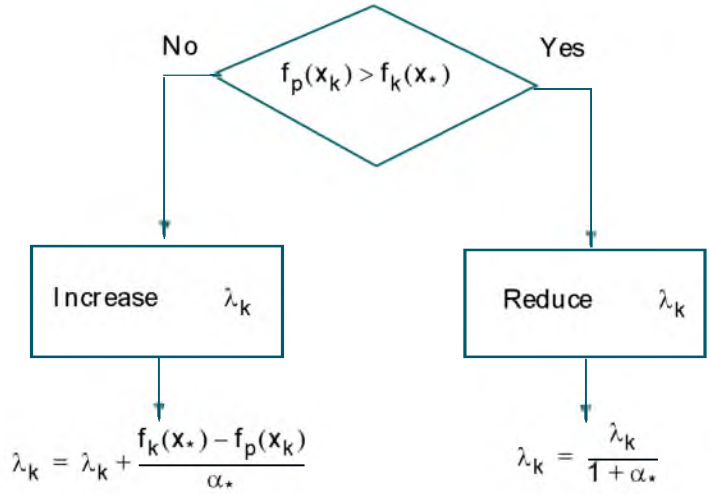
The linear predicted sum of squares is calculated as

$$f_p(x_k) = (J(x_{k-1}))^T d_{k-1} + F(x) \qquad (2\text{-}23)$$

and the term $f_k(x_*)$ is obtained by cubicly interpolating the points $f(x_k)$ and $f(x_{k-1})$. A step length parameter α^* is also obtained from this interpolation, which is the estimated step to the minimum. If $f_p(x_k)$ is greater than $f_k(x_*)$, then λ_k is reduced, otherwise it is increased. The justification for this is that

the difference between $f_p(x_k)$ and $f_k(x_*)$ is a measure of the effectiveness of the Gauss-Newton method and the linearity of the problem. This determines whether to use a direction approaching the steepest descent direction or the Gauss-Newton direction. The formulas for the reduction and increase in λ_k, which have been developed through consideration of a large number of test problems, are shown in Fig. 2-7 below.

Figure 2-7: Updating λ_k



Following the update of λ_k, a solution of Eq. 2-22 is used to obtain a search direction, d_k. A step length of unity is then taken in the direction d_k, which is followed by a line search procedure similar to that discussed for the unconstrained implementation. The line search procedure ensures that $f(x_{k+1}) < f(x_k)$ at each major iteration and the method is therefore a descent method.

The implementation has been successfully tested on a large number of nonlinear problems. It has proved to be more robust than the Gauss-Newton method and iteratively more efficient than an unconstrained method. The Levenberg-Marquardt algorithm is the default method used by `lsqnonlin`. The Gauss-Newton method can be selected by setting `options(5) = 1`.

Constrained Optimization

In constrained optimization, the general aim is to transform the problem into an easier subproblem that can then be solved and used as the basis of an iterative process. A characteristic of a large class of early methods is the translation of the constrained problem to a basic unconstrained problem by using a penalty function for constraints, which are near or beyond the constraint boundary. In this way the constrained problem is solved using a sequence of parameterized unconstrained optimizations, which in the limit (of the sequence) converge to the constrained problem. These methods are now considered relatively inefficient and have been replaced by methods that have focused on the solution of the Kuhn-Tucker (KT) equations. The KT equations are necessary conditions for optimality for a constrained optimization problem. If the problem is a so-called convex programming problem, that is, f(x) and $G_i(x), i = 1, \ldots, m$, are convex functions, then the KT equations are both necessary and sufficient for a global solution point.

Referring to GP (Eq. 2-1), the Kuhn-Tucker equations can be stated as

$$\begin{aligned} & f(x^*) + \sum_{i=1}^{m} \lambda_i^* \cdot \nabla G_i(x^*) = 0 \\ & \nabla G_i(x^*) = 0 \qquad i = 1, \ldots, m_e \\ & \lambda_i^* \geq 0 \qquad i = m_e + 1, \ldots, m \end{aligned} \tag{2-24}$$

The first equation describes a canceling of the gradients between the objective function and the active constraints at the solution point. For the gradients to be canceled, Lagrange Multipliers (λ_i, $i = 1, \ldots m$) are necessary to balance the deviations in magnitude of the objective function and constraint gradients. Since only active constraints are included in this canceling operation, constraints that are not active must not be included in this operation and so are given Lagrange multipliers equal to zero. This is stated implicitly in the last two equations of Eq. 2-24.

The solution of the KT equations forms the basis to many nonlinear programming algorithms. These algorithms attempt to compute directly the Lagrange multipliers. Constrained quasi-Newton methods guarantee superlinear convergence by accumulating second order information regarding the KT equations using a quasi-Newton updating procedure. These methods are commonly referred to as Sequential Quadratic Programming (SQP) methods since a QP

sub-problem is solved at each major iteration (also known as Iterative Quadratic Programming, Recursive Quadratic Programming, and Constrained Variable Metric methods).

Sequential Quadratic Programming (SQP)

SQP methods represent state-of-the-art in nonlinear programming methods. Schittowski [22], for example, has implemented and tested a version that out performs every other tested method in terms of efficiency, accuracy, and percentage of successful solutions, over a large number of test problems.

Based on the work of Biggs [9], Han [10], and Powell [11,12], the method allows you to closely mimic Newton's method for constrained optimization just as is done for unconstrained optimization. At each major iteration an approximation is made of the Hessian of the Lagrangian function using a quasi-Newton updating method. This is then used to generate a QP sub-problem whose solution is used to form a search direction for a line search procedure. An overview of SQP is found in Fletcher [2], Gill et al. [1], Powell [13], and Schittowski [14]. The general method, however, is stated here.

Given the problem description in GP (Eq. 2.1) the principal idea is the formulation of a QP sub-problem based on a quadratic approximation of the Lagrangian function.

$$L(x,\lambda) = f(x) + \sum_{i=1}^{m} \lambda_i \cdot g_i(x) \tag{2-25}$$

Here Eq. 2.1 is simplified by assuming that bound constraints have been expressed as inequality constraints. The QP sub-problem is obtained by linearizing the nonlinear constraints.

QP Subproblem

$$\begin{aligned} &\underset{d \in \Re^n}{\text{minimize}} \quad \frac{1}{2} d^T H_k d + \nabla f(x_k)^T d \\ &\nabla g_i(x_k)^T d + g_i(x_k) = 0 \qquad i = 1, \ldots m_e \\ &\nabla g_i(x_k)^T d + g_i(x_k) \le 0 \qquad i = m_e + 1, \ldots m \end{aligned} \tag{2-26}$$

This sub-problem can be solved using any QP algorithm (see, for instance, the "Quadratic Programming Solution" section). The solution is used to form a new iterate

$$x_{k+1} = x_k + \alpha_k d_k$$

The step length parameter α_k is determined by an appropriate line search procedure so that a sufficient decrease in a merit function is obtained (see the "Updating the Hessian Matrix" section). The matrix H_k is a positive definite approximation of the Hessian matrix of the Lagrangian function (Eq. 2-25). H_k can be updated by any of the quasi-Newton methods, although the BFGS method (see the section "Updating the Hessian Matrix") appears to be the most popular.

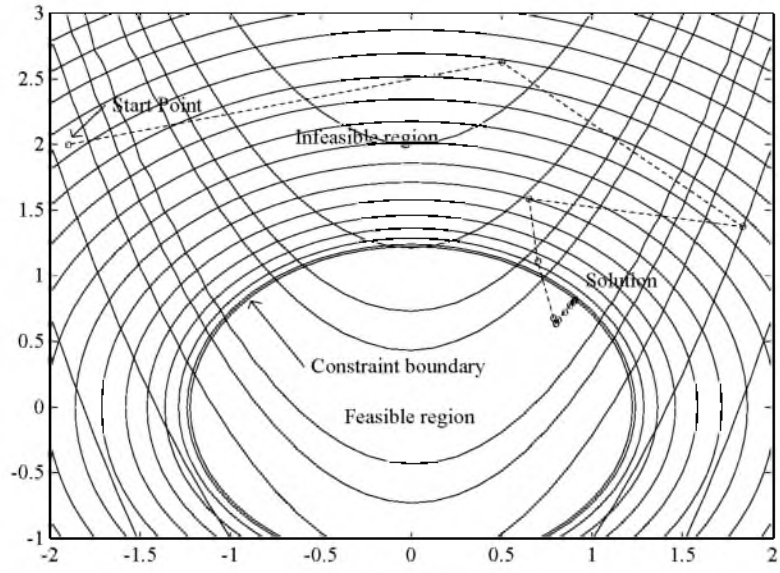
A nonlinearly constrained problem can often be solved in fewer iterations than an unconstrained problem using SQP. One of the reasons for this is that, because of limits on the feasible area, the optimizer can make well-informed decisions regarding directions of search and step length.

Consider Rosenbrock's function (Eq. 2-2) with an additional nonlinear inequality constraint, g(x)

$$x_1^2 + x_2^2 - 1.5 \leq 0 \qquad (2\text{-}27)$$

This was solved by an SQP implementation in 96 iterations compared to 140 for the unconstrained case. Fig. 2-8 shows the path to the solution point $x = [0.9072, 0.8228]$ starting at $x = [-1.9, 2]$.

Figure 2-8: SQP Method on Nonlinear Linearly Constrained Rosenbrock's Function



SQP Implementation

The MATLAB SQP implementation consists of three main stages, which are discussed briefly in the following sub-sections:

- Updating of the Hessian matrix of the Lagrangian function
- Quadratic programming problem solution
- Line search and merit function calculation

Updating the Hessian Matrix

At each major iteration a positive definite quasi-Newton approximation of the Hessian of the Lagrangian function, H,is calculated using the BFGS method where λ_i (i = 1,...,m) is an estimate of the Lagrange multipliers.

Hessian Update (BFGS)

$$H_{k+1} = H_k + \frac{q_k q_k'}{q_k^T s_k} - \frac{H_k' H_k}{s_k^T H_k s_k} \quad \text{where}$$

$$s_k = x_{k+1} - x_k \tag{2-28}$$

$$q_k = \nabla f(x_{k+1}) + \sum_{i=1}^{n} \lambda_i \cdot \nabla g_i(x_{k+1}) - \left(\nabla f(x_k) + \sum_{i=1}^{n} \lambda_i \cdot \nabla g_i(x_k) \right)$$

Powell [11] recommends keeping the Hessian positive definite even though it may be positive indefinite at the solution point. A positive definite Hessian is maintained providing $q_k^T s_k$ is positive at each update and that H is initialized with a positive definite matrix. When $q_k^T s_k$ is not positive, q_k is modified on an element by element basis so that $q_k^T s_k > 0$. The general aim of this modification is to distort the elements of q_k, which contribute to a positive definite update, as little as possible. Therefore, in the initial phase of the modification, the most negative element of $q_k .* s_k$ is repeatedly halved. This procedure is continued until $q_k^T s_k$ is greater than or equal to 1e-5. If after this procedure, $q_k^T s_k$ is still not positive, q_k is modified by adding a vector v multiplied by a constant scalar w, that is,

$$q_k = q_k + wv \tag{2-29}$$

where

$$v_i = \begin{cases} \nabla g_i(x_{k+1}) \cdot g_i(x_{k+1}) - \nabla g_i(x_k) \cdot g_i(x_k), & \text{if } (q_k)_i \cdot w < 0 \text{ and } (q_k)_i \cdot (s_k)_i < 0 \\ & (i = 1, \ldots m) \\ 0, & \text{otherwise} \end{cases}$$

and w is systematically increased until $q_k^T s_k$ becomes positive.

The functions `constr`, `minimax`, `attgoal`, and `seminf` all use SQP. If `options(1)` is set to 1, then various information is given such as function values and the maximum constraint violation. When the Hessian has to be modified using the first phase of the procedure described above to keep it positive definite, then `Hessian modified` is displayed. If the Hessian has to be modified again using the second phase of the approach described above, then `Hessian modified twice` is displayed. When the QP sub-problem is infeasible, then infeasible will be displayed. Such displays are usually not a cause for concern but indicate that the problem is highly nonlinear and that convergence may take longer than usual. Sometimes the message `no update` is displayed indicating that $q_k^T s_k$ is nearly zero. This can be an indication that the problem setup is wrong or you are trying to minimize a noncontinuous function.

Quadratic Programming Solution

At each major iteration of the SQP method a QP problem is solved of the form where A_i refers to the i th row of the m-by-n matrix A.

QP

$$\begin{aligned} \underset{d \in \Re^n}{\text{minimize}} \quad & q(d) = \frac{1}{2} d^T H d + c^T d \\ & A_i d = b_i \qquad i = 1, \ldots, m_e \\ & A_i d \le b_i \qquad i = m_e + 1, \ldots, m \end{aligned} \tag{2-30}$$

The method used in the Optimization Toolbox is an active set strategy (also known as a projection method) similar to that of Gill et al., described in [16] and [17]. It has been modified for both LP and QP problems.

The solution procedure involves two phases: the first phase involves the calculation of a feasible point (if one exists), the second phase involves the generation of an iterative sequence of feasible points that converge to the solution. In

this method an active set is maintained, A_k, which is an estimate of the active constraints (i.e., which are on the constraint boundaries) at the solution point. Virtually all QP algorithms are active set methods. This point is emphasized because there exist many different methods that are very similar in structure but that are described in widely different terms.

$\bar{A}_k$ is updated at each iteration, k, and this is used to form a basis for a search direction $\hat{d}_k$. Equality constraints always remain in the active set, A_k. The notation for the variable, $\hat{d}_k$, is used here to distinguish it from d_k in the major iterations of the SQP method. The search direction, $\hat{d}_k$, is calculated and minimizes the objective function while remaining on any active constraint boundaries. The feasible subspace for $\hat{d}_k$ is formed from a basis, Z_k whose columns are orthogonal to the estimate of the active set A_k (i.e., $A_k Z_k = 0$). Thus a search direction, which is formed from a linear summation of any combination of the columns of Z_k, is guaranteed to remain on the boundaries of the active constraints.

The matrix Z_k is formed from the last m-l columns of the QR decomposition of the matrix $\bar{A}_k$, where l is the number of active constraints and l < m. That is, Z_k is given by

$$Z_k = Q[:, l+1:m]$$

$$\text{where} \qquad Q^T \bar{A}_k^T = \begin{bmatrix} R \\ 0 \end{bmatrix} \tag{2-31}$$

Having found Z_k, a new search direction $\hat{d}_k$ is sought that minimizes q(d) where $\hat{d}_k$ is in the null space of the active constraints, that is, $\hat{d}_k$ is a linear combination of the columns of Z_k: $\hat{d}_k = Z_k p$ for some vector p.

Then if we view our quadratic as a function of p, by substituting for $\hat{d}_k$, we have

$$q(p) = \frac{1}{2} p^T Z_k^T H Z_k p + c^T Z_k p \tag{2-32}$$

Differentiating this with respect to p yields

$$\nabla q(p) = Z_k^T H Z_k p + Z_k^T c \tag{2-33}$$

$\nabla q(p)$ is referred to as the projected gradient of the quadratic function because it is the gradient projected in the subspace defined by Z_k. The term $Z_k^T H Z_k$ is

called the projected Hessian. Assuming the Hessian matrix H is positive definite (which is the case in this implementation of SQP), then the minimum of the function q(p) in the subspace defined by Z_k occurs when $\nabla q(p) = 0$, which is the solution of the system of linear equations

$$Z_k^T H Z_k p = -Z_k^T c \tag{2-34}$$

A step is then taken of the form

$$x_{k+1} = x_k + \alpha \hat{d}_k \qquad \text{where } \hat{d}_k = Z_k^T p \tag{2-35}$$

At each iteration, because of the quadratic nature of the objective function, there are only two choices of step length α. A step of unity along $\hat{d}_k$ is the exact step to the minimum of the function restricted to the null space of A_k. If such a step can be taken, without violation of the constraints, then this is the solution to QP (Eq. 2.31). Otherwise, the step along $\hat{d}_k$ to the nearest constraint is less than unity and a new constraint is included in the active set at the next iterate. The distance to the constraint boundaries in any direction $\hat{d}_k$ is given by

$$\alpha = \min_i \left\{ \frac{-(A_i x_k - b_i)}{A_i \hat{d}_k} \right\} \qquad (i = 1, \ldots, m) \tag{2-36}$$

which is defined for constraints not in the active set, and where the direction $\hat{d}_k$ is towards the constraint boundary, i.e., $A_i \hat{d}_k > 0$, $i = 1, \ldots, m$.

When n independent constraints are included in the active set, without location of the minimum, Lagrange multipliers, λ_k are calculated that satisfy the nonsingular set of linear equations

$$\bar{A}_k^T \lambda_k = c \tag{2-37}$$

If all elements of λ_k are positive, x_k is the optimal solution of QP (Eq. 2.31). However, if any component of λ_k is negative, and it does not correspond to an equality constraint, then the corresponding element is deleted from the active set and a new iterate is sought.

Initialization

The algorithm requires a feasible point to start. If the current point from the SQP method is not feasible, then a point can be found by solving the linear programming problem

$$\begin{aligned} \underset{\gamma \in \Re,\, x \in \Re^n}{\text{minimize}} \quad & \gamma \\ A_i x &= b_i \qquad i = 1, \ldots, m_e \\ A_i x - \gamma &\le b_i \qquad i = m_e + 1, \ldots, m \end{aligned} \tag{2-38}$$

The notation A_i indicates the i th row of the matrix A. A feasible point (if one exists) to Eq. 2.38 can be found by setting x to a value that satisfies the equality constraints. This can be achieved by solving an under- or over-determined set of linear equations formed from the set of equality constraints. If there is a solution to this problem, then the slack variable γ is set to the maximum inequality constraint at this point.

The above QP algorithm is modified for LP problems by setting the search direction to the steepest descent direction at each iteration where g_k is the gradient of the objective function (equal to the coefficients of the linear objective function)

$$\hat{d}_k = -Z_k Z_k^T g_k \tag{2-39}$$

If a feasible point is found using the above LP method, the main QP phase is entered. The search direction $\hat{d}_k$ is initialized with a search direction $\hat{d}_1$ found from solving the set of linear equations

$$H\hat{d}_1 = -g_k \tag{2-40}$$

where g_k is the gradient of the objective function at the current iterate x_k (i.e., $Hx_k + c$).

If a feasible solution is not found for the QP problem, the direction of search for the main SQP routine $\hat{d}_k$ is taken as one that minimizes γ.

Line Search and Merit Function

The solution to the QP sub-problem produces a vector d_k, which is used to form a new iterate

$$x_{k+1} = x_k + \alpha d_k \tag{2-41}$$

The step length parameter α_k is determined in order to produce a sufficient decrease in a merit function. The merit function used by Han [15] and Powell [15] of the form below has been used in this implementation

Merit Function

$$\Psi(x) = f(x) + \sum_{i=1}^{m_e} r_i \cdot g_i(x) + \sum_{i=m_e+1}^{m} r_i \cdot \max\{0, g_i(x)\} \tag{2-42}$$

Powell recommends setting the penalty parameter

$$r_i = (r_{k+1})_i = \max_i\left\{\lambda_i, \frac{1}{2}((r_k)_i + \lambda_i)\right\}, \qquad i = 1, \ldots, m \tag{2-43}$$

This allows positive contribution form constraints that are inactive in the QP solution but were recently active. In this implementation, initially the penalty parameter r_i is set to

$$r_i = \frac{\| \nabla f(x) \|}{\| \nabla g_i(x) \|} \tag{2-44}$$

where $\| \cdot \|$ represents the Euclidean norm.

This ensures larger contributions to the penalty parameter from constraints with smaller gradients, which would be the case for active constraints at the solution point.

Multiobjective Optimization

The rigidity of the mathematical problem posed by the general optimization formulation given in GP (Eq. 2-1) is often remote from that of a practical design problem. Rarely does a single objective with several hard constraints adequately represent the problem being faced. More often there is a vector of objectives $F(x) = \{F_1(x), F_2(x), ..., F_m(x)\}$ that must be traded off in some way. The relative importance of these objectives is not generally known until the system's best capabilities are determined and trade-offs between the objectives fully understood. As the number of objectives increases, trade-offs are likely to become complex and less easily quantified. There is much reliance on the intuition of the designer and his or her ability to express preferences throughout the optimization cycle. Thus, requirements for a multiobjective design strategy are to enable a natural problem formulation to be expressed, yet be able to solve the problem and enter preferences into a numerically tractable and realistic design problem.

This section begins with an introduction to multiobjective optimization, looking at a number of alternative methods. Attention is focused on the Goal Attainment method, which can be posed as a nonlinear programing problem. Algorithm improvements to the SQP method are presented for use with the Goal Attainment method.

Introduction to Multiobjective Optimization

Multiobjective optimization is concerned with the minimization of a vector of objectives $F(x)$ that may be the subject of a number of constraints or bounds.

MO

$$\begin{aligned} &\underset{x \in \Re^n}{\text{minimize}} \quad F(x) \\ &\qquad G_i(x) = 0 \qquad i = 1, ..., m_e \\ &\qquad G_i(x) \le 0 \qquad i = m_e + 1, ..., m \\ &\qquad x_l \le x \le x_u \end{aligned} \tag{2-45}$$

Note that, because $F(x)$ is a vector, if any of the components of $F(x)$ are competing, there is no unique solution to this problem. Instead, the concept of noninferiority [25] (also called Pareto optimality [24], [26]) must be used to characterize the objectives. A noninferior solution is one in which an improve-

ment in one objective requires a degradation of another. To define this concept more precisely, consider a feasible region, Ω, in the parameter space $x \in \Re^n$ that satisfies all the constraints, i.e.,

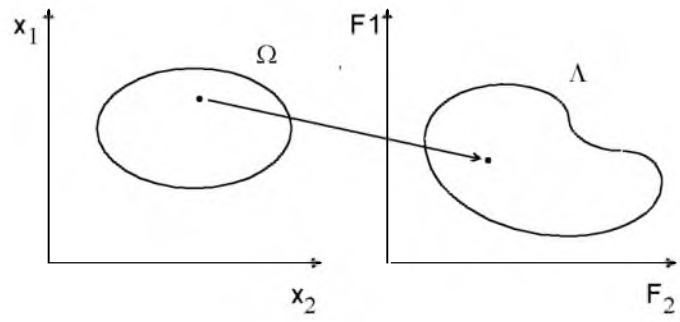
$$\begin{aligned} \Omega &= \{x \in \Re^n\} \\ \text{subject to} \quad g_i(x) &= 0 \qquad i = 1, \ldots, m_e \\ g_i(x) &\le 0 \qquad i = m_e + 1, \ldots, m \\ x_l &\le x \le x_u \end{aligned} \tag{2-46}$$

This allows us to define the corresponding feasible region for the objective function space Λ

$$\Lambda = \{y \in \Re^m\} \quad \text{where} \quad y = F(x) \quad \text{subject to} \quad x \in \Omega. \tag{2-47}$$

The performance vector, F(x), maps parameter space into objective function space as is represented for a two-dimensional case in Fig. 2-9 below.

Figure 2-9: Mapping from Parameter Space into Objective Function Space.



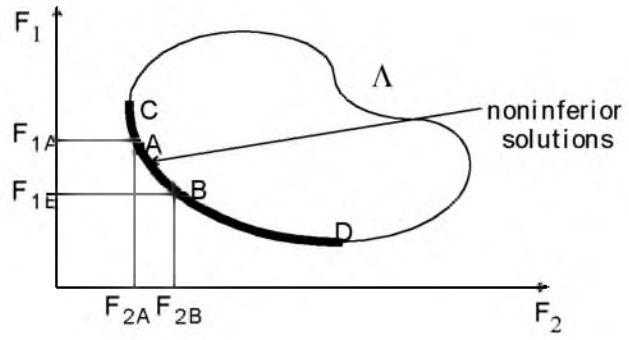
A noninferior solution point can now be defined.

Definition: A point $x^* \in \Omega$ is a noninferior solution if for some neighborhood of x^* there does not exist a Δx such that $(x^* + \Delta x) \in \Omega$ and

$$\begin{aligned} F_i(x^* + \Delta x) &\le F_i(x^*) \qquad i = 1, \ldots, m \\ F_j(x^* + \Delta x) &< F_j(x^*) \qquad \text{for some } j. \end{aligned} \tag{2-48}$$

In the two-dimensional representation of Fig. 2-10 the set of noninferior solutions lies on the curve between C and D. Points A and B represent specific noninferior points.

Figure 2-10: Set of Noninferior Solutions.



A and B are clearly noninferior solution points because an improvement in one objective, F_1, requires a degradation in the other objective, F_2, i.e., $F_{1B} < F_{1A}$, $F_{2B} > F_{2A}$.

Since any point in Ω that is not a noninferior point represents a point in which improvement can be attained in all the objectives, it is clear that such a point is of no value. Multiobjective optimization is, therefore, concerned with the generation and selection of noninferior solution points. The techniques for multiobjective optimization are wide and varied and all the methods cannot be covered within the scope of this toolbox. However, some of the techniques are described below.

Weighted Sum Strategy

The weighted sum strategy converts the multiobjective problem of minimizing the vector $F(x)$ into a scalar problem by constructing a weighted sum of all the objectives.

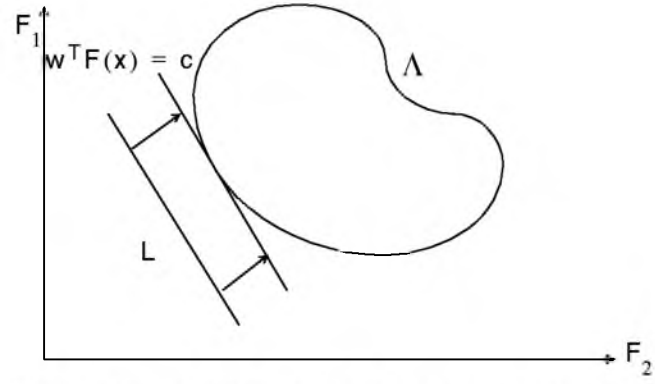
Weighted Sum

$$\underset{x \in \Omega}{\text{minimize}} \quad f(x) = \sum_{i=1}^{m} w_i \cdot F_i(x)^2 \tag{2-49}$$

The problem can then be optimized using a standard unconstrained optimization algorithm. The problem here is in attaching weighting coefficients to each of the objectives. The weighting coefficients do not necessarily correspond directly to the relative importance of the objectives or allow trade-offs between the objectives to be expressed. Further, the noninferior solution boundary may be nonconcurrent so that certain solutions are not accessible.

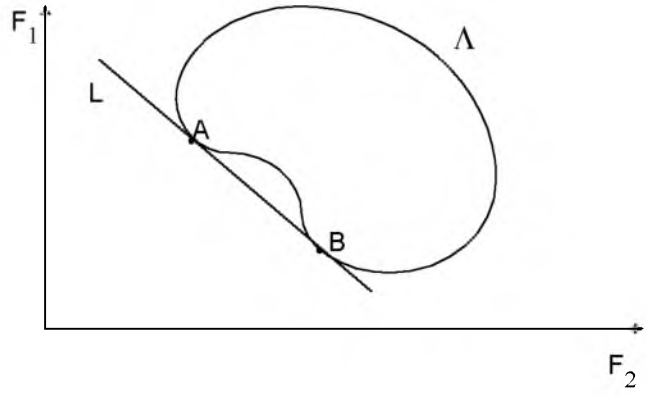
This can be illustrated geometrically. Consider the two objective case in Fig. 2-11. In the objective function space a line, L, $w^T F(x) = c$ is drawn. The minimization of Eq. 2-49 can be interpreted as finding the value of c for which L just touches the boundary of Λ as it proceeds outwards from the origin. Selection of weights w_1 and w_2, therefore, defines the slope of L, which in turn leads to the solution point where L touches the boundary of Λ.

Figure 2-11: Geometrical Representation of the Weighted Sum Method.



The aforementioned convexity problem arises when the lower boundary of Λ is nonconvex as shown in Fig. 2-12. In this case the set of noninferior solutions between A and B is not available.

Figure 2-12: Nonconvex Solution Boundary.



ε-Constraint Method

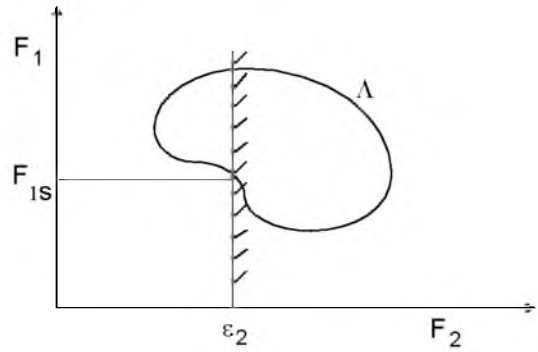
A procedure that overcomes some of the convexity problems of the weighted sum technique is the ε-constraint method. This involves minimizing a primary objective, F_p, and expressing the other objectives in the form of inequality constraints

$$\begin{aligned} &\underset{x \in \Omega}{\text{minimize}} \quad F_p(x) \\ &\text{subject to} \qquad F_i(x) \le \varepsilon_i \qquad i = 1, \ldots, m \qquad i \neq p \end{aligned} \tag{2-50}$$

Fig. 2-13 shows a two-dimensional representation of the ε-constraint method for a two objective problem.

Figure 2-13: Geometrical Representation of ε-Constraint Method

$$\underset{x \in \Omega}{\text{minimize}}\ F_1(x) \qquad \text{subject to: } F_2 x \leq \varepsilon_2$$



This approach is able to identify a number of noninferior solutions on a nonconvex boundary that are not obtainable using the weighted sum technique, for example, at the solution point $F_1 = F_{1s}$ and $F_2 = \varepsilon_2$. A problem with this method is, however, a suitable selection of ε to ensure a feasible solution. A further disadvantage of this approach is that the use of hard constraints is rarely adequate for expressing true design objectives. Similar methods exist, such as that of Waltz [31], which prioritize the objectives. The optimization proceeds with reference to these priorities and allowable bounds of acceptance. The difficulty here is in expressing such information at early stages of the optimization cycle.

In order for the designers' true preferences to be put into a mathematical description, the designers must express a full table of their preferences and satisfaction levels for a range of objective value combinations. A procedure must then be realized that is able to find a solution with reference to this. Such methods have been derived for discrete functions using the branches of statistics known as decision theory and game theory (for a basic introduction, see [28]). Implementation for continuous functions requires suitable discretization strategies and complex solution methods. Since it is rare for the designer to know such detailed information, this method is deemed impractical for most practical design problems. It is, however, seen as a possible area for further research.

What is required is a formulation that is simple to express, retains the designers preferences, and is numerically tractable.

Goal Attainment Method

The method described here is the Goal Attainment method of Gembicki [27]. This involves expressing a set of design goals, $F^* = \{F_1^*, F_2^*, \ldots, F_m^*\}$, which is associated with a set of objectives, $F(x) = \{F_1(x), F_2(x), \ldots, F_m(x)\}$. The problem formulation allows the objectives to be under- or over-achieved enabling the designer to be relatively imprecise about initial design goals. The relative degree of under- or over-achievement of the goals is controlled by a vector of weighting coefficients, $w = \{w_1, w_2, \ldots, w_m\}$, and is expressed as a standard optimization problem using the following formulation:

Goal Attainment

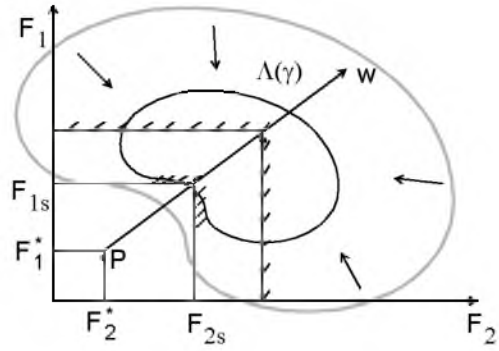
$$\begin{aligned} &\underset{\gamma \in \Re,\, x \in \Omega}{\text{minimize}} \quad \gamma \\ &\text{such that} \quad F_i(x) - w_i\gamma \le F_i^* \qquad i = 1, \ldots, m \end{aligned} \tag{2-51}$$

The term $w_i\gamma$ introduces an element of slackness into the problem, which otherwise imposes that the goals be rigidly met. The weighting vector, w, enables the designer to express a measure of the relative trade-offs between the objectives. For instance, setting the weighting vector, w, equal to the initial goals indicates that the same percentage under- or over-attainment of the goals, F^*, is achieved. Hard constraints can be incorporated into the design by setting a particular weighting factor to zero (i.e., $w_i = 0$). The Goal Attainment method provides a convenient intuitive interpretation of the design problem, which is solvable using standard optimization procedures. Illustrative examples of the use of Goal Attainment method in control system design can be found in Fleming [29,30].

The Goal Attainment method is represented geometrically in Fig. 2-14 for the two-dimensional problem.

Figure 2-14: Geometrical Representation of Goal Attainment Method.

$$\underset{\gamma,\, x \in \Omega}{\text{minimize}}\ \gamma \quad \text{subject to:} \quad F_1(x) - w_1\gamma \le F_1^*$$
$$F_2(x) - w_2\gamma \le F_2^*$$



Specification of the goals, $\{F_1^*, F_2^*\}$, defines the goal point, P. The weighting vector defines the direction of search from P to the feasible function space, $\Lambda(\gamma)$. During the optimization γ is varied, which changes the size of the feasible region. The constraint boundaries converge to the unique solution point F_{1s}, F_{2s}.

Algorithm Improvements for Goal Attainment Method

The Goal Attainment method has the advantage that it can be posed as a nonlinear programming problem. Characteristics of the problem can also be exploited in a nonlinear programming algorithm. In Sequential Quadratic Programming (SQP) the choice of merit function for the line search is not easy because, in many cases, it is difficult to "define" the relative importance between improving the objective function and reducing constraint violations. This has resulted in a number of different schemes for constructing the merit function (see, for example, Schittowski [22]). In Goal Attainment programming

there may be a more appropriate merit function, which can be achieved by posing Eq. 2-51 as the minimax problem

$$\underset{x \in \Re^n}{\text{minimize}} \ \max_i \{\Lambda_i\}$$

$$\text{where} \qquad \Lambda_i = \frac{F_i(x) - F_i^*}{w_i} \qquad i = 1, \ldots, m \tag{2-52}$$

Following the argument of Brayton et al. [32] for minimax optimization using SQP, using the merit function of Eq. 2-43 for the Goal Attainment problem of Eq. 2-52, gives

$$\psi(x, \gamma) = \gamma + \sum_{i=1}^{m} r_i \cdot \max \{0, F_i(x) - w_i\gamma - F_i^*\} \tag{2-53}$$

When the merit function of Eq. 2-53 is used as the basis of a line search procedure, then, although $\psi(x, \gamma)$ may decrease for a step in a given search direction, the function $\max \Lambda_i$ may paradoxically increase. This is accepting a degradation in the worst case objective. Since the worst case objective is responsible for the value of the objective function γ, this is accepting a step that ultimately increases the objective function to be minimized. Conversely, $\psi(x, \gamma)$ may increase when $\max \Lambda_i$ decreases implying a rejection of a step that improves the worst case objective.

Following the lines of Brayton et al. [32], a solution is therefore to set $\psi(x)$ equal to the worst case objective, i.e.,

$$\psi(x) = \max_i \Lambda_i . \tag{2-54}$$

A problem in the Goal Attainment method is that it is common to use a weighting coefficient equal to zero to incorporate hard constraints. The merit function of Eq. 2-54 then becomes infinite for arbitrary violations of the constraints. To overcome this problem while still retaining the features of Eq. 2-54 the merit function is combined with that of Eq. 2-43 giving the following:

$$\psi(x) = \sum_{i=1}^{m} \begin{cases} r_i \cdot \max \{0, F_i(x) - w_i\gamma - F_i^*\} & \text{if } w_i = 0 \\ \max_i \Lambda_i, \ i = 1, \ldots, m & \text{otherwise} \end{cases} \tag{2-55}$$

Another feature that can be exploited in SQP is the objective function γ. From the KT equations (Eq. 2-24) it can be shown that the approximation to the Hessian of the Lagrangian, H, should have zeros in the rows and columns associated with the variable γ. By initializing H as the identity matrix, this property does not appear. H is therefore initialized and maintained to have zeros in the rows and columns associated with γ.

These changes make the Hessian, H, indefinite, therefore H is set to have zeros in the rows and columns associated with γ, except for the diagonal element, which is set to a small positive number (e.g., 1e–10). This allows use of the fast converging positive definite QP method described in the "Quadratic Programming Solution" section.

The above modifications have been implemented in `attgoal` and have been found to make the method more robust. However, due to the rapid convergence of the SQP method, the requirement that the merit function strictly decrease sometimes requires more function evaluations than an implementation of SQP using the merit function of (Eq. 2-43).

Review

A number of different optimization strategies have been discussed. The algorithms used (e.g., BFGS, Levenberg-Marquardt and SQP) have been chosen for their robustness and iterative efficiency. The choice of problem formulation (e.g., unconstrained, least squares, constrained, minimax, multiobjective, or goal attainment) depends on the problem being considered and the required execution efficiency.

References

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

[2] R. Fletcher, "Practical Methods of Optimization," Vol. 1, Unconstrained Optimization, and Vol. 2, Constrained Optimization, John Wiley and Sons., 1980.

[3] C.G. Broyden, "The Convergence of a Class of Double-rank Minimization Algorithms," J.Inst. Maths. Applics., Vol. 6, pp. 76-90, 1970.

[4] R. Fletcher, "A New Approach to Variable Metric Algorithms," Computer Journal, Vol. 13, pp. 317-322, 1970.

[5] D. Goldfarb, "A Family of Variable Metric Updates Derived by Variational Means," Mathematics of Computing, Vol. 24, pp. 23-26, 1970.

[6] D.F. Shanno, "Conditioning of Quasi-Newton Methods for Function Minimization," Mathematics of Computing, Vol. 24, pp. 647-656, 1970.

[7] W.C. Davidon, "Variable Metric Method for Minimization," A.E.C. Research and Development Report, ANL-5990, 1959.

[8] R. Fletcher and M.J.D. Powell, "A Rapidly Convergent Descent Method for Minimization," Computer J., Vol. 6, pp. 163-168, 1963.

[9] M.C. Biggs, "Constrained Minimization Using Recursive Quadratic Programming," Towards Global Optimization (L.C.W.Dixon and G.P.Szergo, eds.), North-Holland, pp.341-349, 1975.

[10] S.P. Han, "A Globally Convergent Method for Nonlinear Programming," J. Optimization Theory and Applications, Vol. 22, p. 297, 1977.

[11] M.J.D. Powell, "A Fast Algorithm for Nonlinearly Constrained Optimization Calculations," Numerical Analysis, G.A.Watson ed., Lecture Notes in Mathematics, Springer Verlag, Vol. 630, 1978.

[12] M.J.D. Powell, "The Convergence of Variable Metric Methods for Nonlinearly Constrained Optimization Calculations," Nonlinear Programming 3, (O.L. Mangasarian, R.R. Meyer and S.M. Robinson, eds.), Academic Press, 1978.

[13] M.J.D. Powell, "Variable Metric Methods for Constrained Optimization," Mathematical Programming: The State of the Art, (A.Bachem, M.Grotschel and B.Korte, eds.) Springer Verlag, pp. 288-311, 1983.

[14] W. Hock, and K. Schittowski, "A Comparative Performance Evaluation of 27 Nonlinear Programming Codes," Computing, Vol. 30, pp. 335, 1983.

[15] G. Dantzig, Linear Programming and Extensions, Princeton University Press, Princeton, 1963.

[16] P.E. Gill, W. Murray, and M.H. Wright, Numerical Linear Algebra and Optimization, Vol.1, Addison Wesley, 1991.

[17] P.E. Gill, W. Murray, M.A. Saunders, and M.H. Wright, "Procedures for Optimization Problems with a Mixture of Bounds and General Linear Constraints," ACM Trans. Math. Software, Vol.10, pp.282-298, 1984.

[18] K. Levenberg, "A Method for the Solution of Certain Problems in Last Squares," Quart. Apl. Math. Vol.2, pp.164-168, 1944.

[19] D. Marquardt, "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," SIAM J. Appl. Math. Vol.11, pp. 431-441, 1963.

[20] J.J. Moré, "The Levenberg-Marquardt Algorithm: Implementation and Theory," Numerical Analysis, ed. G. A. Watson, Lecture Notes in Mathematics 630, Springer Verlag, pp.105-116, 1977.

[21] J.E. Dennis, Jr. "Nonlinear Least Squares," State of the Art in Numerical Analysis ed. D. Jacobs, Academic Press., pp. 269-312, 1977.

[22] K. Schittowski, "NLQPL: A FORTRAN-Subroutine Solving Constrained Nonlinear Programming Problems," Annals of Operations Research, Vol. 5, pp. 485-500, 1985.

[23] NAG Fortran Library Manual, Mark 12, Vol.4, E04UAF, p.16.

[24] Y. Censor, "Pareto Optimality in Multiobjective Problems, " Appl. Math. Optimiz., Vol. 4, pp. 41-59, 1977.

[25] L.A. Zadeh, "Optimality and Nonscalar-valued Performance Criteria," IEEE Trans. Automat. Contr., Vol. AC-8, p. 1, 1963.

[26] N.O. Da Cunha and E. Polak, "Constrained Minimization Under Vector-valued Criteria in Finite Dimensional Spaces," J.Math. Anal. Appl., Vol. 19, pp. 103-124, 1967.

[27] F.W. Gembicki, "Vector Optimization for Control with Performance and Parameter Sensitivity Indices," Ph.D. Dissertation, Case Western Reserve Univ., Cleveland, Ohio, 1974.

[28] S.H. Hollingdale, Methods of Operational Analysis in Newer Uses of Mathematics (James Lighthill, ed.), Penguin Books, 1978.

[29] P.J. Fleming, "Application of Multiobjective Optimization to Compensator Design for SISO Control Systems," Electronics Letters, Vol.22, No.5, pp.258-259, 1986.

[30] P.J. Fleming, "Computer-Aided Control System Design of Regulators using a Multiobjective Optimization Approach," Proc. IFAC Control Applications of Nonlinear Porg. and Optim., Capri, Italy, pp. 47-52, 1985.

[31] F.M. Waltz, "An Engineering Approach: Hierarchical Optimization Criteria," IEEE Trans., Vol. AC-12, pp. 179-180, April, 1967.

[32] R.K. Brayton, S.W. Director, G.D. Hachtel, and L.Vidigal, “A New Algorithm for Statistical Circuit Design Based on Quasi-Newton Methods and Function Splitting,” IEEE Transactions on Circuits and Systems, Vol. CAS-26, pp. 784-794, Sept. 1979.

[33] J.A. Nelder, and R. Mead, “A Simplex Method for Function Minimization,” Computer J., Vol.7, pp. 308-313.

[34] A.C.W. Grace, “Computer–Aided Control System Design Using Optimization Techniques”, Ph.D. Thesis, University of Wales, Bangor, Gwynedd, UK, 1989.

[35] K. Madsen and H. Schjaer-Jacobsen, “Algorithms for Worst Case Tolerance Optimization,” IEEE Transactions of Circuits and Systems, Vol. CAS-26, Sept. 1979.

[36] G.F. Forsythe, M.A. Malcolm, and C.B. Moler, Computer Methods for Mathematical Computations, Prentice Hall, 1976.

[37] G.B. Dantzig, A. Orden, and P. Wolfe, “Generalized Simplex Method for Minimizing a Linear from Under Linear Inequality Constraints,” Pacific J. Math. Vol. 5, pp. 183–195.

3

Reference

This chapter contains descriptions of the Optimization Toolbox functions, listed alphabetically. Information is also available through the online Help facility.

Nonlinear Minimization

Function	Purpose
attgoal	Multiobjective goal attainment
constr	Constrained nonlinear minimization
fmin	Scalar nonlinear minimization
fminu, fmins	Unconstrained nonlinear minimization
lp	Linear programming
minimax	Minimax optimization
qp	Quadratic programming
seminf	Semi-infinite minimization

Equation Solving

Function	Purpose
\	Linear equation solving (see MATLAB Language Reference guide)
fsolve	Nonlinear equation solving
fzero	Scalar nonlinear equation solving

Least-Squares (Curve fitting)

Function	Purpose
\	Linear least squares (see MATLAB Language Reference guide)
conls	Constrained linear least squares
curvefit	Nonlinear curve fitting
leastsq	Nonlinear least squares
nnls	Nonnegative linear least squares

Utility

Function	Purpose
foptions	Parameter settings

Demonstrations

Function	Purpose
bandemo	Minimization of the banana function
dfildemo	Finite-precision filter design (requires Signal Processing Toolbox)
goaldemo	Goal attainment example
optdemo	Menu of demonstration routines
tutdemo	Tutorial walk-through

Purpose

Solve multiobjective goal attainment problem,

$$\underset{x,\gamma}{\text{minimize}} \ \gamma \qquad \text{such that} \qquad F(x) - w\gamma \leq \text{goal}$$

given $F(x)$, w and $goal$, where x, w, and $goal$ are vectors, γ is a scalar variable, and $F(x)$ is a function that returns a vector value.

Synopsis

```
x = attgoal('fun',x0,goal,w)
x = attgoal('fun',x0,goal,w,options)
x = attgoal('fun',x0,goal,w,options,vlb,vub)
x = attgoal('fun',x0,goal,w,options,vlb,vub,'grad')
x = attgoal('fun',x0,goal,w,options,vlb,vub,'grad',p1,p2,...)
[x,options] = attgoal('fun',x0,...)
```

Description

`attgoal` solves the goal attainment problem, which is one formulation for minimizing a multiobjective optimization problem.

`x = attgoal('fun',x0,goal,w)` starts at `x0` and solves the goal attainment problem, given a weight vector `w` and a goal vector `goal`, for the function defined in the M-file `fun.m`.

`x = attgoal('fun',x0,goal,w,options)` uses the parameter values in the vector `options` rather than the default option values.

`x = attgoal('fun',x,goal,w,options,vlb,vub)` defines a set of lower and upper bounds on `x` through the matrices `vlb` and `vub`. This restricts the solution to the range `vlb <= x <= vub`.

`x = attgoal('fun',x0,goal,w,options,vlb,vub,'grad')` uses the gradient information calculated by the function `grad`, defined in the M-file `grad.m`, rather than the default of approximating the partial derivatives via finite differencing.

`x = attgoal('fun',x0,goal,w,options,vlb,vub,'grad',p1,p2,...)` passes the problem-dependent parameters `p1`, `p2`, etc., directly to the functions `fun` and `grad`.

`[x, options] = attgoal('fun',x0,goal,w)` returns the parameters used in the optimization method. For example, `options(10)` contains the number of function evaluations used.

Arguments

fun

A string containing the name of the function that computes the objective function to be minimized at the point x. The function fun returns one argument: a vector value f,

```
f = fun(x)
```

attgoal attempts to minimize the values in the vector f to attain the goal values given by goal.

Alternatively, a string expression can be used with x representing the independent variables. For example,

```
x = attgoal('sin(x.*x)',x0,goal,w)
```

To make an objective function as near as possible to a goal value, (i.e., neither greater than nor less than) set `options(15)` to the number of objectives required to be in the neighborhood of the goal values. Such objectives must be partitioned into the first elements of the vector f returned by fun.m

goal

Vector of values that the objectives attempt to attain. Prior to the optimization, it is generally unknown whether the objectives will be minimized less than the goals (over attainment), or will only approach the goals (under attainment).

w

A weighting vector to control the relative under-attainment or over-attainment of the objectives. When the values of goal are all nonzero, to ensure the same percentage of under- or over-attainment of the active objectives, set the weighting function w = abs(goal). (The active objectives are the set of objectives that are barriers to further improvement of the goals at the solution.) When the weighting function w is positive, attgoal attempts to make the objectives less than the goal values. To make the objective functions greater than the goal values, set w to be negative rather than positive.To make an objective function as near as possible to a goal value is described below under fun.

options

A vector of control parameters. Of the 18 elements of options, the input options used by attgoal are: 1, 2, 3, 4, 7, 9, 14, 15, 16, 17. When options is an output parameter, the options used by attgoal to return values are: 8, 10, 11, 18.

- options(1) controls display. Setting this to a value of 1 produces a tabular display of intermediate results.
- options(2) controls the accuracy of x at the solution.
- options(3) controls the accuracy of f at the solution.
- options(4) sets the maximum constraint violation that is acceptable.

The termination criteria involving options(2), options(3), and options(4) must all hold true for the algorithm to terminate.

The use of options(15) by attgoal is discussed under the description of fun above. The use of options(7) and options(8) by attgoal is discussed in the "Algorithm" section below. For more information on the options vector, including default settings, see the foptions reference page and the "Default Parameters Settings" section in the Tutorial.

grad — A string containing the name of the function that computes the gradient of the function at the point x. This function has the form

```
df = grad(x)
```

The variable df is a matrix where the columns of df contain the partial derivatives for each of the objectives respectively, (i.e., the i th column of df corresponds to the partial derivative of the i th objective with respect to each of the elements in x).

x0, p1, p2, . . . , vlb, vub — See constr.

Examples

Consider a linear system of differential equations.

An output feedback controller, K, is designed producing a closed loop system

$$\dot{x} = (A + BKC)x + Bu$$
$$y = Cx$$

The eigenvalues of the closed loop system are determined from the matrices A, B, C, and K using the command eig(A+B*K*C). Closed loop eigenvalues must lie on the real axis in the complex plane to the left of the points [–5,–3,–1]. In order not to saturate the inputs, no element in K can be greater than 4 or be less than –4.

The system is a two-input, two-output, open loop, unstable system, with state-space matrices.

$$A = \begin{bmatrix} -0.5 & 0 & 0 \\ 0 & -2 & 10 \\ 0 & 1 & -2 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 \\ 2 & 2 \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The set of goal values for the closed loop eigenvalues are initialized as

```
goal  = [–5, –3, –1];
```

To ensure the same percentage of under- or over-attainment in the active objectives at the solution, the weighting matrix, `w`, is set to `abs(goal)`.

Starting with a controller, `K = [0, 0;  0, 0]`, first write an M-file.

Step 1: Write an M-file fun.m:

```
function F  = fun(K, A, B, C)
F = sort(eig(A+B*K*C));  % Evaluate objectives
```

Step 2: Enter system matrices and invoke an optimization routine:

```
A = [–0.5 0 0;  0 –2 10;  0 1 –2];
B = [1 0;  2 2;  0 1];
C = [1 0 0;  0 0 1];
K = zeros(2, 2)              % Initialize controller matrix
goal  = [–5  –3  –1];        % Set goal values for the
                             % eigenvalues
w = abs(goal)                % Set w for same percentage
                             % attainment
vlb = –4*ones(size(K));      % Set lower bounds on the
                             % controller
vub = 4*ones(size(K));       % Set upper bounds on the
                             % controller
options = 1;                 % Set display parameter
[K, options] = ...
   attgoal('fun', K, goal, w, options, vlb, vub, [ ], A, B, C)
```

This example can be run by using the demonstration script `goaldemo`. After 118 function evaluations, a solution is

```
Active constraints:
     1
     2
K =
   -4.0000    0.2564
    4.0000   -4.0000
fun(K,A,B,C)
ans  =
   -6.9313
   -4.1588
   -1.4099
```

The attainment factor is `options(8)`

```
options(8)
ans  =
   -0.3863
```

Discussion

The attainment factor indicates that each of the objectives has been over-achieved by at least 38.63% over the original design goals. The active constraints, in this case constraints 1 and 2, are the objectives that are barriers to further improvement and for which the percentage of over-attainment is met exactly.

In the above design, the optimizer tries to make the objectives less than the goals. For a worst case problem where the objectives must be as near to the goals as possible, set `options(15)` to the number of objectives for which this is required.

Consider the above problem when you want eigenvalues to be equal to the goal values. A solution to this problem is found by invoking `attgoal` with `options(15)` set to 3.

```
options(15) = 3;
[K,options] = ...
attgoal('fun',K,goal,w,options,vlb,vub,[ ],A,B,C)
```

After 37 function evaluations the solution is

```
K =
    -2.4294    -0.4891
     3.9999    -2.0706

fun(K,A,B,C)
ans =
    -5.0000
    -3.0000
    -1.0000
```

The attainment factor is

```
options(8)
ans =
    1.0859e-20
```

In this case the optimizer has tried to match the objectives to the goals. The attainment factor of 1.0859e–20 indicates that the goals have been matched almost exactly.

Notes

This problem has discontinuities when the eigenvalues become complex; this explains why the convergence is slow. Although the underlying methods are based on functions that are continuous, the method is able to make steps toward the solution since the discontinuities do not occur at the solution point. When the objectives and goals are complex, `attgoal` tries to achieve the goals in a least-squares sense.

Algorithm

Multiobjective optimization concerns the minimization of a set of objectives simultaneously. One formulation for this problem, and implemented in `attgoal`, is the goal attainment problem of Gembicki[1]. This entails the construction of a set of goal values for the objective functions. Multiobjective optimization is discussed fully in the Introduction to Algorithms chapter.

In this implementation, the slack variable γ is used as a dummy argument to minimize the vector of objectives $F(x)$ simultaneously; `goal` is a set of values that the objectives attain. Generally, prior to the optimization, it is unknown whether the objectives will reach the goals (under attainment) or be minimized less than the goals (over attainment). A weighting vector, `w`, controls the relative under-attainment or over-attainment of the objectives.

attgoal uses a Sequential Quadratic Programming (SQP) method, which is described fully in the Introduction to Algorithms chapter. Modifications are made to the line search and Hessian. In the line search an exact merit function (see [5] and [6]) is used together with the merit function proposed by [2, 3]. The line search is terminated when either merit function shows improvement. A modified Hessian, which takes advantage of special structure of this problem, is also used (see [5] and [6]). A full description of the modifications used is found in the "Goal Attainment Method" section of the Introduction to Algorithms. Setting options(7) = 1 uses the merit function and Hessian used in constr.

options(8) contains the value of γ at the solution. A negative value of γ indicates over attainment in the goals.

See also SQP implementation section in the Introduction to Algorithms chapter for more details on the algorithm used and the display of procedures for options(1) = 1 setting.

Limitations

The objectives must be continuous. attgoal may give only local solutions.

See Also

constr, foptions

References

[1] F.W. Gembicki, "Vector Optimization for Control with Performance and Parameter Sensitivity Indices," Ph.D. Dissertation, Case Western Reserve Univ., Cleveland, Ohio, 1974.

[2] S.P. Han, "A Globally Convergent Method For Nonlinear Programming," Journal of Optimization Theory and Applications, Vol. 22, p. 297, 1977.

[3] M.J.D. Powell, "A Fast Algorithm for Nonlineary Constrained Optimization Calculations," Numerical Analysis, ed. G.A. Watson, Lecture Notes in Mathematics, Springer Varleg, Vol. 630, 1978.

[4] P.J. Fleming and A.P. Pashkevich, Computer Aided Control System Design Using a Multi-Objective Optimisation Approach, Control 1985 Conference, Cambridge, UK, p. 174-179.

[5] R.K. Brayton, S.W. Director, G.D. Hachtel, and L.Vidigal, "A New Algorithm for Statistical Circuit Design Based on Quasi–Newton Methods and Function Splitting," IEEE Transactions on Circuits and Systems, Vol. CAS-26, pp. 784–794, Sept. 1979.

[6] A.C.W. Grace, "Computer–Aided Control System Design Using Optimization Techniques", Ph.D. Thesis, University of Wales, Bangor, Gwynedd, UK, 1989.

conls

Purpose

Solve the constrained linear least-squares problem,

$$\min_{x} \frac{1}{2}\|Ax - b\|_2^2 \qquad \text{such that} \qquad Cx \le d$$

where A and C are matrices and b, d, and x are vectors.

Synopsis

```
x = conls(A, b, C, d)
x = conls(A, b, C, d, vlb)
x = conls(A, b, C, d, vlb, vub)
x = conls(A, b, C, d, vlb, vub, x0)
x = conls(A, b, C, d, vlb, vub, x0, neqcstr)
x = conls(A, b, C, d, vlb, vub, x0, neqcstr, display)
[x, lambda, how] = conls(A, b, C, d, ...)
```

Description

`conls` solves the constrained linear least-squares problem.

`x = conls(A, b, C, d)` returns a vector `x` that finds the least-squares solution to `Ax=b` subject to `C*x <= d`.

`x = conls(A, b, C, d, vlb, vub)` sets lower and upper bounds on `x`. This restricts the solution to the range `vlb <= x <= vub`.

`x = conls(A, b, C, d, vlb, vub, x0)` sets the initial starting point to `x0`.

`x = conls(A, b, C, d, vlb, vub, x0, neqcstr)` specifies that the first `neqcstr` constraints are equality constraints.

`x = conls(A, b, C, d, vlb, vub, x0, neqcstr, display)` controls the display of warning messages.

`[x, lambda] = conls(A, b, C, d)` returns values for the Lagrange multipliers at the solution in the variable `lambda`.

`[x, lambda, how] = conls(A, b, C, d)` also returns a string `how` that indicates error conditions at the final iteration.

Arguments

A, b	The matrix A and vector b form the set of coefficients of the over- or under-determined linear system to be solved.

C, d	The matrix C and vector d are the coefficients of the linear constraints. The coefficients for the equality constraints must be partitioned into the first rows of C and the first elements of d.
vlb, vub	Upper and lower bound vectors. The variables, vlb and vub, are normally the same size as x. However, if vlb has n elements and less elements than x then only the first n elements in x are bounded below; upper bounds in vub are defined in the same manner.
x0	Starting vector. conls generally starts its search at the point zeros(size(x)). Setting the initial starting point can result in faster convergence. If the problem is badly conditioned, this can also result in an improved solution.
neqcstr	Number of equality constraints.
display	Flag to control the display of warning messages. The default value for the parameter display is 0, which displays warning messages. A value of –1 suppresses warning messages.
lambda	A vector that returns the set of Lagrange multipliers at the solution. The length of lambda is length(b)+length(vlb)+length(vub) and the Lagrange multipliers are given in the corresponding order: first the multipliers for A, then vlb, then vub.
how	A string that indicates error conditions at the solution. The string how='infeasible' indicates that the problem is infeasible (i.e., the constraints are overly restrictive); how='unbounded' indicates that the problem has an unbounded solution; how = 'dependent' indicates that dependent equality constraints were detected and removed; how='ok' indicates that the problem was solved without difficulty.

As with all Optimization Toolbox functions, empty matrices in the calling sequence result in the use of default options. For example, the command

```
conls(A, b, C, d, [ ], [ ], [ ], length(b))
```

indicates that the problem is an equality constrained problem, having no upper or lower bounds on the variables, and using a default starting point.

Examples

Find the least-squares solution to the over-determined system $Ax = b$ subject to $Cx \le d$ and $vlb \le x \le vub$.

Step 1: Enter the coefficient matrices:

```
A =
    0.9501    0.7620    0.6153    0.4057
    0.2311    0.4564    0.7919    0.9354
    0.6068    0.0185    0.9218    0.9169
    0.4859    0.8214    0.7382    0.4102
    0.8912    0.4447    0.1762    0.8936
b =
    0.0578
    0.3528
    0.8131
    0.0098
    0.1388
C =
    0.2027    0.2721    0.7467    0.4659
    0.1987    0.1988    0.4450    0.4186
    0.6037    0.0152    0.9318    0.8462
d =
    0.5251
    0.2026
    0.6721
vlb = 0.1*ones(4,1);
vub = 2*ones(4,1);
```

Step 2: Invoke the constrained linear least-squares routine:

```
[x,lambda] = conls(A,b,C,d,vlb,vub)
```

This generates the solution

```
x =
   -0.1000
   -0.1000
    0.2152
    0.3502
lambda =
         0
    0.2392
         0
    0.0409
    0.2784
         0
         0
         0
         0
         0
         0
```

The first three elements of the Lagrange multipliers (i.e., `lambda`) are associated with the inequality constraints. Nonzero elements of `lambda` indicate active constraints at the solution. In this case, the second linear inequality constraint and the first two lower bound constraints are active constraints (i.e., the solution is on their constraint boundaries).

The last two elements of the Lagrange multipliers are associated with the lower bounds on `x`. In this case, the bounds are inactive.

Algorithm

`conls` is based on `qp`, which uses an active set method similar to that described in [1]. It finds an initial feasible solution by first solving a linear programming problem. See the quadratic programming method discussed in the Introduction to Algorithms chapter.

Diagnostics

`conls` gives a warning when the solution is infeasible:

```
Warning: The constraints are overly stringent;
         there is no feasible solution.
```

In this case, `conls` produces a result that minimizes the worst case constraint violation.

When the equality constraints are inconsistent, conls gives

```
Warning: The equality constraints are overly stringent;
     there is no feasible solution.
```

Unbounded solutions, which can occur when the Hessian H is negative semidefinite, may result in

```
Warning: The solution is unbounded and at infinity;
     the constraints are not restrictive enough.
```

In this case, conls returns a value of x that satisfies the constraints.

Notes

For problems with no constraints, \ should be used: x= A\b.

See Also

qp, \, nnls.

References

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

Purpose

Find the minimum of a constrained nonlinear multivariable function,

$$\min_{x} f(x) \quad \text{such that} \quad G(x) \leq 0$$

where x is a vector, G(x) is a function that returns a vector, and f(x) is a function that returns a scalar. Both f(x) and G(x) can be nonlinear functions. G(x) can define both equality and inequality constraints.

Synopsis

```
x = constr('fun',x0)
x = constr('fun',x0,options)
x = constr('fun',x0,options,vlb,vub,'grad')
x = constr('fun',x0,options,vlb,vub,'grad',p1,p2,...)
[x,options] = constr('fun',x0,...)
[x,options,lambda] = constr('fun',x0,...)
[x,options,lambda,hess] = constr('fun',x0,...)
```

Description

`constr` finds the constrained minimum of a scalar function of several variables starting at an initial estimate. This is generally referred to as constrained nonlinear optimization.

`x = constr('fun',x0)` starts at the point `x0` and finds a minimum of the function and constraints defined in the M-file named `fun.m`.

`x = constr('fun',x0,options)` uses the parameter values in the vector `options` rather than the default option values.

`x = constr('fun',x,options,vlb,vub)` defines a set of lower and upper bounds on `x` through the matrices `vlb` and `vub`. This restricts the solution to the range `vlb <= x <= vub`.

`x = constr('fun',x0,options,vlb,vub,'grad')` uses the gradient information calculated by the function `grad`, defined in the M-file `grad.m`, rather than the default of approximating the partial derivatives via finite differencing.

`x = constr('fun',x0,options,vlb,vub,'grad',p1,p2,...)` passes the problem-dependent parameters `p1`, `p2`, etc., directly to the functions `fun` and `grad`.

`[x, options] = constr('fun', x0)` returns the parameters used in the optimization method. For example, `options(10)` contains the number of function evaluations used.

`[x, options, lambda] = constr('fun', x0)` returns the vector `lambda` of the Lagrange multipliers at the solution `x`.

`[x, options, lambda, hess] = constr('fun', x0)` also returns the approximation to the Hessian at the final iteration.

Arguments

`x0` — Starting vector.

`fun` — A string containing the name of the function that computes the objective function to be minimized and the constraint function at the point `x`. The function `fun` returns two arguments: a scalar valued function `f` to be minimized and a vector of constraint values `g`,

```
[f, g] = fun(x)
```

When inequality constraints are present, the objective function `f` is minimized such that `g <= zeros(size(g))`.

Equality constraints, when present, are placed in the first elements of `g`. When using equality constraints, `options(13)` must be set to the number of equality constraints (see the "Equality Constrained Example" section in the Tutorial).

Alternatively, a string expression can be used with `x` representing the independent variables and with `f` and `g` representing the function and constraints. For example,

```
x = constr('f = fun(x); g = cstr(x);', x0)
```

`vlb, vub` — Upper and lower bound vectors. The variables, `vlb` and `vub`, are normally the same size as `x`. However, if `vlb` has `n` elements and fewer elements than `x`, then only the first `n` elements in `x` are lower bounded; upper bounds in `vub` are defined in the same manner.

options

A vector of control parameters. Of the 18 elements of options, the input options used by constr are: 1, 2, 3, 4, 9, 13, 14, 16, 17. When options is an output parameter, the options used by constr to return values are: 8, 10, 11, 18.

- options(1) controls display. Setting this to a value of 1 produces a tabular display of intermediate results.
- options(2) controls the accuracy of x at the solution.
- options(3) controls the accuracy of f at the solution.
- options(4) sets the maximum constraint violation that is acceptable.

The termination criteria involving options(2), options(3), and options(4) must all hold true for the algorithm to terminate.

For more information on the options vector, including default settings, see the foptions reference page and the "Default Parameters Settings" section in the Tutorial.

grad

A string containing the name of the function that computes the gradient of the function and the gradient of the constraints at the point x. This function has the form

```
[df, dg] = grad(x)
```

The variable df is a vector that contains the partial derivatives of f with respect to x. The variable dg is a matrix where the columns of dg contain the partial derivatives for each of the constraints respectively, (i.e., the i th column of dg corresponds to the partial derivative of the i th constraint with respect to each of the elements in x).

p1, p2, . . . Additional arguments to be passed to fun, that is, when constr calls fun, and grad when it exists, the calls are

```
[f, g] = fun(x, p1, p2, ...)
[df, dg] = grad(x, p1, p2, ...
```

Using this feature, the same M-file can solve a number of similar problems with different parameters while avoiding the need to use global variables. Note that since all the arguments preceding p1, p2, etc., in the call to constr. mmust be defined, empty matrices may be passed in for options, vlb, vub, and 'grad' to indicate that default arguments are to be used, as in

```
x = constr('fun', x0, [], [], [], [], p1, p2, ...)
```

lambda A vector that returns the set of Lagrange multipliers at the solution. The length of lambda is length(g)+length(vlb)+length(vub) and the Lagrange multipliers are given in the corresponding order: first the multipliers for g, then vlb, then vub.

hess The Quasi-Newton approximation to the Hessian matrix at the final iteration.

Examples

Find values of x that minimize $f(x) = -x_1x_2x_3$, starting at the point $x = [10 \quad 10 \quad 10]$ and subject to the constraints

$$-x_1 - 2x_2 - 2x_3 \le 0$$

$$x_1 + 2x_2 + 2x_3 \le 72.$$

Step 1: Write an M–file:

```
function [f, g] = fun(x)
f = -x(1) * x(2) * x(3);
g(1) = -x(1) - 2 * x(2) - 2 * x(3); % Evaluate Constraints
g(2) =  x(1) + 2 * x(2) + 2 * x(3) - 72;
```

Step 2: Invoke an optimization routine:

```
x0 = [10, 10, 10];          % Starting guess at the solution
x = constr('fun', x0)       % Invoke optimizer
```

After 49 function evaluations, the solution is

```
x =
    24.0000    12.0000    12.0000
[f,g] = fun(x)
f =
    -3.4560e+03
g =
   -72     0
```

Algorithm

constr uses a Sequential Quadratic Programming (SQP) method. In this method, a Quadratic Programming (QP) subproblem is solved at each iteration. An estimate of the Hessian of the Lagrangian is updated at each iteration using the BFGS formula (see fminu, references [3, 6]).

A line search is performed using a merit function similar to that proposed by [1] and [2, 3]. The QP subproblem is solved using an active set strategy similar to that described in [4]. A full description of this algorithm is found in the "Constrained Optimization" section of the Introduction to Algorithms chapter.

See also SQP implementation section in the Introduction to Algorithms chapter for more details on the algorithm used and the display of procedures for options(1) = 1 setting.

Limitations

The function to be minimized and the constraints must both be continuous. constr may only give local solutions.

When the problem is infeasible, constr attempts to minimize the maximum constraint value.

The objective function and constraint function must be real-valued, that is they cannot return complex values.

Notes

If equality constraints are present and dependent equalities are detected and removed in the quadratic subproblem, 'dependent' will be printed under the Procedures heading (when output is asked for using options(1)=1). The dependent equalities are only removed when the equalities are consistent. If the system of equalities is not consistent, the subproblem is infeasible and 'infeasible' will be printed under the Procedures heading.

See Also

fminu, foptions

References

[1] S.P. Han, “A Globally Convergent Method for Nonlinear Programming,” Journal of Optimization Theory and Applications, Vol. 22, 1977, p. 297.

[2] M.J.D. Powell, “The Convergence of Variable Metric Methods For Nonlinearly Constrained Optimization Calculations,” Nonlinear Programming 3, (O.L. Mangasarian, R.R. Meyer, and S.M. Robinson, eds.) Academic Press, 1978.

[3] M.J.D. Powell, “A Fast Algorithm for Nonlineary Constrained Optimization Calculations,” Numerical Analysis, ed. G.A. Watson, Lecture Notes in Mathematics, Springer Verlag, Vol. 630, 1978.

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

Purpose

Solve nonlinear curve-fitting (data-fitting) problems in the least-squares sense. That is, given input data xdata, the observed output ydata, we want to find coefficients x that "best-fit" the equation F(x, xdata)

$$\min_x \frac{1}{2}\|F(x,xdata)-ydata\|_2^2 = \frac{1}{2}\sum_i (F(x,xdata_i)-ydata_i)^2$$

where xdata and ydata are vectors and F(x, xdata) is a vector valued function.

The function `curvefit` uses the same algorithm as `leastsq`. Its purpose is to provide an interface designed specifically for data-fitting problems.

Synopsis

```
x = curvefit('fun',x0,xdata,ydata)
x = curvefit('fun',x0,xdata,ydata,options)
x = curvefit('fun',x0,xdata,ydata,options,'grad')
x = curvefit('fun',x0,xdata,ydata,options,'grad',p1,p2,...)
[x,options] = curvefit('fun',x0,xdata,ydata ...)
[x,options,funval] = curvefit('fun',x0,xdata,ydata ...)
[x,options,funval,jacob] = curvefit('fun',x0,xdata,ydata ...)
```

Description

`curvefit` solves nonlinear data-fitting problems.

`curvefit` requires an user-defined function to compute the vector-valued function F(x, xdata). The size of the vector returned by the user-defined function must be the same as the size of ydata.

`x = curvefit('fun',x0,xdata,ydata)` starts at `x0` and finds the least squares minimum of the functions described in the M-file `fun`.

`x = curvefit('fun',x0,xdata,ydata,options)` uses the parameter values in the vector options rather than the default option values.

`x = curvefit('fun',x0,xdata,ydata,options,'grad')` calls the function `grad` to obtain the partial derivatives of the functions

`x = curvefit('fun',x0,xdata,ydata,options,'grad',p1,p2,...)` passes parameters (i.e., `p1`, `p2`, etc.), directly to the function `fun`.

`[x, options] = curvefit('fun', x0, xdata, ydata)` returns the parameters used in the optimization. For example, `options(10)` contains the number of function evaluations used.

`[x, options, funval] = curvefit('fun', x0, xdata, ydata)` returns the function value `fun(x)` at the solution `x`.

`[x, options, funval, jacob] = curvefit('fun', x0, xdata, ydata)` also returns the approximation to the Jacobian of the function at the solution `x`.

Arguments

fun — A string containing the name of the function that computes the equation to be fitted evaluated at the point x. The function `fun` returns one argument: a vector-valued function `f` to be minimized,

```
f = fun(x, xdata)
```

NOTE The sum of squares should not be formed explicitly. Instead your function should return a vector of function values. See the examples below.

---

grad — A string containing the name of the function that computes the gradient of the objective functions at the point x. This function has the form

```
df = grad(x, xdata)
```

The variable `df` is a matrix that contains the partial derivatives of F with respect to x. The `i` th column of `df` corresponds to the partial derivative of the `i` th function in `f` with respect to `x`. (This is the transpose of the Jacobian matrix of F(x).)

options

- A vector of control parameters. Of the 18 elements of options, the input options used by curvefit are: 1, 2, 3, 5, 7, 9, 14, 16, 17. When options is an output parameter, the options used by curvefit to return values are: 8, 10, 11, 18.
- options(1) controls display. Setting this to a value of 1 produces a tabular display of intermediate results.
- options(2) controls the accuracy of x at the solution.
- options(3) controls the accuracy of f at the solution.

The termination criteria involving options(2) and options(3) must both hold true for the algorithm to terminate.

The use of options(5) and options(7) by curvefit is discussed in the "Algorithm" section below.

For more information on the options vector, including default settings, see the foptions reference page and the "Default Parameters Settings" section in the Tutorial.

x0, p1, p2, . . . — See fminu.

funval — The value of the function at the solution x.

jacob — The Jacobian of the function at the solution x.

Examples

Say you have a vectors of data xdata and ydata of length n, and you want to find coefficients x to find the best fit to the equation $ydata(i) = x(1) + x(2) \cdot e^{(xdata(i) + x(3))}$; that is, you want to minimize 0

$$\min_{x} \frac{1}{2} \sum_{i=1} (F(x, xdata_i) - ydata_i)^2$$

where $F(x, xdata) = x(1) + x(2) \cdot e^{(xdata(i) + x(3))}$, starting at the point x= [0.3, 0.4, 0.1].

Step 1: Write an M-file:

```
function f = fun(x, xdata)
f = x(1) + x(2)*exp(xdata + x(3)); %Note: f is a vector
```

Step 2: Invoke an optimization routine:

```
% Assume: xdata and ydata exist and are the same size
x0 = [0.3 0.4 0.1]                    % Starting guess
x = curvefit('fun', x0, xdata, ydata)  % Invoke optimizer
```

Note that at the time that `curvefit` is called, we assume that `xdata` and `ydata` both exist and that they are the vectors of the same size. This is necessary as the value `f` returned by `fun` must be the same size as `ydata`.

After 41 function evaluations, this example gives the solution:

```
x =
      0.25783   0.25783
sum(fun(x, xdata) .* fun(x, xdata)) % residual or sum of squares
ans =
      124.3622
```

Algorithm

The choice of algorithm is made by setting `options(5)`. The default is the Levenberg-Marquardt method [1–3]. Setting `options(5) = 1` implements a Gauss-Newton method [4], which is generally faster when the residual $\|F(x, xdata) - ydata\|_2^2$ is small.

The default line search algorithm, `options(7) = 0`, is a safeguarded mixed quadratic and cubic polynomial interpolation and extrapolation method. A safeguarded cubic polynomial method can be selected by setting `options(7) = 1`. This method generally requires fewer function evaluations but more gradient evaluations. Thus, if gradients are being supplied and can be calculated inexpensively, the cubic polynomial line search method is preferable. The algorithms used are described fully in the Introduction to Algorithms chapter.

Limitations

The function to be minimized must be continuous. `curvefit` may only give local solutions.

`curvefit` only handles real variables (the user-defined function must only return real values). When x has complex variables, the variables must be split into real and imaginary parts.

See Also

foptions, lsqnonlin, \, lsqlin, nnls.

References

[1] K. Levenberg, "A Method for the Solution of Certain Problems in Least Squares," Quart. Appl. Math. 2, pp. 164–168, 1944.

[2] D. Marquardt, "An Algorithm for Least–squares Estimation of Nonlinear Parameters," SIAM J. Appl. Math. Vol 11, pp. 431–441, 1963.

[3] J.J. More, "The Levenberg–Marquardt Algorithm: Implementation and Theory," Numerical Analysis, ed. G. A. Watson, Lecture Notes in Mathematics 630, Springer-Verlag, pp. 105–116, 1977.

[4] J.E. Dennis, Jr., "Nonlinear Least Squares", State of the Art in Numerical Analysis, ed. D. Jacobs, Academic Press, pp. 269-312, 1977.

Purpose

Find the minimum of a function of one variable on a fixed interval,

$$\min_{a} f(a) \quad \text{such that} \quad a_1 < a < a_2$$

where a, a_1, and a_2 are scalars and f(a) is a function that returns a scalar.

Synopsis

```
a = fmin('fun', a1, a2)
a = fmin('fun', a1, a2, options)
a = fmin('function', a1, a2, options, p1, p2, ...)
[a, options] = fmin('function', a1, a2, ...)
```

Description

`fmin` finds the minimum of a function of one variable within a fixed interval.

`a = fmin('fun', a1, a2)` returns a value of `x` that is a local minimizer of `fun(a)` on the interval `a1 < a < a2`.

`a = fmin('fun', a1, a2, options)` uses the parameter values in the vector `options` rather than the default option values.

`a = fmin('fun', a1, a2, options, p1, p2, ...)` passes the problem-dependent parameters `p1`, `p2`, etc., directly to the function `fun`.

`[a, options] = fmin('fun', a1, a2)` returns the parameters used in the optimization method. For example, `options(10)` contains the number of function evaluations used.

Arguments

fun — A string containing the name of the function that computes the objective function to be minimized at the point x. The function fun returns one argument: a scalar valued function f to be minimized,

```
[f] = fun(x)
```

Alternatively, an expression can be substituted for the function name, with x representing the independent variable. For example, a = fmin('sin(x*x)',a1,a2) (we have been using a's to emphasize that this function is for one-dimensional problems only; here x must be used as the independent variable in the string expression).

a1, a2 — Interval over which fun is minimized.

options — A vector of control parameters. Of the 18 elements of options, the input options used by fmin are: 1, 2, 14. When options is an output parameter, the options used by fmin to return values are: 8, 10.

- options(1) controls display. Setting this to a value of 1 produces a tabular display of intermediate results.
- options(2) controls the accuracy of x at the solution.
- options(14) sets the maximum number of function evaluations.

For more information on the options vector, including default settings, see the foptions reference page and the "Default Parameters Settings" section in the Tutorial.

p1, p2, ... — Additional arguments to be passed to fun, that is, when fmin calls fun, the call is

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

Using this feature, the same M-file can solve a number of similar problems with different parameters while avoiding the need to use global variables.

Examples

A minimum of sin(a) occurs at

```
a = fmin('sin', 0, 2*pi)
a =
    4.7118
```

The value of the function at the minimum is

```
y = sin(a)
y =
   -1.0000
```

To find the minimum of the function

$$f(a) = (a-3)^2 - 1$$

on the interval (0,5), write an M-file, and then invoke fmin.

Step 1: Write an M–file:

```
function f = fun(a)
f = (a-3).^2 - 1;
```

Step 2: Invoke an optimization routine:

```
a = fmin('fun', 0, 5)
```

This generates the solution

```
a =
    3
```

The value at the minimum is

```
y = f(a)
y =
   -1
```

Algorithm

fmin is an M-file in the MATLAB Toolbox. The algorithm is based on golden section search and parabolic interpolation. A Fortran program implementing the same algorithm is given in [1].

Limitations

The function to be minimized must be continuous. fmin may only give local solutions.

fmin often exhibits slow convergence when the solution is on a boundary interval. In such a case, constr often gives faster and more accurate solutions.

fmin only handles real variables.

See Also

fmins, fminu, foptions

References

[1] G.F. Forsythe, M.A. Malcolm, and C.B. Moler, Computer Methods for Mathematical Computations, Prentice Hall, 1976.

fminu, fmins

Purpose

Find the minimum of an unconstrained multivariable function,

$$\min_{x} f(x)$$

where x is a vector, and $f(x)$ is a function that returns a scalar.

Synopsis

```
x = fminu('fun',x0)
x = fminu('fun',x0,options)
x = fminu('fun',x0,options,'grad')
x = fminu('fun',x0,options,'grad',p1,p2,...)
[x,options] = fminu('fun',x0,...)
[...] = fmins('fun',x0,...)
```

Description

`fminu` and `fmins` find the minimum of a scalar function of several variables, starting at an initial estimate. This is generally referred to as unconstrained nonlinear optimization.

`x = fminu('fun',x0)` starts at the point `x0` and finds a minimum of the function `fun` described in the M-file `fun.m`.

`x = fminu('fun',x0,options)` uses the parameter values in the vector `options` rather than the default option values.

`x = fminu('fun',x0,options,'grad')` uses the gradient information calculated by the function `grad`, defined in the M-file `grad.m`, rather than the default of approximating the partial derivatives via finite differencing.

`x = fminu('fun',x0,options,'grad',p1,p2,...)` passes the problem-dependent parameters `p1`, `p2`, etc., directly to the functions `fun` and `grad`.

`[x,options] = fminu('fun',x0)` returns the parameters used in the optimization method. For example, `options(10)` contains the number of function evaluations used.

Arguments

`x0`	Starting vector.

fun

A string containing the name of the function that computes the objective function to be minimized at the point x. The function fun returns one argument: a scalar valued function f to be minimized,

```
[f] = fun(x)
```

Alternatively, an expression can be substituted for the function name, with x representing the independent variables. For example,

```
x = fminu('sin(x.*x)',x0)
```

options

A vector of control parameters. Of the 18 elements of options, the input options used by fminu are: 1, 2, 3, 6, 7, 9, 14, 16, 17. When options is an output parameter, the options used by fminu to return values are: 8, 10, 11, 18. Of the 18 elements of options, the input options used by fmins are: 1, 2, 3, 14. When options is an output parameter, the options used by fmins to return values are: 8, 10.

- options(1) controls display. Setting this to a value of 1 produces a tabular display of intermediate results.
- options(2) controls the accuracy of x at the solution.
- options(3) controls the accuracy of f at the solution.

The termination criteria involving options(2) and options(3) must both hold true for the algorithm to terminate.

The use of options(6) and options(7) by fminu is discussed in the "Algorithms" section below.

For more information on the options vector, including default settings, see the foptions reference page and the "Default Parameters Settings" section in the Tutorial.

grad — A string containing the name of the function that computes the gradient of the function at the point x. This function has the form

```
df = grad(x)
```

The variable df is a vector that contains the partial derivatives of f with respect to x. Note that this parameter is ignored by fmins as it does not use gradient information.

p1, p2, ... — Additional arguments to be passed to fun, that is, when fminu (fmins) calls fun, and grad when it exists, the calls are

```
[f, g] = fun(x, p1, p2, ...)
[df, dg] = grad(x, p1, p2, ...)
```

Using this feature, the same M-file can solve a number of similar problems with different parameters avoiding the need to use global variables. Note that since all the arguments preceding p1, p2, etc., in the call to fminu.m(fmins.m) must be defined, empty matrices may be passed in for options and 'grad' to indicate that default arguments are to be used, as in

```
x = fminu('fun', x0, [], [], p1, p2, ...)
```

Examples

Find values that minimize

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

starting at the point

$$x = \begin{bmatrix} -1.2 & 1 \end{bmatrix}$$

Step 1: Write an M-file:

```
function f = fun(x)
f = 100*(x(2)-x(1)^2)^2+(1-x(1))^2;      % Cost function
```

Step 2: Invoke an optimization routine:

```
x = [–1, 1]          % Make a starting guess at the solution
x = fminu('fun',x)
```

After 132 function evaluations, this example generates the solution

```
x =
      1.0000  1.0000
fun(x) =
      8.8348e–11
```

Algorithms

fminu: The default algorithm for `fminu` is a quasi-Newton method. This quasi-Newton method uses the BFGS [2–5] formula for updating the approximation of the Hessian matrix. The DFP [7, 8] formula, which approximates the inverse Hessian matrix, is selected by setting `options(6) = 1`. A steepest descent method is selected by setting `options(6) = 2`, although this is not recommended.

For `fminu`, the default line search algorithm, i.e., when `options(7) = 0`, is a safeguarded mixed quadratic and cubic polynomial interpolation and extrapolation method. A safeguarded cubic polynomial method can be selected by setting `options(7) = 1`. This second method generally requires fewer function evaluations but more gradient evaluations. Thus, if gradients are being supplied and can be calculated inexpensively, the cubic polynomial line search method is preferable. A full description of the algorithms is given in the Introduction to Algorithms chapter.

fmins: `fmins` uses the simplex search method of [1]. This is a direct search method that does not use numerical or analytic gradients like `fminu`.

If `n` is the length of `x`, a simplex in `n`-dimensional space is characterized by the `n+1` distinct vectors that are its vertices. In two-space, a simplex is a triangle; in three-space, it is a pyramid. At each step of the search, a new point in or near the current simplex is generated. The function value at the new point is compared with the function's values at the vertices of the simplex and, usually, one of the vertices is replaced by the new point, giving a new simplex. This step is repeated until the diameter of the simplex is less than the specified tolerance.

fmins is generally less efficient than fminu for problems of order greater than two. However, when the problem is highly discontinuous, fmins may be more robust.

fmins has the identical calling syntax as fminu. Note that since fmins does not use gradient information, grad is always ignored.

Limitations

For fminu, the function to be minimized must be continuous. fmins can often handle discontinuity, particularly if it does not occur near the solution. fminu and fmins may only give local solutions.

fminu and fmins only minimize over the real numbers, that is, x must only consist of real numbers and f(x) must only return real numbers. When x has complex variables, they must be split into real and imaginary parts.

NOTE fmins and fminu should not be used to solve problems that are sums-of-squares, that is, of the form: $\min f(x) = f_1(x)^2 + f_2(x)^2 + f_3(x)^2 + \mathrm{L}$ Instead use the leastsq function, which has been optimized for problems of this form, for better performance.

See Also

foptions

References

[1] J.A. Nelder and R. Mead, “A Simplex Method for Function Minimization,” Computer J., Vol. 7, pp. 308–313.

[2] C.G. Broyden, “The Convergence of a Class of Double–rank Minimization Algorithms,” J. Inst. Math. Applic., Vol. 6, pp. 76–90, 1970.

[3] R. Fletcher, “A New Approach to Variable Metric Algorithms,” Computer J., Vol. 13, pp. 317–322, 1970.

[4] D. Goldfarb, “A Family of Variable Metric Updates Derived by Variational Means,” Mathematics of Computing, Vol. 24, pp. 23–26, 1970.

[5] D.F. Shanno,” Conditioning of Quasi–Newton Methods for Function Minimization,” Mathematics of Computing, Vol. 24, pp. 647–656, 1970.

[6] W.C. Davidon, “Variable Metric Method for Minimization,” A.E.C. Research and Development Report, ANL-5990, 1959.

[7] R. Fletcher and M.J.D. Powell, “A Rapidly Convergent Descent Method for Minimization,” Computer J., Vol. 6, pp. 163–168, 1963.

[8] R. Fletcher, “Practical Methods of Optimization,” Vol. 1, Unconstrained Optimization, John Wiley and Sons, 1980.

foptions

Purpose

Set optimization parameters and display parameter values.

Synopsis

```
help foptions
options = foptions
```

Description

The Optimization Toolbox functions `fmin`, `fmins`, `fminu`, `constr`, `attgoal`, `minimax`, `leastsq`, `seminf`, and `fsolve` use optimization parameters that can be changed by setting new values in the `options` vector.

For consistency, the optimization parameters have the same meaning, where possible, throughout the Optimization Toolbox functions.

The function `foptions` returns a set of default options that are used when the `options` vector is not supplied to the appropriate routines. Default values are also used for elements in `options` that are set to 0 and for undefined parameters caused when `options` has fewer than 18 elements. These values may also be returned by specifying a second left-hand argument to the particular optimization routine. For example,

```
[x, options] = fminu('fun', x0);
options(10)
options(11)
```

enables the number of function and gradient evaluations to be obtained and displayed.

See the Table 1-4, “Option Parameters,” on page 25 in the Tutorial chapter for more information.

See Also

`attgoal`, `constr`, `fmin`, `fmins`, `fminu`, `fsolve`, `leastsq`, `minimax`, `seminf`

Purpose

Solve a system of nonlinear equations,

$$F(x) = 0$$

for x, where x is a vector and $F(x)$ is a function that returns a vector value.

Synopsis

```
x = fsolve('fun',x0)
x = fsolve('fun',x0,options)
x = fsolve('fun',x0,options,'grad')
x = fsolve('fun',x0,options,'grad',p1,p2,...)
[x,options] = fsolve('fun',x0,...)
```

Description

`fsolve` finds a root (zero) of a system of nonlinear equations.

`x = fsolve('fun',x0)` starts at `x0` and returns `x`, solving the equations defined in the M-file `fun.m`.

`x = fsolve('fun',x0,options)` uses the parameter values in the vector options rather than the default option values.

`x = fsolve('fun',x0,options,'grad')` uses the gradient information calculated by the function `grad`, defined in the M-file `grad.m`, rather than the default of approximating the partial derivatives via finite differencing.

`x = fsolve('fun',x0,options,'grad',p1,p2,...)` passes the problem-dependent parameters `p1`, `p2`, etc., directly to the functions `fun` and `grad`.

`[x,options] = fsolve('fun',x0)` returns the parameters used in the optimization method. For example, `options(10)` contains the number of function evaluations used.

Arguments

fun

A string containing the name of the function that computes the objective function to be minimized at the point x. The function fun returns one argument: a vector-valued function f to be minimized,

```
[f] = fun(x)
```

Alternatively, an expression can be substituted for the function name, with x representing the independent variable. For example,

```
x = fsolve('sin(x*x)',x0)
```

options

A vector of control parameters. Of the 18 elements of options, the input options used by leastsq are: 1, 2, 3, 5, 7, 9, 14, 16, 17. The use of options(5) and options(7) by fsolve is discussed in the "Algorithm" section below. When options is an output parameter, the options used by fsolve to return values are: 8, 10, 11, 18.

- options(1) controls display. Setting this to a value of 1 produces a tabular display of intermediate results.
- options(2) controls the accuracy of x at the solution.
- options(3) controls the accuracy of f at the solution.

The termination criteria involving options(2) and options(3) must both hold true for the algorithm to terminate.

For more information on the options vector, including default settings, see the foptions reference page and the "Default Parameters Settings" section in the Tutorial.

grad — A string containing the name of the function that computes the gradient of the objective functions at the point x. This function has the form

```
df = grad(x)
```

The variable df is a matrix that contains the partial derivatives of F with respect to x. The i th column of df corresponds to the partial derivative of the i th function in f with respect to x. (This is the transpose of the Jacobian matrix of F(x).)

x0, — See fminu.

p1, p2, . . .

Examples

Example 1: Find a zero of the system of two equations and two unknowns

$$2x_1 - x_2 = e^{-x_1}$$

$$-x_1 + 2x_2 = e^{-x_2}$$

Thus we want to solve the following system for x

$$2x_1 - x_2 - e^{-x_1} = 0$$

$$-x_1 + 2x_2 - e^{-x_2} = 0$$

starting at x0 = [–5 –5].

Step 1: Write an M-file:

```
function F = fun(x)
F = [2*x(1) – x(2) – exp(–x(1));
      –x(1) + 2*x(2) – exp(–x(2))];
```

Step 2: Invoke an optimization routine:

```
x0 = –5*ones(2, 1);   %Make a starting guess at the solution
options=foptions;     %Set default options
options(1)=1;         %Set option to display output
x = fsolve('fun', x0, options)          % Invoke optimizer
f = fun(x)
```

After 25 function evaluations, a zero is found:

```
 f-COUNT          RESID      STEP-SIZE          GRAD/SD
       3        47071.2              1        -9.41e+04
       8        966.828              1        -1.81e+03
      15        1.99465           3.85             5.6
      20   0.000632051          0.895          -0.0867
      25   1.39647e-15          0.998        -1.89e-09
Optimization Terminated Successfully
x =
    0.5671
    0.5671
f =
   1.0e-07 *
    0.2642
    0.2642
```

Example 2: Find a matrix X that satisfies the equation

$$X * X * X = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix},$$

starting at the point X= [1, 1; 1, 1].

Step 1: Write an M-file:

```
function F = fun(x)
F = x*x*x-[1,2;3,4];
```

Step 2: Invoke an optimization routine:

```
x0 = ones(2,2);   %Make a starting guess at the solution
x = fsolve('fun',x0)          %Invoke optimizer
```

After 44 function evaluations, the solution is

```
x =
     -0.1291      0.8602
      1.2903      1.1612
F = x.*x.*x-[1,2;3,4]
F =
      1.0e-05 *
      0.0350      0.1268
      0.0721     -0.1293
sum(sum(F.*F))
ans =
      3.9218e-12
```

Notes

If the system of equations is linear, then \ (the backslash operator: see `help slash`) should be used for better speed and accuracy. For example, say we want to find the solution to the following linear system of equations:

$$3x_1 + 11x_2 - 2x_3 = 7$$
$$x_1 + x_2 - 2x_3 = 4$$
$$x_1 - x_2 + x_3 = 19$$

Then the problem is formulated and solved as

```
A = [ 3 11 -2;  1  1 -2;  1 -1  1];
b = [ 7;  4;  19];
x  = A\b
x =
    13.2188
    -2.3438
     3.4375
```

Algorithm

The method is based on the nonlinear least-squares algorithm also used in `leastsq`. The advantage of using a least-squares method is that if the system of equations is never zero due to small inaccuracies, or because it just does not have a zero, the algorithm still returns a point where the residual is small. However, if the Jacobian of the system is singular, the algorithm may converge to a point that is not a solution of the system of equations (see "Limitations" and "Diagnostics" below).

The choice of algorithm is made by setting `options(5)`. The default algorithm is the Gauss-Newton method [4]. Setting `options(5) = 1` implements the Levenberg-Marquardt method [1–3].

The default line search algorithm, `options(7) = 0`, is a safeguarded mixed quadratic and cubic polynomial interpolation and extrapolation method. A safeguarded cubic polynomial method can be selected by setting `options(7) = 1`. This method generally requires fewer function evaluations but more gradient evaluations. Thus, if gradients are being supplied and can be calculated inexpensively, the cubic polynomial line search method is preferable.

Limitations

The function to be solved must be continuous. When successful, `fsolve` only gives one root. `fsolve` may converge to a nonzero point, in which case, try other starting values.

Diagnostics

```
Optimizer is stuck at a minimum that is not a root
Try again with a new starting guess
```

See Also

`\`, `foptions`, `leastsq`

References

[1] K. Levenberg, “A Method for the Solution of Certain Problems in Least Squares,” Quart. Appl. Math. 2, pp. 164-168, 1944.

[2] D. Marquardt, “An Algorithm for Least–squares Estimation of Nonlinear Parameters,” SIAM J. Appl. Math., Vol. 11, pp. 431-441, 1963.

[3] J.J. More, “The Levenberg–Marquardt Algorithm: Implementation and Theory,” Numerical Analysis, ed. G. A. Watson, Lecture Notes in Mathematics 630, Springer Verlag, pp. 105-116, 1977.

[4] J.E. Dennis, Jr., “Nonlinear Least Squares,” State of the Art in Numerical Analysis, ed. D. Jacobs, Academic Press, pp. 269-312,

Purpose

Zero of a function of one variable

Syntax

```
z = fzero('fun',x0)
z = fzero('fun',x0,tol)
z = fzero('fun',x0,tol,trace)
z = fzero('fun',x0,tol,trace,P1,P2,...)
```

Description

`fzero('fun',x)` finds a zero of `fun`. `fun` is a string containing the name of a real-valued function of a single real variable. The value returned is near a point where `fun` changes sign, or `NaN` if the search fails.

`fzero('fun',x)` where `x` is a vector of length 2, assumes `x` is an interval where the sign of `f(x(1))` differs from the sign of `f(x(2))`. An error occurs if this is not true. Calling `fzero` with an interval guarantees `fzero` will return a value near a point where `fun` changes sign.

`fzero('fun',x)` where `x` is a scalar value, uses `x` as a starting point. `fzero` looks for an interval containing a sign change for `fun` and containing `x`. If no such interval is found, `NaN` is returned. In this case, the search terminates when the search interval is expanded until an `Inf`, `NaN`, or complex value is found.

`fzero('fun',x,tol)` returns an answer accurate to within a relative error of `tol`.

`z = fzero('fun',x,tol,trace)` displays information at each iteration.

`z = fzero('fun',x,tol,trace,P1,P2,...)` provides for additional arguments passed to the function `fun(x,P1,P2,...)`. Pass an empty matrix for `tol` or `trace` to use the default value, for example: `fzero('fun',x,[],[],P1)`

For the purposes of this command, zeros are considered to be points where the function actually crosses, not just touches, the x-axis.

Arguments

`fun`	A string containing the name of a file in which an arbitrary function of one variable is defined.
`x0`	Your initial estimate of the x-coordinate of a zero of the function or an interval in which you think a zero is found.
`tol`	The relative error tolerance. By default, tol is `eps`.

trace	A nonzero value that causes the fzero command to display information at each iteration of its calculations.
P1, P2, . . .	Additional arguments passed to the function

Examples

Calculate π by finding the zero of the sine function near 3.

```
x = fzero('sin', 3)
x =
    3.1416
```

To find the zero of cosine between 1 and 2:

```
x = fzero('cos', [1 2])
x =
    1.5708
```

Note that cos(1) and cos(2) differ in sign.

To find a zero of the function:

$$f(x) = x^3 - 2x - 5$$

write an M-file called f.m

```
function y = f(x)
y = x.^3-2*x-5;
```

To find the zero near 2

```
z = fzero('f', 2)
z =
    2.0946
```

Since this function is a polynomial, the statement roots([1 0 -2 -5]) finds the same real zero, and a complex conjugate pair of zeros.

```
    2.0946
   -1.0473 + 1.1359i
   -1.0473 - 1.1359i
```

Algorithm

The fzero command is an M-file. The algorithm, which was originated by T. Dekker, uses a combination of bisection, secant, and inverse quadratic inter-

polation methods. An Algol 60 version, with some improvements, is given in [1]. A Fortran version, upon which the `fzero` M-file is based, is in [2].

Limitations

The `fzero` command defines a zero as a point where the function crosses the x-axis. Points where the function touches, but does not cross, the x-axis are not valid zeros. For example, `y = x.^2` is a parabola that touches the x-axis at (0,0). Since the function never crosses the x-axis, however, no zero is found. For functions with no valid zeros, `fzero` executes until `Inf`, `NaN`, or a complex value is detected.

See Also

`eps`, `fmin`, `fsolve`, `\`

References

[1] Brent, R., Algorithms for Minimization Without Derivatives, Prentice-Hall, 1973.

[2] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, Computer Methods for Mathematical Computations, Prentice-Hall, 1976.

leastsq

Purpose

Solve nonlinear least-squares (nonlinear data-fitting) problems,

$$\min_x f(x) = f_1(x)^2 + f_2(x)^2 + f_3(x)^2 + \ldots + f_m(x)^2 + L$$

where L is a constant.

Synopsis

```
x = leastsq('fun',x0)
x = leastsq('fun',x0,options
x = leastsq('fun',x0,options,'grad')
x = leastsq('fun',x0,options,'grad',p1,p2,...)
[x,options] = leastsq('fun',x0,...)
[x,options,funval] = leastsq('fun',x0,...)
[x,options,funval,jacob] = leastsq('fun',x0,...)
```

Description

`leastsq` solves nonlinear least-squares problems, including nonlinear data-fitting problems.

Rather than compute the value f(x), `leastsq` requires the user-defined function to compute the vector-valued function

$$F(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ f_3(x) \end{bmatrix}$$

In vector terms this optimization problem may be restated as

$$\min_x \frac{1}{2}\|F(x)\|_2^2 = \frac{1}{2}\sum_i f_i(x)^2$$

where x is a vector and F(x) is a function that returns a vector value.

`x = leastsq('fun',x0)` starts at `x0` and finds the least-squares minimum of the functions described in the M-file `fun`.

`x = leastsq('fun',x0,options)` uses the parameter values in the vector options rather than the default option values.

`x = leastsq('fun', x0, options, 'grad')` calls the function `grad` to obtain the partial derivatives of the functions.

`x = leastsq('fun', x0, options, 'grad', p1, p2, ...)` passes parameters (i.e., p1, p2, etc.), directly to the function `fun`.

`[x, options] = leastsq('fun', x0)` returns the parameters used in the optimization. `options(10)` contains the number of function evaluations used.

`[x, options, funval] = leastsq('fun', x0)` returns the function value `fun(x)` at the solution `x`.

`[x, options, funval, jacob] = leastsq('fun', x0)` also returns the approximation to the Jacobian of the function at the solution `x`.

Arguments

`fun` A string containing the name of the function that computes the objective function to be minimized at the point `x`. The function `fun` returns one argument: a vector-valued function `f` to be minimized,

```
[f] = fun(x)
```

NOTE The sum of squares should not be formed explicitly. Instead your function should return a vector of function values. See the example below.

Alternatively, an expression can be substituted for the function name, with `x` representing the independent variables. For example,

```
x = leastsq('sin(x)', x0)
```

options

- A vector of control parameters. Of the 18 elements of options, the input options used by leastsq are: 1, 2, 3, 5, 7, 9, 14, 16, 17. When options is an output parameter, the options used by leastsq to return values are: 8, 10, 11, 18.
- options(1) controls display. Setting this to a value of 1 produces a tabular display of intermediate results.
- options(2) controls the accuracy of x at the solution.
- options(3) controls the accuracy of f at the solution.

The termination criteria involving options(2) and options(3) must both hold true for the algorithm to terminate.

The use of options(5) and options(7) by leastsq is discussed in the "Algorithm" section below.

For more information on the options vector, including default settings, see the foptions reference page and the "Default Parameters Settings" section in the Tutorial.

grad

A string containing the name of the function that computes the gradient of the objective functions at the point x. This function has the form

```
df = grad(x)
```

The variable df is a matrix that contains the partial derivatives of F with respect to x. The i th column of df corresponds to the partial derivative of the i th function in f with respect to x. (This is the transpose of the Jacobian matrix of F(x).)

x0,

p1, p2, . . .

See fminu.

funval

The value of the function at the solution x.

jacob

The Jacobian of the function at the solution x.

Examples

Find x that minimizes

$$\sum_{k=1}^{10} (2 + 2k - e^{kx_1} - e^{kx_2})^2$$

starting at the point x= [0.3, 0.4].

Because least sq assumes that the sum-of-squares is not explicitly formed, the function passed to least sq should compute the vector valued function

$$F_k(x) = 2 + 2k - e^{kx_1} - e^{kx_2}$$

for k = 1 to 10.

Step 1: Write an M-file:

```
function f = fun(x)
k = [1:10];
f = 2 + 2*k-exp(k*x(1))-exp(k*x(2));
```

Step 2: Invoke an optimization routine:

```
x0 = [0.3 0.4]          % Starting guess
x = leastsq('fun',x0)   % Invoke optimizer
```

After 41 function evaluations, this example gives the solution:

```
x =
    0.25783   0.25783
sum(fun(x).*fun(x))%esidual or sum of squares
ans =
    124.3622
```

Notes

For the best accuracy and performance, the sum-of-squares should not be formed explicitly. Instead your function should return a vector of function values. See the example above.

Algorithm

The choice of algorithm is made by setting options(5). The default is the Levenberg-Marquardt method [1–3]. Setting options(5) = 1 implements a Gauss-Newton method [4], which is generally faster when the residual $\|F(x)\|_2^2$ is small.

The default line search algorithm, `options(7) = 0`, is a safeguarded mixed quadratic and cubic polynomial interpolation and extrapolation method. A safeguarded cubic polynomial method can be selected by setting `options(7) = 1`. This method generally requires fewer function evaluations but more gradient evaluations. Thus, if gradients are being supplied and can be calculated inexpensively, the cubic polynomial line search method is preferable. The algorithms used are described fully in the Introduction to Algorithms chapter.

Limitations

The function to be minimized must be continuous. `leastsq` may only give local solutions.

`leastsq` only handles real variables. When x has complex variables, the variables must be split into real and imaginary parts.

See Also

`foptions`

References

[1] K. Levenberg, "A Method for the Solution of Certain Problems in Least Squares," Quart. Appl. Math. 2, pp. 164–168, 1944.

[2] D. Marquardt, "An Algorithm for Least–squares Estimation of Nonlinear Parameters," SIAM J. Appl. Math. Vol 11, pp. 431–441, 1963.

[3] J.J. More, "The Levenberg–Marquardt Algorithm: Implementation and Theory," Numerical Analysis, ed. G. A. Watson, Lecture Notes in Mathematics 630, Springer-Verlag, pp. 105–116, 1977.

[4] J.E. Dennis, Jr., "Nonlinear Least Squares," State of the Art in Numerical Analysis, ed. D. Jacobs, Academic Press, pp. 269-312, 1977.

Purpose

Solve a linear programming problem,

$$\min_{x} c^{T}x \quad \text{such that} \quad Ax \leq b$$

where c and b are vectors and A is a matrix. Both equality and inequality constraints can be defined using A and b.

Synopsis

```
x = lp(c, A, b)
x = lp(c, A, b, vlb)
x = lp(c, A, b, vlb, vub)
x = lp(c, A, b, vlb, vub, x0)
x = lp(c, A, b, vlb, vub, x0, neqcstr)
x = lp(c, A, b, vlb, vub, x0, neqcstr, display)
[x, lambda, how] = lp(c, A, b, ...)
```

Description

lp solves linear programming problems.

x = lp(c, A, b) returns a vector x that minimizes the equation c' *x subject to A*x <= b.

x = lp(c, A, b, vlb, vub) sets lower and upper bounds on x. This restricts the solution to the range vlb <= x <= vub.

x = lp(c, A, b, vlb, vub, x0) sets the initial starting point to x0.

x = lp(c, A, b, vlb, vub, x0, neqcstr) specifies that the first neqcstr constraints are equality constraints.

x = lp(c, A, b, vlb, vub, x0, neqcstr, display) controls the display of warning messages.

[x, lambda] = lp(c, A, b) returns the vector lambda of the Lagrange multipliers at the solution x.

[x, lambda, how] = lp(c, A, b) also returns a string how that indicates error conditions at the final iteration.

Arguments

c	The vector c is the set of coefficients of the linear objective function.

A, b	The matrix A and vector b are the coefficients of the linear constraints. The coefficients for the equality constraints must be partitioned into the first rows of A and the first elements of b, followed by the coefficients for the inequality constraints.
vlb, vub	Upper and lower bound vectors. The variables, vlb and vub, are normally the same size as x. However, if vlb has n elements and less elements than x then only the first n elements in x are lower bounded; upper bounds in vub are defined in the same manner.
x0	Starting vector. lp generally starts its search at the point zeros(size(x)). Setting the initial starting point may result in faster convergence. If the problem is badly conditioned, this may also result in an improved solution.
neqcstr	Number of equality constraints.
display	Flag to control the display of warning messages. The default value for the parameter display is 0, which displays warning messages. A value of –1 suppresses warning messages.
lambda	A vector that returns the set of Lagrange multipliers at the solution. The length of lambda is length(b)+length(vlb)+length(vub) and the Lagrange multipliers are given in the corresponding order: first the multipliers for A, then vlb, then vub.
how	A string that indicates error conditions at the solution. The string how='infeasible' indicates that the problem is infeasible (i.e., the constraints are overly restrictive); how='unbounded' indicates that the problem has an unbounded solution; how='dependent' indicates that dependent equality constraints were detected and removed; how='ok' indicates that the problem was solved without difficulty.

As with all Optimization Toolbox functions, empty matrices in the calling sequence result in the use of default variables. For example, the command

```
lp(f,A,b,[],[],[],length(b))
```

indicates that the problem is equality constrained, has no upper or lower bounds on the variables, and uses the default starting point.

Examples

Find x that minimizes $f(x) = -5x_1 - 4x_2 - 6x_3$

subject to

$$x_1 - x_2 + x_3 \le 20$$
$$3x_1 + 2x_2 + 4x_3 \le 42$$
$$3x_1 + 2x_2 \le 30$$
$$0 \le x_1, 0 \le x_2, 0 \le x_3$$

Step 1: Enter the coefficients:

```
c = [ –5, –4, –6]
a =  [ 1 –1   1
       3   2   4
       3   2   0] ;
b = [ 20;  42;  30] ;
```

Step 2: Invoke a linear programming routine:

```
[ x, lambda] = lp( c, a, b, zeros( 3, 1) )
```

This gives the solution

```
x =
      0  15. 0000  3. 000
lambda =
      0
      1. 5000
      0. 5000
      1. 0000
      0
      0
```

The first three elements of the Lagrange multipliers, lambda, are associated with the inequality constraints. Nonzero elements of lambda indicate active constraints at the solution. In this case, constraints two and three are active constraints (i.e., the solution is on their constraint boundaries).

The last three elements of the Lagrange multipliers are associated with the lower bounds on x. Thus, the lower bound on x_1 is also active.

Algorithm

`lp` uses a projection method as used in the `qp` algorithm. `lp` is an active set method and is thus a variation of the well-known simplex method for linear programming [1]. It finds an initial feasible solution by first solving another linear programming problem. `lp` calls `qp` with special flags in order to use efficiencies for the `lp` case.

Diagnostics

`lp` gives a warning when the solution is infeasible,

```
Warning: The constraints are overly stringent;
there is no feasible solution.
```

In this case, `lp` produces a result that minimizes the worst case constraint violation.

When the equality constraints are inconsistent, `lp` gives

```
Warning: The equality constraints are overly
stringent; there is no feasible solution.
```

Unbounded solutions result in the warning

```
Warning: The solution is unbounded and at infinity;
the constraints are not restrictive enough
```

In this case, `lp` returns a value of x that satisfies the constraints.

See Also

`qp`

References

[1] G.B. Dantzig, A. Orden, and P. Wolfe, “Generalized Simplex Method for Minimizing a Linear from Under Linear Inequality Constraints,” Pacific J. Math. Vol. 5, pp. 183–195.

Purpose

Solve the minimax problem,

$$\min_{x} \max_{\{F_i\}} \{F_i(x)\} \quad \text{such that} \quad G(x) \le 0$$

where x is a vector and F(x) and G(x) are functions that return vector values. $F_i(x)$ is the value of the i th element of the vector returned by F(x). G(x) can be used to define equality or inequality constraints.

Synopsis

```
x = minimax('fun',x0)
x = minimax('fun',x0,options)
x = minimax('fun',x0,options,vlb,vub,'grad')
x = minimax('fun',x0,options,vlb,vub,'grad',p1,p2,...)
[x,options] = minimax('fun',x0,...)
```

Description

`minimax` minimizes the worst-case value of a set of multivariable functions, starting at an initial estimate. The values may be subject to constraints. This is generally referred to as the minimax problem.

`x = minimax('fun',x0)` starts at `x0` and finds the minimax solution to the functions described in the M-file `fun.m`.

`x = minimax('fun',x0,options)` defines a vector of optional parameters.

`x = minimax('fun',x,options,vlb,vub)` defines a set of lower and upper bounds on `x` through the vectors `vlb` and `vub`. This restricts the solution to the range `vlb <= x <= vub`.

`x = minimax('fun',x0,options,vlb,vub,'grad')` calls the function `grad` to obtain the partial derivatives of the function and the constraints,

`x = minimax('fun',x0,options,vlb,vub,'grad',p1,p2,...)` passes parameters (i.e., `p1`, `p2`, etc.), directly to the function `fun`.

`[x,options] = minimax('fun',x0)` returns the parameters used in the optimization. For example, `options(10)` contains the number of function evaluations used.

Arguments

fun

A string containing the name of the function that computes the objective function to be minimized and the constraint function at the point x. The function fun returns two arguments: a scalar valued function f to be minimized and a vector of constraint values g,

```
[f,g] = fun(x)
```

When inequality constraints are present, the objective function f is minimized such that g <= zeros(size(g)).

Equality constraints, when present, are placed in the first elements of g. When using equality constraints, options(13) must be set to the number of equality constraints (see the "Equality Constrained Example" section in the Tutorial).

To minimize the worst case absolute values of any of the elements of the vector F(x) (i.e., minimax abs{F(x)}), partition those objectives into the first elements of F(x) and set options(15) to be the number of such objectives.

Alternatively, a string expression can be used with x representing the independent variables and with f and g representing the function and constraints. For example,

```
x = minimax('f = fun(x); g = cstr(x);',x0)
```

When there are no constraints, set g to the empty matrix (i.e., g = [ ]).

grad

A string containing the name of the function that computes the gradient of the function and the gradient of the constraints at the point x. This function has the form

```
[df,dg] = grad(x)
```

The variable df is a vector that contains the partial derivatives of f with respect to x. The variable dg is a matrix where the columns of dg contain the partial derivatives for each of the constraints respectively, (i.e., the i th column of dg corresponds to the partial derivative of the i th constraint with respect to each of the elements in x).

options — A vector of control parameters. Of the 18 elements of options, the input options used by minimax are: 1, 2, 3, 4, 7, 9, 13, 14, 15, 16, 17. When options is an output parameter, the options used by minimax to return values are: 8, 10, 11, 18.

- options(1) controls display. Setting this to a value of 1 produces a tabular display of intermediate results.
- options(2) controls the accuracy of x at the solution.
- options(3) controls the accuracy of f at the solution.
- options(4) sets the maximum constraint violation that is acceptable.

The termination criteria involving options(2), options(3), and options(4) must all hold true for the algorithm to terminate.

The use of options(15) by minimax is discussed under the description of fun above. The use of options(7) and options(8) by minimax is discussed in the "Algorithm" section below. For more information on the options vector, including default settings, see the foptions reference page and the "Default Parameters Settings" section in the Tutorial.

x0, — See constr.

p1, p2, . . . ,

vlb, vub

Examples

Find values of x that minimize the maximum value of

$$\left[f_1(x), f_2(x), f_3(x), f_4(x), f_5(x) \right]$$

where

$$f_1(x) = 2x_1^2 + x_2^2 - 48x_1 - 40x_2 + 304$$

$$f_2(x) = -x_2^2 - 3x_2^2$$

$$f_3(x) = x_1 + 3x_2 - 18$$

$$f_4(x) = -x_1 - x_2$$

$$f_5(x) = x_1 + x_2 - 8.$$

Step 1: Write an M-file:

```
function [f, g] = fun(x)
f(1)=2*x(1)^2+x(2)^2-48*x(1)-40*x(2)+304;       %Objectives
f(2)= x(1)^2 - 3*x(2);
f(3)= x(1) + 3*x(2) -18;
f(4)= -x(1)- x(2);
f(5) = x(1) + x(2) - 8;
g = [ ];                    % No constraints
```

Step 2: Invoke an optimization routine:

```
x0 = [0.1, 0.1];          % Make a starting guess at solution
x = minimax('fun', x0)
```

After 29 function evaluations, the solution is

```
x =
    4.0000  4.0000
fun(x)
ans =
    0.0000  -16.0000  -2.0000  -8.0000  0.000
```

Notes

The number of objectives for which the worst case absolute values of f are minimized is set in options(15). Such objectives should be partitioned into the first elements of f.

For example, consider the above problem, which requires finding values of x that minimize the maximum absolute value of

$$\left[f_1(x),\ f_2(x),\ f_3(x),\ f_4(x),\ f_5(x) \right]$$

This is solved by invoking minimax with the commands

```
x0 = [0.1, 0.1];    % Make a starting guess at the solution
options(15) = 5;    % Minimize absolute values
x = minimax('fun', x0, options)
```

After 39 function evaluations, the solution is

```
x =
      8.7769      0.6613
fun(x)
ans =
    10.7609   -10.7609    -7.2391    -9.4382     1.4382
```

If equality constraints are present and dependent equalities are detected and removed in the quadratic subproblem, 'dependent' will be printed under the Procedures heading (when output is asked for using options(1)=1). The dependent equalities are only removed when the equalities are consistent. If the system of equalities is not consistent, the subproblem is infeasible and 'infeasible' will be printed under the Procedures heading.

Algorithm

minimax uses a Sequential Quadratic Programming (SQP) method [3]. Modifications are made to the line search and Hessian. In the line search an exact merit function (see [4] and [5]) is used together with the merit function proposed by [2]. The line search is terminated when either merit function shows improvement. A modified Hessian that takes advantage of special structure of this problem is also used. Setting options(7) = 1 uses the merit function and Hessian used in constr.

See also SQP implementation section in the Introduction to Algorithms chapter for more details on the algorithm used and the display of procedures for options(1) = 1 setting.

Limitations

The function to be minimized must be continuous. minimax may only give local solutions.

See Also

foptions

References

[1] S.P. Han, "A Globally Convergent Method For Nonlinear Programming," J. of Optimization Theory and Applications, Vol. 22, 1977, p. 297.

[2] M.J.D. Powell, “A Fast Algorithm for Nonlineary Constrained Optimization Calculations,” Numerical Analysis, ed. G.A. Watson, Lecture Notes in Mathematics, Springer Varleg, Vol. 630, 1978.

[3] R.K. Brayton, S.W. Director, G.D. Hachtel, and L.Vidigal, “A New Algorithm for Statistical Circuit Design Based on Quasi–Newton Methods and Function Splitting,” IEEE Trans. Circuits and Systems, Vol. CAS-26, pp. 784-794, Sept. 1979.

[4] A.C.W. Grace, “Computer–Aided Control System Design Using Optimization Techniques”, Ph.D. Thesis, University of Wales, Bangor, Gwynedd, UK, 1989.

[5] K. Madsen and H. Schjaer-Jacobsen, “Algorithms for Worst Case Tolerance Optimization,” IEEE Transactions of Circuits and Systems, Vol. CAS-26, Sept. 1979.

Purpose

Solves the nonnegative least squares problem,

$$\min_{x} \frac{1}{2}\|Ax - b\|_2^2 \qquad \text{such that} \qquad x \geq 0$$

where the matrix A and the vector b are the coefficients of the objective function. The vector, x, of independent variables is restricted to be nonnegative.

Synopsis

```
nnls(A, b)
nnls(A, b, tol)
[x, w] = nnls(A, b)
[x, w] = nnls(A, b, tol)
```

Description

`x = nnls(A, b)` solves the non-negative least-squares problem.

`x = nnls(A, b, tol)` overrides the default tolerance that determines when elements in the vector x are less than zero. The default tolerance is

```
tol = max(size(A)) * norm(A, 1) * eps
```

`[x, w] = nnls(A, b)` returns the dual vector w of Lagrange multipliers. The elements of x and w are related by

```
w(i) <= 0, (i | x(i) = 0)
w(i)  = 0, (i | x(i) > 0)
```

Examples

Compare the unconstrained least squares solution to the nnls solution for a 4-by-2 problem.

```
a =
    0.0372    0.2869
    0.6861    0.7071
    0.6233    0.6245
    0.6344    0.6170

b =
    0.8587
    0.1781
    0.0747
    0.8405
[a\b,   nnls(a,b)] = -2.5625    0
                      3.1106    0.6929
[norm(a*(a\b)-b),   norm(a*nnls(a,b)-b)] = 0.6677  0.9119
```

The solution from nnls does not fit as well as the least squares solution. However, the nonnegative least-squares solution has no negative components.

Algorithm

nnls uses the algorithm described in [1]. The algorithm starts with a set of possible basis vectors and computes the associated dual vector w. It then selects the basis vector corresponding to the maximum value in w in order to swap out of the basis in exchange for another possible candidate. This continues until w <= 0.

Notes

The nonnegative least squares problem is a subset of the constrained linear least-squares problem.

Thus, when A has more rows than columns (i.e., the system is over-determined)

```
[x,w] = nnls(A,b)
```

is equivalent to

```
[m,n] = size(A);
[x,l] = conls(A,b,-eye(n,n),zeros(n,1));
w = -l;
```

For problems greater than order twenty, conls may be faster than nnls, otherwise nnls is generally more efficient.

See Also \, conls

References [1] C.L. Lawson and R.J. Hanson, Solving Least Squares Problems, Prentice-Hall, 1974, Chapter 23, p. 161.

qp

Purpose

Solve the quadratic programming problem,

$$\min_{x} \frac{1}{2} x^T H x + c^T x \quad \text{such that} \quad Ax \leq b$$

where H and A are matrices, and c, b, and x are vectors.

Synopsis

```
x = qp(H, c, A, b)
x = qp(H, c, A, b, vlb)
x = qp(H, c, A, b, vlb, vub)
x = qp(H, c, A, b, vlb, vub, x0)
x = qp(H, c, A, b, vlb, vub, x0, neqcstr)
x = qp(H, c, A, b, vlb, vub, x0, neqcstr, display)
[x, lambda, how] = qp(H, c, A, b, ...)
```

Description

qp solves the quadratic programming problem.

x = qp(H, c, A, b) returns a vector x that minimizes 1/2*x' *H*x + c' *x subject to A*x <= b.

x = qp(H, c, A, b, vlb, vub) sets lower and upper bounds on x. This restricts the solution to the range vlb <= x <= vub.

x = qp(H, c, A, b, vlb, vub, x0) sets the initial starting point to x0.

x = qp(H, c, A, b, vlb, vub, x0, neqcstr) specifies that the first neqcstr constraints are equality constraints.

x = qp(H, c, A, b, vlb, vub, x0, neqcstr, display) controls the display of warning messages.

[x, lambda] = qp(H, c, A, b) returns values for the Lagrange multipliers at the solution in the variable lambda.

[x, lambda, how] = qp(H, c, A, b) also returns a string how that indicates error conditions at the final iteration.

Arguments

H, c — The Hessian matrix H and vector c are the set of coefficients of the quadratic objective function. H must be symmetric.

A, b	The matrix A and vector b are the coefficients of the linear constraints. The coefficients for the equality constraints must be partitioned into the first rows of A and the first elements of b.
vlb, vub	Upper and lower bound vectors. The variables, vlb and vub, are normally the same size as x. However, if vlb has n elements and fewer elements than x, then only the first n elements in x are bounded below; upper bounds in vub are defined in the same manner.
x0	Starting vector. qp generally starts its search at the point zeros(size(x)). Setting the initial starting point can result in faster convergence. If the problem is badly conditioned, this can also result in an improved solution.
neqcstr	Number of equality constraints.
display	Flag to control the display of warning messages. The default value for the parameter display is 0, which displays warning messages. A value of –1 suppresses warning messages.
lambda	A vector that returns the set of Lagrange multipliers at the solution. The length of lambda is length(b)+length(vlb)+length(vub) and the Lagrange multipliers are given in the corresponding order: first the multipliers for A, then vlb, then vub.
how	A string that indicates error conditions at the solution. The string how='infeasible' indicates that the problem is infeasible (i.e. the constraints are overly restrictive); how='unbounded' indicates that the problem has an unbounded solution; how = 'dependent' indicates that dependent equality constraints were detected and removed; how='ok' indicates that the problem was solved without difficulty.

As with all Optimization Toolbox functions, empty matrices in the calling sequence result in the use of default options. For example, the command

```
qp(H, c, A, b, [ ], [ ], [ ], length(b))
```

indicates that the problem is an equality constrained problem, having no upper or lower bounds on the variables and uses a default starting point.

Examples

Find values of x that minimize

$$f(x) = \frac{1}{2}x_1^2 + x_2^2 - x_1x_2 - 2x_1 - 6x_2$$

subject to

$$\begin{aligned} x_1 + x_2 &\le 2 \\ -x_1 + 2x_2 &\le 2 \\ 2x_1 + x_2 &\le 3 \\ 0 \le x_1, &\quad 0 \le x_2 \end{aligned}$$

First we note that this function may be written in matrix notation as $f(x) = \frac{1}{2}x^T H x + c^T x$ where

$$H = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix}, \qquad c = \begin{bmatrix} -2 \\ -6 \end{bmatrix}, \qquad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Step 1: Enter the coefficient matrices:

```
H = [1 -1; -1 2]
c = [-2; -6]
A = [1 1; -1 2; 2 1]
b = [2; 2; 3]
vlb = zeros(2, 1)
```

Step 2: Invoke a quadratic programming routine:

```
[x, lambda] = qp(H, c, A, b, vlb)
```

This generates the solution

```
x =
        0.6667
        1.3333

lambda =
        3.1111
        0.4444
        0
        0
        0
```

The first three elements of the Lagrange multipliers (i.e., `lambda`) are associated with the inequality constraints. Nonzero elements of `lambda` indicate active constraints at the solution. In this case, constraints one and two are active constraints (i.e., the solution is on their constraint boundaries).

The last two elements of the Lagrange multipliers are associated with the lower bounds on `x`. In this case, the bounds are inactive.

Algorithm

`qp` uses an active set method, which is also a projection method, similar to that described in [1]. It finds an initial feasible solution by first solving a linear programming problem. This method is discussed in the Introduction to Algorithms chapter.

Diagnostics

`qp` gives a warning when the solution is infeasible:

```
Warning: The constraints are overly stringent;
         there is no feasible solution.
```

In this case, `qp` produces a result that minimizes the worst case constraint violation.

When the equality constraints are inconsistent, `qp` gives

```
Warning: The equality constraints are overly stringent;
         there is no feasible solution.
```

Unbounded solutions,which can occur when the Hessian H is negative semidefinite, may result in

```
Warning: The solution is unbounded and at infinity;
         the constraints are not restrictive enough.
```

In this case, qp returns a value of x that satisfies the constraints.

Limitations

The solution to indefinite or negative definite problems is often unbounded, and when a finite solution does exist, qp may only give local minima since the problem may be nonconvex.

References

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

Purpose

Find minimum of a semi-infinitely constrained multivariable nonlinear function,

$$\min_{x} f(x) \quad \text{subject to} \quad \begin{aligned} &G(x) \le 0, \\ &K_1(x, w_1) \le 0, \\ &K_2(x, w_2) \le 0, \\ &\cdots \\ &K_n(x, w_n) \le 0 \end{aligned}$$

where x and G(x) are vectors and f(x) is a scalar function. G(x) can be used to define both equality and inequality constraints. The vectors (or matrices) $K_n(x, w_n) \le 0$ are continuous functions of both x and an additional set of variables $w_1, w_2, \ldots, w_n$. The variables $w_1, w_2, \ldots, w_n$ are vectors of, at most, length two.

Synopsis

```
x = seminf('fun', n, x0)
x = seminf('fun', n, x0, options)
x = seminf('fun', n, x0, options, vlb, vub)
x = seminf('fun', n, x0, options, vlb, vub, p1, p2, ...)
[x, options] = seminf('fun', n, x0, ...)
```

Description

`seminf` finds the minimum of a semi-infinitely constrained scalar function of several variables, starting at an initial estimate. The aim is to minimize f(x) so the constraints hold for all possible values of $w_i \in \Re^1$ (or $w_i \in \Re^2$). Since it is impossible to calculate all possible values of $K_i(x, w_i)$, a region must be chosen for w_i over which to calculate an appropriately sampled set of values.

`x = seminf('fun', n, x0)` starts at the point `x0` and finds a minimum of the function and constraints, including `n` semi-infinite constraints, defined in the M-file named `fun.m`

`x = seminf('fun', n, x0, options)` uses the parameter values in the vector `options` rather than the default option values.

`x = seminf('fun', n, x, options, vlb, vub)` defines a set of lower and upper bounds on `x` through the matrices `vlb` and `vub`. This restricts the solution to the range `vlb <= x <= vub`.

`x = seminf('fun', x0, options, vlb, vub, p1, p2, ...)` passes the problem-dependent parameters `p1`, `p2`, etc., directly to the function `fun`.

`[x, options] = seminf('fun', n, x0)` returns the parameters used in the optimization method. For example, `options(10)` contains the number of function evaluations used.

Arguments

`fun` A string containing the name of the function that computes the objective function to be minimized and the constraint function at the point `x`. The function `fun`

```
[f, g, K1, K2, ..., Kn, s] = fun(x, s)
```

returns a scalar value, `f`, of the function to be minimized, and a vector of constraints, `g`. The vectors, or matrices, `K1`, `K2`, ..., `Kn` contain the semi-infinite constraints evaluated for a sampled set of values for the independent variables, `w1`, `w2`, ... `w3`, respectively. The two column matrix, `s`, contains a recommended sampling interval for values of `w1`, `w2`, .. `wn` which are used to evaluate `K1`, `K2`, .. `Kn`. The `i` th row of `s` contains the recommended sampling interval for evaluating `Ki`. When `Ki` is a vector, use only `s(i,1)`. When `Ki` is a matrix, `s(i,2)` is used for the sampling of the rows in `Ki`, `s(i,1)` is used for the sampling interval of the columns of `Ki` (see "Two-Dimensional Example" in the "Examples" section.). On the first iteration `s` is set to `NaN`, so that some initial sampling interval must be determined.

Equality constraints, when present, are placed in the first elements of `g`. When using equality constraints, `options(13)` must be set to the number of equality constraints (see the "Equality Constrained Example" section in the Tutorial).

`n` The number of semi-infinite constraints.

options — A vector of control parameters. Of the 18 elements of options, the input options used by seminf are: 1, 2, 3, 4, 9, 13, 14, 16, 17. When options is an output parameter, the options used by seminf to return values are: 8, 10, 11, 18.

- options(1) controls display. Setting this to a value of 1 produces a tabular display of intermediate results.
- options(2) controls the accuracy of x at the solution.
- options(3) controls the accuracy of f at the solution.
- options(4) sets the maximum constraint violation that is acceptable.

For more information on the options vector, including default settings, see the foptions reference page and the "Default Parameters Settings" section in the Tutorial.

p1, p2, ... — Additional arguments to be passed to fun, that is, when seminf calls fun, the calls are

```
[f, g, K1, K2, ..., Kn, s] = fun(x, s, p1, p2, ...)
```

Using this feature, the same M-file can solve a number of similar problems with different parameters avoiding the need to use global variables. Note that since all the arguments preceding p1, p2, etc., in the call to seminf must be defined, empty matrices may be passed in for options, vlb, and vub to indicate that default arguments are to be used, as in

```
x = seminf('fun', n, x0, [], [], [], p1, p2, ...)
```

x0, vlb, vub — See constr.

Notes

The recommended sampling interval, s, set in fun may be varied by the optimization routine during the computation. Other values may be more appropriate for efficiency or robustness. Also, the finite region w_i, over which $K_i(x, w_i)$ is calculated, is allowed to vary during the optimization provided that it does not result in significant changes in the number of local minima in $K_i(x, w_i)$.

Examples

One-Dimensional Example

Find values of x that minimize

$$f(x) = (x_1 - 0.5)^2 + (x_2 - 0.5)^2 + (x_3 - 0.5)^2$$

where

$$K_1(x, w_1) = \sin(w_1 x_1)\cos(w_1 x_2) - \frac{1}{1000}(w_1 - 50)^2 - \sin(w_1 x_3) - x_3 \le 1$$

$$K_2(x, w_2) = \sin(w_2 x_2)\cos(w_2 x_1) - \frac{1}{1000}(w_2 - 50)^2 - \sin(w_2 x_3) - x_3 \le 1$$

for all values of w_1 and w_2 over the ranges

$$1 \le w_1 \le 100$$

$$1 \le w_2 \le 100$$

Note that the semi-infinite constraints are one-dimensional, that is, vectors.

Step 1: Write an M-file:

```
function [f, G, K1, K2, s] = fun(X, s)
                % Initial sampling interval
if isnan(s(1,1)), s = [0.2  0;  0.2  0]; end
                % Sample set
w1 = 1:s(1,1):100;
w2 = 1:s(2,1):100;
                % Semi-infinite constraints
K1 = sin(w1*X(1)).*cos(w1*X(2)) - 1/1000*(w1-50).^2 -...
     sin(w1* X(3))-X(3)-1;
K2 = sin(w2*X(2)).*cos(w2*X(1)) - 1/1000*(w2-50).^2 -...
     sin(w2*X(3))-X(3)-1;
                % No constraints
G = [];
                % Objective function
f = sum((X-0.5).^2);
                % Plot a graph of semi-infinite constraints
plot(w1, K1, w2, K2), title('Semi-infinite constraints')
```

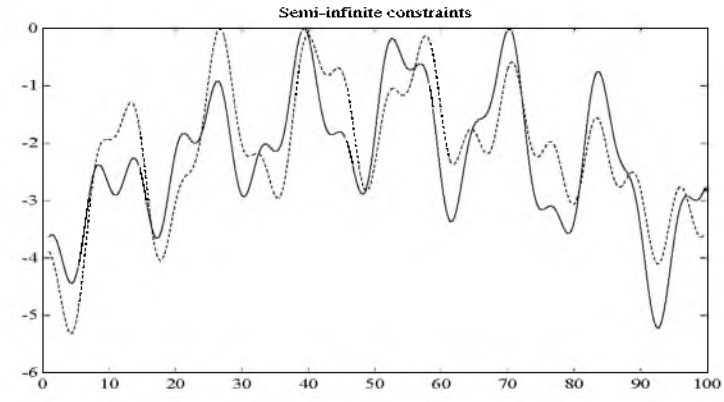
Step 2: Invoke an optimization routine:

```
x0 = [0.5, 0.2, 0.3];          % Starting guess at the solution
x = seminf('fun', 2, x0)
```

After 37 function evaluations, the solution is

```
x =
    0.6956    0.3052    0.4261
[f, G, K1, K2] = fun(x, NaN);
f =
    0.0817
max(K1)
ans =
    -1.0617e-04
max(K2)
ans =
    -0.0023
```

A plot of the semi-infinite constraints is produced.



This plot shows how peaks in both functions are on the constraint boundary.

Two-Dimensional Example
Find values of x that minimize

$$f(x) = (x_1 - 0.5)^2 + (x_2 - 0.5)^2 + (x_3 - 0.5)^2$$

where

$$K_1(x,w) = \sin(w_1 x_1)\cos(10 w_2 x_2) - \frac{1}{1000}(w_1 - 50)^2 - \sin(10 w_1 x_3) - x_3 +$$

$$\sin(w_2 x_2)\cos(w_1 x_1) - \frac{1}{1000}(w_2 - 50)^2 - \sin(w_2 x_3) + -x_3 \le 1.5$$

for all values of w_1 and w_2 over the ranges:

$$1 \le w_1 \le 100$$

$$1 \le w_2 \le 100$$

starting at the point $x = [0.2, 0.2, 0.2]$.

Note that the semi-infinite constraint is two-dimensional, that is, a matrix.

Step 1: Write an M-file:

```
function [f, G, K1, s] = fun(X, s)
                        % Initial sampling intervals
if isnan(s(1,1)), s = [2  2]; end
                        % Sampling sets
w1 = 1:s(1,1):100;
w2 = 1:s(1,2):100;
[wx, wy] = meshdom(w1, w2);
                        % Semi-infinite constraint
K1 = sin(wx*X(1)).*cos(wy*X(2))-1/1000*(wx-50).^2-...
     sin(wx*X(3))-X(3)+sin(wy*X(2)).*cos(wx*X(1))-...
     1/1000*(wy-50).^2-sin(wy*X(3))-X(3)-1.5;
                % No constraints
G = [];
                        % Objective function
f = sum((X-0.2).^2);
                        % Mesh plot
mesh(K1), title('Semi-infinite constraint')
```

Step 2: Invoke an optimization routine:

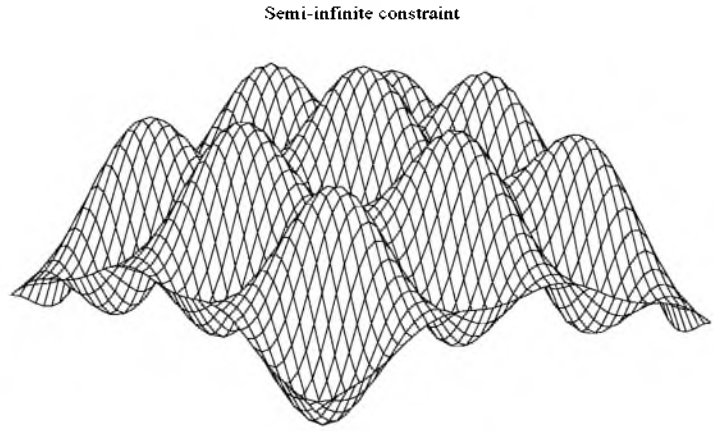
```
x0 = [0.2, 0.2, 0.2];      % Starting guess at the solution
x = seminf('fun', 1, x0)
```

After 65 function evaluations, the solution is

```
x =
    0.2033    0.2034    0.1930
[f, G, K1] = fun(x, NaN);
max(max(K1))
ans =
   -0.0273
```

Note, due to sampling there appears to be no active constraint (i.e., no point on the constraint boundary). Taking a smaller sampling interval shows that the constraint is active.

The following mesh plot is produced.



Algorithm

`seminf` uses cubic and quadratic interpolation techniques to estimate peak values in the semi-infinite constraints. The peak values are used to form a set of constraints that are supplied to the function `constr`. When the number of

constraints changes, Lagrange multipliers are reallocated to the new set of constraints.

The recommended sampling interval calculation uses the difference between the interpolated peak values and peak values appearing in the data set to estimate whether more or fewer points need to be taken. The effectiveness of the interpolation is also taken into consideration by extrapolating the curve and comparing it to other points in the curve. The recommended sampling interval is decreased when the peak values are close to constraint boundaries, i.e., zero.

See also SQP implementation section in the Introduction to Algorithms chapter for more details on the algorithm used and the display of procedures for `options(1) = 1` setting.

Limitations

The function to be minimized, the constraints, and semi-infinite constraints, must be continuous functions of `x` and `w`. `seminf` may only give local solutions.

When the problem is not feasible, `seminf` attempts to minimize the maximum constraint value.

See Also

`constr`, `foptions`

Index

W

Z