fmincon

Find a minimum of a constrained nonlinear multivariable function

$$\min_x f(x) \text{ subject to}$$

$$c(x) \le 0$$
$$ceq(x) = 0$$
$$A \cdot x \le b$$
$$Aeq \cdot x = beq$$
$$lb \le x \le ub$$

where *x, b, beq, lb,* and *ub* are vectors, *A* and *Aeq* are matrices, *c(x)* and *ceq(x)* are functions that return vectors, and *f(x)* is a function that returns a scalar. *f(x)*, *c(x)*, and *ceq(x)* can be nonlinear functions.

Syntax

```
x = fmincon(fun,x0,A,b)
x = fmincon(fun,x0,A,b,Aeq,beq)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
[x,fval] = fmincon(...)
[x,fval,exitflag] = fmincon(...)
[x,fval,exitflag,output] = fmincon(...)
[x,fval,exitflag,output,lambda] = fmincon(...)
[x,fval,exitflag,output,lambda,grad] = fmincon(...)
[x,fval,exitflag,output,lambda,grad,hessian] = fmincon(...)
```

Description

`fmincon` attempts to find a constrained minimum of a scalar function of several variables starting at an initial estimate. This is generally referred to as *constrained nonlinear optimization* or *nonlinear programming*

`x = fmincon(fun,x0,A,b)` starts at `x0` and attempts to find a minimum `x` to the function described in `fun` subject to the linear inequalities `A*x <= b`. `x0` can be a scalar, vector, or matrix.

`x = fmincon(fun,x0,A,b,Aeq,beq)` minimizes `fun` subject to the linear equalities `Aeq*x = beq` as well as `A*x <= b`. Set `A=[]` and `b=[]` if no inequalities exist.

`x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)` defines a set of lower and upper bounds on the design variables in `x`, so that the solution is always in the range `lb <= x <= ub`. Set `Aeq=[]` and `beq=[]` if no equalities exist.

`x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)` subjects the minimization to the nonlinear inequalities `c(x)` or equalities `ceq(x)` defined in `nonlcon`. `fmincon` optimizes such that `c(x) <= 0` and `ceq(x) = 0`. Set `lb=[]` and/or `ub=[]` if no bounds exist.

`x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)` minimizes with the optimization options specified in the structure `options`. Use `optimset` to set these options. Set `nonlcon = []` if there are no nonlinear inequality or equality constraints.

`[x,fval] = fmincon(...)` returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fmincon(...)` returns a value `exitflag` that describes the exit condition of `fmincon`.

`[x,fval,exitflag,output] = fmincon(...)` returns a structure `output` with information about the optimization.

`[x,fval,exitflag,output,lambda] = fmincon(...)` returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

`[x,fval,exitflag,output,lambda,grad] = fmincon(...)` returns the value of the gradient of `fun` at the solution `x`.

`[x,fval,exitflag,output,lambda,grad,hessian] = fmincon(...)` returns the value of the Hessian at the solution `x`. See Hessian

Avoiding Global Variables via Anonymous and Nested Functions explains how to parameterize the objective function `fun`, if necessary.

Input Arguments

Function Arguments contains general descriptions of arguments passed in to `fmincon`. This "Arguments" section provides function-specific details for `fun`, `nonlcon`, and `options`:

fun — The function to be minimized. fun is a function that accepts a vector x and returns a scalar f, the objective function evaluated at x. The function fun can be specified as a function handle for an M-file function

```
x = fmincon(@myfun,x0,A,b)
```

where myfun is a MATLAB function such as

```
function f = myfun(x)
f = ...                  % Compute function value at x
```

fun can also be a function handle for an anonymous function.

```
x = fmincon(@(x)norm(x)^2,x0,A,b);
```

If the gradient of fun can also be computed *and* the GradObj option is 'on', as set by

```
options = optimset('GradObj','on')
```

then the function fun must return, in the second output argument, the gradient value g, a vector, at x. Note that by checking the value of nargout the function can avoid computing g when fun is called with only one output argument (in the case where the optimization algorithm only needs the value of f but not g).

```
function [f,g] = myfun(x)
f = ...            % Compute the function value at x
if nargout > 1     % fun called with two output arguments
   g = ...         % Compute the gradient evaluated at x
end
```

The gradient consists of the partial derivatives of f at the point x. That is, the ith component of g is the partial derivative of f with respect to the ith component of x.

If the Hessian matrix can also be computed *and* the Hessian option is 'on', i.e., options = optimset('Hessian','on'), then the function fun must return the Hessian value H, a symmetric matrix, at x in a third output argument. Note that by checking the value of nargout you can avoid computing H when fun is called with only one or two output arguments (in the case where the optimization algorithm only needs the values of f and g but not H).

```
function [f,g,H] = myfun(x)
f = ...      % Compute the objective function value at x
if nargout > 1      % fun called with two output arguments
   g = ...   % Gradient of the function evaluated at x
   if nargout > 2
      H = ...   % Hessian evaluated at x
   end
end
```

The Hessian matrix is the second partial derivatives matrix of f at the point x. That is, the (i,j)th component of H is the second partial derivative of f with respect to x_i and x_j, $\partial^2 f/\partial x_i \partial x_j$. The Hessian is by definition a symmetric matrix.

`nonlcon` The function that computes the nonlinear inequality constraints `c(x)<= 0` and the nonlinear equality constraints `ceq(x) = 0`. The function `nonlcon` accepts a vector `x` and returns two vectors `c` and `ceq`. The vector `c` contains the nonlinear inequalities evaluated at `x`, and `ceq` contains the nonlinear equalities evaluated at `x`. The function `nonlcon` can be specified as a function handle.

```
x = fmincon(@myfun,x0,A,b,Aeq,beq,lb,ub,@mycon)
```

where `mycon` is a MATLAB function such as

```
function [c,ceq] = mycon(x)
c = ...        % Compute nonlinear inequalities at x.
ceq = ...      % Compute nonlinear equalities at x.
```

If the gradients of the constraints can also be computed *and* the `GradConstr` option is `'on'`, as set by

```
options = optimset('GradConstr','on')
```

then the function `nonlcon` must also return, in the third and fourth output arguments, `GC`, the gradient of `c(x)`, and `GCeq`, the gradient of `ceq(x)`. Note that by checking the value of `nargout` the function can avoid computing `GC` and `GCeq` when `nonlcon` is called with only two output arguments (in the case where the optimization algorithm only needs the values of `c` and `ceq` but not `GC` and `GCeq`).

Avoiding Global Variables via Anonymous and Nested Functions explains how to parameterize the nonlinear constraint function `nonlcon`, if necessary
.

```
function [c,ceq,GC,GCeq] = mycon(x)
c = ...              % Nonlinear inequalities at x
ceq = ...            % Nonlinear equalities at x
if nargout > 2        % nonlcon called with 4 outputs
   GC = ...          % Gradients of the inequalities
   GCeq = ...        % Gradients of the equalities
end
```

If `nonlcon` returns a vector `c` of `m` components and `x` has length `n`, where `n` is the length of `x0`, then the gradient `GC` of `c(x)` is an `n`-by-`m` matrix, where `GC(i,j)` is the partial derivative of `c(j)` with respect to `x(i)` (i.e., the `j`th column of `GC` is the gradient of the `j`th inequality constraint `c(j)`). Likewise, if `ceq` has `p` components, the gradient `GCeq` of `ceq(x)` is an `n`-by-`p` matrix, where `GCeq(i,j)` is the partial derivative of `ceq(j)` with respect to `x(i)` (i.e., the `j`th column of `GCeq` is the gradient of the `j`th equality constraint `ceq(j)`).

`options` Options provides the function-specific details for the `options` values.

Output Arguments

Function Arguments contains general descriptions of arguments returned by `fmincon`. This section provides function-specific details for `exitflag`, `lambda`, and `output`:

`exitflag` Integer identifying the reason the algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

1	First order optimality conditions were satisfied to the specified tolerance.

	2	Change in `x` was less than the specified tolerance.
	3	Change in the objective function value was less than the specified tolerance.
	4	Magnitude of the search direction was less than the specified tolerance and constraint violation was less than `options.TolCon`.
	5	Magnitude of directional derivative was less than the specified tolerance and constraint violation was less than `options.TolCon`.
	0	Number of iterations exceeded `options.MaxIter` or number of function evaluations exceeded `options.FunEvals`
	-1	Algorithm was terminated by the output function.
	-2	No feasible point was found.
`grad`	Gradient at `x`	
`hessian`	Hessian at `x`	
`lambda`	Structure containing the Lagrange multipliers at the solution `x` (separated by constraint type). The fields of the structure are	
	`lower`	Lower bounds `lb`
	`upper`	Upper bounds `ub`
	`ineqlin`	Linear inequalities
	`eqlin`	Linear equalities
	`ineqnonlin`	Nonlinear inequalities
	`eqnonlin`	Nonlinear equalities
`output`	Structure containing information about the optimization. The fields of the structure are	
	`iterations`	Number of iterations taken
	`funcCount`	Number of function evaluations
	`algorithm`	Algorithm used.
	`cgiterations`	Number of PCG iterations (large-scale algorithm only)
	`stepsize`	Final step size taken (medium-scale algorithm only)
	`firstorderopt`	Measure of first-order optimality For large-scale bound constrained problems, the first-order optimality is the infinity norm of `v.*g`, where `v` is defined as in Box Constraints, and `g` is the gradient. For large-scale problems with only linear equalities, the first-order optimality is the infinity norm of the *projected* gradient (i.e. the gradient projected onto the nullspace of `Aeq`).

Hessian

`fmincon` computes the output argument `hessian` as follows:

- When using the medium-scale algorithm, the function computes a quasi-Newton approximation to the Hessian of the Lagrangian at `x`.
- When using the large-scale algorithm, the function uses
  - `options.Hessian`, if you supply it, to compute the Hessian at `x`
  - A finite-difference approximation to the Hessian at `x`, if you supply only the gradient. Note that because the large-scale algorithm does not take nonlinear constraints, the Hessian of the Lagrangian is the same as the Hessian of the objective function.

Options

Optimization options used by `fmincon`. Some options apply to all algorithms, some are only relevant when you are using the large-scale algorithm, and others are only relevant when you are using the medium-scale algorithm.You can use `optimset` to set or change the values of these fields in the options structure `options`. See Optimization Options, for detailed information.

The `LargeScale` option specifies a preference for which algorithm to use. It is only a preference because certain conditions must be met to use the large-scale algorithm. For `fmincon`, you must provide the gradient (see the preceding description of `fun` to see how) or else the medium-scale algorithm is used:

`LargeScale`	Use the large-scale algorithm if possible when set to `'on'`. Use the medium-scale algorithm when set to `'off'`.

Medium-Scale and Large-Scale Algorithms. These options are used by both the medium-scale and large-scale algorithms:

`DerivativeCheck`	Compare user-supplied derivatives (gradients of the objective and constraints) to finite-differencing derivatives.
`Diagnostics`	Display diagnostic information about the function to be minimized.
`DiffMaxChange`	Maximum change in variables for finite differencing
`DiffMinChange`	Minimum change in variables for finite differencing
`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` (default) displays just the final output.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays a warning when the objective function returns a value that is `complex`, `Inf`, or `NaN`. `'off'` displays no warning.
`GradObj`	Gradient for the objective function defined by the user. See the preceding description of `fun` to see how to define the gradient in `fun`. You must provide the gradient to use the large-scale method. It is optional for the medium-scale method.
`MaxFunEvals`	Maximum number of function evaluations allowed

MaxIter	Maximum number of iterations allowed
OutputFcn	Specify a user-defined function that an opimization function calls at each iteration. See Output Function.
TolFun	Termination tolerance on the function value.
TolCon	Termination tolerance on the constraint violation.
TolX	Termination tolerance on x.
TypicalX	Typical x values.

Large-Scale Algorithm Only. These options are used only by the large-scale algorithm:

Hessian	If 'on', fmincon uses a user-defined Hessian (defined in fun), or Hessian information (when using HessMult), for the objective function. If 'off', fmincon approximates the Hessian using finite differences.

HessMult	Function handle for Hessian multiply function. For large-scale structured problems, this function computes the Hessian matrix product H*Y without actually forming H. The function is of the form
	`W = hmfun(Hinfo,Y,p1,p2,...)`
	where Hinfo and possibly the additional parameters p1,p2,... contain the matrices used to compute H*Y.
	The first argument must be the same as the third argument returned by the objective function fun, for example by
	`[f,g,Hinfo] = fun(x)`
	Y is a matrix that has the same number of rows as there are dimensions in the problem. W = H*Y although H is not formed explicitly. fminunc uses Hinfo to compute the preconditioner. The optional parameters p1, p2, ... can be any additional parameters needed by hmfun. See Avoiding Global Variables via Anonymous and Nested Functions for information on how to supply values for the parameters.
	Note 'Hessian' must be set to 'on' for Hinfo to be passed from fun to hmfun.
	See Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints for an example.

`HessPattern`	Sparsity pattern of the Hessian for finite differencing. If it is not convenient to compute the sparse Hessian matrix `H` in `fun`, the large-scale method in `fmincon` can approximate `H` via sparse finite differences (of the gradient) provided the *sparsity structure* of `H` -- i.e., locations of the nonzeros -- is supplied as the value for `HessPattern`. In the worst case, if the structure is unknown, you can set `HessPattern` to be a dense matrix and a full finite-difference approximation is computed at each iteration (this is the default). This can be very expensive for large problems, so it is usually worth the effort to determine the sparsity structure.
`MaxPCGIter`	Maximum number of PCG (preconditioned conjugate gradient) iterations (see the *Algorithm* section following).
`PrecondBandWidth`	Upper bandwidth of preconditioner for PCG. By default, diagonal preconditioning is used (upper bandwidth of 0). For some problems, increasing the bandwidth reduces the number of PCG iterations.
`TolPCG`	Termination tolerance on the PCG iteration.

Medium-Scale Algorithm Only. These options are used only by the medium-scale algorithm:

`MaxSQPIter`	Maximum number of SQP iterations allowed

Examples

Find values of *x* that minimize $f(x) = -x_1 x_2 x_3$, starting at the point `x = [10; 10; 10]` and subject to the constraints

$$0 \le x_1 + 2x_2 + 2x_3 \le 72$$

First, write an M-file that returns a scalar value `f` of the function evaluated at `x`.

```
function f = myfun(x)
f = -x(1) * x(2) * x(3);
```

Then rewrite the constraints as both less than or equal to a constant,

$$-x_1 - 2x_2 - 2x_3 \le 0$$
$$x_1 + 2x_2 + 2x_3 \le 72$$

Since both constraints are linear, formulate them as the matrix inequality $A \cdot x \le b$ where

$$A = \begin{bmatrix} -1 & -2 & -2 \\ 1 & 2 & 2 \end{bmatrix} \qquad b = \begin{bmatrix} 0 \\ 72 \end{bmatrix}$$

Next, supply a starting point and invoke an optimization routine.

```
x0 = [10;  10;  10];   % Starting guess at the solution
[x,fval] = fmincon(@myfun,x0,A,b)
```

After 66 function evaluations, the solution is

```
x =
    24.0000
    12.0000
    12.0000
```

where the function value is

```
fval =
  -3.4560e+03
```

and linear inequality constraints evaluate to be less than or equal to `0`.

```
A*x-b=
    -72
      0
```

Notes

Large-Scale Optimization. To use the large-scale method, you must

- Supply the gradient in `fun`
- Set `GradObj` to `'on'` in `options`
- Specify the feasible region using one, but not both, of the following types of constraints:
  - Upper and lower bounds constraints
  - Linear equality constraints, in which the equality constraint matrix `Aeq` cannot have more rows than columns. `Aeq` is typically sparse.

You cannot use inequality constraints with the large-scale algorithm. If the preceding conditions are not met, `fmincon` reverts to the medium-scale algorithm.

The function `fmincon` returns a warning if no gradient is provided and the `LargeScale` option is not `'off'`. `fmincon` permits *g*(*x*) to be an approximate gradient but this option is not recommended; the numerical behavior of most optimization methods is considerably more robust when the true gradient is used. See Table 2-4, Large-Scale Problem Coverage and Requirements, for more information on what problem formulations are covered and what information you must be provide.

The large-scale method in `fmincon` is most effective when the matrix of second derivatives , i.e., the Hessian matrix *H*(*x*), is also computed. However, evaluation of the true Hessian matrix is not required. For example, if you can supply the Hessian sparsity structure (using the `HessPattern` option in `options`), `fmincon` computes a sparse finite-difference approximation to *H*(*x*).

If `x0` is not strictly feasible, `fmincon` chooses a new strictly feasible (centered) starting point.

If components of *x* have no upper (or lower) bounds, then `fmincon` prefers that the corresponding components of `ub` (or `lb`) be set to `Inf` (or `-Inf` for `lb`) as opposed to an arbitrary but very large positive (or negative in the case of lower bounds) number.

Several aspects of linearly constrained minimization should be noted:

- A dense (or fairly dense) column of matrix `Aeq` can result in considerable fill and computational cost.
- `fmincon` removes (numerically) linearly dependent rows in `Aeq`; however, this process involves repeated matrix factorizations and therefore can be costly if there are many dependencies.
- Each iteration involves a sparse least-squares solution with matrix

$$\overline{Aeq} = Aeq^T R^{-T}$$

where *RT* is the Cholesky factor of the preconditioner. Therefore, there is a potential conflict between choosing an effective preconditioner and minimizing fill in $\overline{Aeq}$.

Medium-Scale Optimization. Better numerical results are likely if you specify equalities explicitly, using `Aeq` and `beq`, instead of implicitly, using `lb` and `ub`.

If equality constraints are present and dependent equalities are detected and removed in the quadratic subproblem, `'dependent'` is displayed under the `Procedures` heading (when you ask for output by setting the `Display` option to `'iter'`). 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'` is displayed under the `Procedures` heading.

Algorithm

Large-Scale Optimization. The large-scale algorithm is a subspace trust region method and is based on the interior-reflective Newton method described in [1], [2]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See the trust region and preconditioned conjugate gradient method descriptions in the Large-Scale Algorithms chapter.

Medium-Scale Optimization. `fmincon` uses a sequential quadratic programming (SQP) method. In this method, the function solves a quadratic programming (QP) subproblem at each iteration. An estimate of the Hessian of the Lagrangian is updated at each iteration using the BFGS formula (see `fminunc`, references [7], [8]).

A line search is performed using a merit function similar to that proposed by [4], [5], and [6]. The QP subproblem is solved using an active set strategy similar to that described in [3]. A full description of this algorithm is found in Constrained Optimization in "Introduction to Algorithms."

See also SQP Implementation in "Introduction to Algorithms" for more details on the algorithm used.

Limitations

`fmincon` only handles real variables.

The function to be minimized and the constraints must both be continuous. `fmincon` might only give local solutions.

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

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

The large-scale method does not allow equal upper and lower bounds. For example if `lb(2)==ub(2)`, then `fmincon` gives the error

```
Equal upper and lower bounds not permitted in this large-scale
method.
Use equality constraints and the medium-scale method instead.
```

If you only have equality constraints you can still use the large-scale method. But if you have both equalities and bounds, you must use the medium-scale method.

See Also

`@` (`function_handle`), `fminbnd`, `fminsearch`, `fminunc`, `optimset`

References

[1] Coleman, T.F. and Y. Li, "An Interior, Trust Region Approach for Nonlinear Minimization Subject to Bounds," *SIAM Journal on Optimization*, Vol. 6, pp. 418-445, 1996.

[2] Coleman, T.F. and Y. Li, "On the Convergence of Reflective Newton Methods for Large-Scale Nonlinear Minimization Subject to Bounds," *Mathematical Programming*, Vol. 67, Number 2, pp. 189-224, 1994.

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

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

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

[6] Powell, M.J.D., "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.