Find minimum of unconstrained multivariable function

Equation

Finds the minimum of a problem specified by

$$\min_x f(x)$$

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

x is a vector or a matrix; see Matrix Arguments.

Syntax

```
x = fminunc(fun,x0)
x = fminunc(fun,x0,options)
x = fminunc(problem)
[x,fval] = fminunc(...)
[x,fval,exitflag] = fminunc(...)
[x,fval,exitflag,output] = fminunc(...)
[x,fval,exitflag,output,grad] = fminunc(...)
[x,fval,exitflag,output,grad,hessian] = fminunc(...)
```

Description

`fminunc` attempts to find a minimum of a scalar function of several variables, starting at an initial estimate. This is generally referred to as *unconstrained nonlinear optimization*.

> **Note:** Passing Extra Parameters explains how to pass extra parameters to the objective function, if necessary.

`x = fminunc(fun,x0)` starts at the point `x0` and attempts to find a local minimum `x` of the function described in `fun`. `x0` can be a scalar, vector, or matrix.

`x = fminunc(fun,x0,options)` minimizes with the optimization options specified in `options`. Use `optimoptions` to set these options.

`x = fminunc(problem)` finds the minimum for `problem`, where `problem` is a structure described in Input Arguments.

Create the `problem` structure by exporting a problem from Optimization app, as described in Exporting Your Work.

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

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

`[x,fval,exitflag,output] = fminunc(...)` returns a structure `output` that contains information about the optimization.

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

`[x,fval,exitflag,output,grad,hessian] = fminunc(...)` returns in `hessian` the value of the Hessian of the objective function `fun` at the solution `x`. See Hessian.

Input Arguments

Function Arguments contains general descriptions of arguments passed into `fminunc`. This section provides function-specific details for `fun`, `options`, and `problem`:

`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 a file
	
	`x = fminunc(@myfun,x0)`
	
	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 = fminunc(@(x)norm(x)^2,x0);`
	
	If the gradient of `fun` can also be computed *and* the `GradObj` option is `'on'`, as set by
	
	`options = optimoptions(@fminunc,'GradObj','on')`
	
	then the function `fun` must return, in the second output argument, the gradient value `g`, a vector, at `x`. The gradient is the partial derivatives *∂f/∂xi* 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 = optimoptions(@fminunc,'GradObj','on','Hessian','on')`, then the function `fun` must return the Hessian value `H`, a symmetric matrix, at `x` in a third output argument. 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*, *∂*^{2}*f/∂xi∂xj*. The Hessian is by definition a symmetric matrix.
	
	Writing Scalar Objective Functions explains how to "conditionalize" the gradients and Hessians for use in solvers that do not accept them. Passing Extra Parameters explains how to parameterize `fun`, if necessary.		

`options`	Options provides the function-specific details for the `options` values.		
`problem`	`objective`	Objective function	
	`x0`	Initial point for `x`	
	`solver`	`'fminunc'`	
	`options`	Options created with `optimoptions`	

Output Arguments

Function Arguments contains general descriptions of arguments returned by `fminunc`. This section provides function-specific details for `exitflag` 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`	Magnitude of gradient smaller than the `TolFun` tolerance.
	`2`	Change in `x` was smaller than the `TolX` tolerance.
	`3`	Change in the objective function value was less than the `TolFun` tolerance.
	`5`	Predicted decrease in the objective function was less than the `TolFun` tolerance.
	`0`	Number of iterations exceeded `MaxIter` or number of function evaluations exceeded `MaxFunEvals`.
	`-1`	Algorithm was terminated by the output function.
	`-3`	Objective function at current iteration went below `ObjectiveLimit`.
`grad`	Gradient at `x`	
`hessian`	Hessian at `x`	
`output`	Structure containing information about the optimization. The fields of the structure are	
	`iterations`	Number of iterations taken

	`funcCount`	Number of function evaluations
	`firstorderopt`	Measure of first-order optimality
	`algorithm`	Optimization algorithm used
	`cgiterations`	Total number of PCG iterations (`'trust-region'` algorithm only)
	`stepsize`	Final displacement in `x` (`'quasi-newton'` algorithm only)
	`message`	Exit message

Hessian

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

- When using the `'quasi-newton'` algorithm, the function computes a finite-difference approximation to the Hessian at `x` using
  - The gradient `grad` if you supply it
  - The objective function `fun` if you do not supply the gradient
- When using the `'trust-region'` 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

Options

`fminunc` uses these optimization options. Some options apply to all algorithms, some are only relevant when you are using the `trust-region` algorithm, and others are only relevant when you are using the `quasi-newton` algorithm. Use `optimoptions` to set or change `options`. See Optimization Options Reference for detailed information.

All Algorithms

All `fminunc` algorithms use the following options:

`Algorithm` If you use `optimset` (not recommended), use `LargeScale` instead of `Algorithm`.	Choose the `fminunc` algorithm. Choices are `'quasi-newton'` and `'trust-region'` (default). The `trust-region` algorithm requires you to provide the gradient (see the preceding description of `fun`), or else `fminunc` uses the `'quasi-newton'` algorithm. For information on choosing the algorithm, see Choosing the Algorithm.
`DerivativeCheck`	Compare user-supplied derivatives (gradient of objective) to finite-differencing derivatives. The choices are `'on'` or the default `'off'`.
`Diagnostics`	Display diagnostic information about the function to be minimized or solved. The choices are `'on'` or the default `'off'`.
`DiffMaxChange`	Maximum change in variables for finite-difference gradients (a positive scalar). The default is `Inf`.
`DiffMinChange`	Minimum change in variables for finite-difference gradients (a positive scalar). The default is `0`.
`Display`	Level of display: • `'off'` or `'none'` displays no output. • `'iter'` displays output at each iteration, and gives the default exit message. • `'iter-detailed'` displays output at each iteration, and gives the technical exit message. • `'notify'` displays output only if the function does not converge, and gives the default exit message. • `'notify-detailed'` displays output only if the function does not converge, and gives the technical exit message.

	• `'final'` (default) displays just the final output, and gives the default exit message. • `'final-detailed'` displays just the final output, and gives the technical exit message.

`FinDiffRelStep`	Scalar or vector step size factor. When you set `FinDiffRelStep` to a vector `v`, forward finite differences `delta` are
	`delta = v.*sign(x).*max(abs(x),TypicalX);`
	and central finite differences are
	`delta = v.*max(abs(x),TypicalX);`
	Scalar `FinDiffRelStep` expands to a vector. The default is `sqrt(eps)` for forward finite differences, and `eps^(1/3)` for central finite differences.
	The trust-region algorithm uses `FinDiffRelStep` only when `DerivativeCheck` is `'on'`.

`FinDiffType`	Finite differences, used to estimate gradients, are either `'forward'` (the default), or `'central'` (centered). `'central'` takes twice as many function evaluations, but should be more accurate. The trust-region algorithm uses `FinDiffType` only when `DerivativeCheck` is `'on'`.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex`, `Inf`, or `NaN`. The default, `'off'`, displays no error.
`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`. Set to `'on'` to have `fminunc` use a user-defined gradient of the objective function. The default `'off'` causes `fminunc` to estimate gradients using finite differences. You must provide the gradient, and set `GradObj` to `'on'`, to use the trust-region algorithm. This option is not required for the quasi-Newton algorithm.
`LargeScale` If you use `optimoptions` (recommended), use `Algorithm` instead of `LargeScale`.	Use `'trust-region'` algorithm if possible when set to the default `'on'`. Use `'quasi-newton'` algorithm when set to `'off'`. The `LargeScale` algorithm requires you to provide the gradient (see the preceding description of `fun`), or else `fminunc` uses the `'quasi-newton'` algorithm. For information on choosing the algorithm, see Choosing the Algorithm.
`MaxFunEvals`	Maximum number of function evaluations allowed, a positive integer. The default value is `100*numberOfVariables`.
`MaxIter`	Maximum number of iterations allowed, a positive integer. The default value is `400`.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[]`). See Output Function.
`PlotFcns`	Plots various measures of progress while the algorithm executes. Select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[]`). • `@optimplotx` plots the current point. • `@optimplotfunccount` plots the function count. • `@optimplotfval` plots the function value. • `@optimplotstepsize` plots the step size. • `@optimplotfirstorderopt` plots the first-order optimality measure. For information on writing a custom plot function, see Plot Functions.
`TolFun`	Termination tolerance on the function value, a positive scalar. The default is `1e-6`.
`TolX`	Termination tolerance on `x`, a positive scalar. The default value is `1e-6`.

`TypicalX`	Typical x values. The number of elements in `TypicalX` is equal to the number of elements in `x0`, the starting point. The default value is `ones(numberofvariables,1)`. `fminunc` uses `TypicalX` for scaling finite differences for gradient estimation. The `trust-region` algorithm uses `TypicalX` only for the `DerivativeCheck` option.

`trust-region` Algorithm Only

The `trust-region` algorithm uses the following options:

`Hessian`	If `'on'`, `fminunc` uses a user-defined Hessian for the objective function. The Hessian is either defined in the objective function (see `fun`), or is indirectly defined when using `HessMult`. If `'off'` (default), `fminunc` 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)`
	
	where `Hinfo` contains the matrix 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. See Passing Extra Parameters for information on how to supply values for any additional parameters `hmfun` needs.
	
	Note `'Hessian'` must be set to `'on'` for `fminunc` to pass `Hinfo` from `fun` to `hmfun`.
	
	See Minimization with Dense Structured Hessian, Linear Equalities for an example.

`HessPattern`	Sparsity pattern of the Hessian for finite differencing. Set `HessPattern(i,j) = 1` when you can have ∂^2`fun`/∂`x(i)`∂`x(j)` ≠ 0. Otherwise, set `HessPattern(i,j) = 0`. Use `HessPattern` when it is inconvenient to compute the Hessian matrix H in `fun`, but you can determine (say, by inspection) when the `i`th component of the gradient of `fun` depends on `x(j)`. `fminunc` can approximate H via sparse finite differences (of the gradient) if you provide the *sparsity structure* of H — i.e., locations of the nonzeros — as the value for `HessPattern`. In the worst case, when the structure is unknown, do not set `HessPattern`. The default behavior is as if `HessPattern` is a dense matrix of ones. Then `fminunc` computes a full finite-difference approximation in each iteration. This can be very expensive for large problems, so it is usually better to determine the sparsity structure.
`MaxPCGIter`	Maximum number of PCG (preconditioned conjugate gradient) iterations, a positive scalar. The default is `max(1,floor(numberOfVariables/2))`. For more information, see Algorithms.
`PrecondBandWidth`	Upper bandwidth of preconditioner for PCG, a nonnegative integer. By default, `fminunc` uses diagonal preconditioning (upper bandwidth of 0). For some problems, increasing the bandwidth reduces the number of PCG iterations. Setting `PrecondBandWidth` to `Inf` uses a direct factorization (Cholesky) rather than the conjugate gradients (CG). The direct factorization is computationally more expensive than CG, but produces a better quality step towards the solution.
`TolPCG`	Termination tolerance on the PCG iteration, a positive scalar. The default is `0.1`.

`quasi-newton` Algorithm Only

The `quasi-newton` algorithm uses the following options:

`HessUpdate`	Method for choosing the search direction in the Quasi-Newton algorithm. The choices are: ▪ `'bfgs'`, the default ▪ `'dfp'` ▪ `'steepdesc'` See Hessian Update for a description of these methods.
`InitialHessMatrix` This option will be removed in a future release.	Initial quasi-Newton matrix. This option is only available if you set `InitialHessType` to `'user-supplied'`. In that case, you can set `InitialHessMatrix` to one of the following: ▪ A positive scalar — The initial matrix is the scalar times the identity. ▪ A vector of positive values— The initial matrix is a diagonal matrix with the entries of the vector on the diagonal. This vector should be the same size as the `x0` vector, the initial point.
`InitialHessType` This option will be removed in a future release.	Initial quasi-Newton matrix type. The options are: ▪ `'identity'` ▪ `'scaled-identity'`, the default ▪ `'user-supplied'` — See `InitialHessMatrix`
`ObjectiveLimit`	A tolerance (stopping criterion) that is a scalar. If the objective function value at an iteration is less than or equal to `ObjectiveLimit`, the iterations halt, since the problem is presumably unbounded. The default value is `-1e20`.

Examples

Minimize the function $f(x) = 3x_1^2 + 2x_1x_2 + x_2^2$.

Create a file `myfun.m`:

```
function f = myfun(x)
f = 3*x(1)^2 + 2*x(1)*x(2) + x(2)^2;    % Cost function
```

Then call `fminunc` to find a minimum of `myfun` near `[1,1]`:

```
x0 = [1,1];
[x,fval] = fminunc(@myfun,x0);
```

After a few iterations, `fminunc` returns the solution, `x`, and the value of the function at `x`, `fval`:

```
x,fval

x =
  1.0e-006 *
    0.2541   -0.2029

fval =
  1.3173e-013
```

To minimize this function with the gradient provided, modify `myfun.m` so the gradient is the second output argument:

```
function [f,g] = myfun(x)
f = 3*x(1)^2 + 2*x(1)*x(2) + x(2)^2;    % Cost function
if nargout > 1
   g(1) = 6*x(1)+2*x(2);
   g(2) = 2*x(1)+2*x(2);
end
```

Indicate that the gradient value is available by creating optimization options with the `GradObj` option set to `'on'` using `optimoptions`. Choose the `'trust-region'` algorithm, which requires a gradient.

```
options = optimoptions('fminunc','GradObj','on','Algorithm','trust-region');
x0 = [1,1];
[x,fval] = fminunc(@myfun,x0,options);
```

After several iterations `fminunc` returns the solution, `x`, and the value of the function at `x`, `fval`:

```
x,fval

x =
  1.0e-015 *
    0.1110   -0.8882

fval =
  6.2862e-031
```

To minimize the function `f(x) = sin(x) + 3` using an anonymous function

```
f = @(x)sin(x)+3;
x = fminunc(f,4);
```

`fminunc` returns a solution

```
x

x =
    4.7124
```

Notes

`fminunc` is not the preferred choice for solving problems that are sums of squares, that is, of the form

$$\min_x \|f(x)\|_2^2 = \min_x \left(f_1(x)^2 + f_2(x)^2 + \ldots + f_n(x)^2 \right)$$

Instead use the `lsqnonlin` function, which has been optimized for problems of this form.

To use the trust-region method, you must provide the gradient in `fun` (and set the `GradObj` option to `'on'` using `optimoptions`). A warning is given if no gradient is provided and the `Algorithm` option is `'trust-region'`.

Limitations

The function to be minimized must be continuous. `fminunc` might only give local solutions.

`fminunc` only minimizes 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.

Trust Region Optimization

To use the `trust-region` algorithm, you must supply the gradient in `fun` (and `GradObj` must be set `'on'` in `options`).

Trust Region Algorithm Coverage and Requirements

Additional Information Needed	For Large Problems
Must provide gradient for `f(x)` in `fun`.	▪ Provide sparsity structure of the Hessian, or compute the Hessian in `fun`. ▪ The Hessian should be sparse.

More About

expand all

Algorithms

- `function_handle`
- Anonymous Functions

- Unconstrained Optimization

References

[1] Broyden, C.G., "The Convergence of a Class of Double-Rank Minimization Algorithms," *Journal Inst. Math. Applic.*, Vol. 6, pp. 76-90, 1970.

[2] 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.

[3] 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.

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

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

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

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

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

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

See Also

`fminsearch` | `optimoptions` | `optimtool`