Provide feedback about this page

fsolve

Solve system of nonlinear equations

Equation

Solves a problem specified by

$$F(x) = 0$$

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

Syntax

```
x = fsolve(fun,x0)
x = fsolve(fun,x0,options)
x = fsolve(problem)
[x,fval] = fsolve(fun,x0)
[x,fval,exitflag] = fsolve(...)
[x,fval,exitflag,output] = fsolve(...)
[x,fval,exitflag,output,jacobian] = fsolve(...)
```

Description

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

`x = fsolve(fun,x0)` starts at `x0` and tries to solve the equations described in `fun`.

`x = fsolve(fun,x0,options)` solves the equations with the optimization options specified in the structure `options`. Use `optimset` to set these options.

`x = fsolve(problem)` solves `problem`, where `problem` is a structure described in Input Arguments.

Create the structure `problem` by exporting a problem from Optimization Tool, as described in Exporting to the MATLAB® Workspace.

`[x,fval] = fsolve(fun,x0)` returns the value of the objective function `fun` at the solution `x`.

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

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

`[x,fval,exitflag,output,jacobian] = fsolve(...)` returns the Jacobian of `fun` at the solution `x`.

Passing Extra Parameters explains how to parameterize the objective function `fun`, if necessary.

Input Arguments

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

`fun` The nonlinear system of equations to solve. `fun` is a function that accepts a vector `x` and returns a vector `F`, the nonlinear equations evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fsolve(@myfun,x0)
```

where `myfun` is a MATLAB® function such as

```
function F = myfun(x)
F = ...            % Compute function values at x
```

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

```
x = fsolve(@(x)sin(x.*x),x0);
```

If the user-defined values for `x` and `F` are matrices, they are converted to a vector using linear indexing.

If the Jacobian can also be computed *and* the Jacobian option is `'on'`, set by

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

then the function `fun` must return, in a second output argument, the Jacobian value `J`, a matrix, at `x`.

If `fun` returns a vector (matrix) of `m` components and `x` has length `n`, where `n` is the length of `x0`, then the Jacobian `J` is an `m`-by-`n` matrix where `J(i,j)` is the partial derivative of `F(i)` with respect to `x(j)`. (Note that the Jacobian `J` is the transpose of the gradient of `F`.)

`problem`

`objective`	Objective function
`x0`	Initial point for `x`
`solver`	`'fsolve'`
`options`	Options structure created with `optimset`

Output Arguments

Function Arguments contains general descriptions of arguments returned by `fsolve`. For more information on the output headings for `fsolve`, see Function-Specific Output Headings.

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	Function converged to a solution `x`.
2	Change in `x` was smaller than the specified tolerance.
3	Change in the residual was smaller than the specified tolerance.

	`4`	Magnitude of search direction was smaller than the specified tolerance.
	`0`	Number of iterations exceeded `options.MaxIter` or number of function evaluations exceeded `options.FunEvals`
	`-1`	Algorithm was terminated by the output function.
	`-2`	Algorithm appears to be converging to a point that is not a root.
	`-3`	Trust radius became too small.
	`-4`	Line search cannot sufficiently decrease the residual along the current search direction.
`output`	Structure containing information about the optimization. The fields of the structure are	
	`iterations`	Number of iterations taken
	`funcCount`	Number of function evaluations
	`algorithm`	Optimization algorithm used.
	`cgiterations`	Total number of PCG iterations (large-scale algorithm only)
	`stepsize`	Final displacement in `x` (Gauss-Newton and Levenberg-Marquardt algorithms)
	`firstorderopt`	Measure of first-order optimality (dogleg or large-scale algorithm, [ ] for others)
	`message`	Exit message

Options

Optimization options used by `fsolve`. Some options apply to all algorithms, some are only relevant when using the large-scale algorithm, and others are only relevant when 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 `fsolve`, the nonlinear system of equations cannot be underdetermined; that is, the number of equations (the number of elements of `F` returned by `fun`) must be at least as many as the length of `x` or else the medium-scale algorithm is used:

`LargeScale`	Use large-scale algorithm if possible when set to `'on'`. Use medium-scale algorithm when set to `'off'`. The default for `fsolve` is `'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 (Jacobian) to finite-differencing derivatives.
`Diagnostics`	Display diagnostic information about the function to be solved.
`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 an error when the objective function returns a value that is `complex`, `Inf`, or `NaN`. `'off'` (the default) displays no error.
`Jacobian`	If `'on'`, `fsolve` uses a user-defined Jacobian (defined in `fun`), or Jacobian information (when using `JacobMult`), for the objective function. If `'off'`, `fsolve` approximates the Jacobian using finite differences.
`MaxFunEvals`	Maximum number of function evaluations allowed.
`MaxIter`	Maximum number of iterations allowed.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration. See Output Function.
`PlotFcns`	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Specifying `@optimplotx`plots the current point; `@optimplotfunccount`plots the function count; `@optimplotfval`plots the function value; `@optimplotresnorm`plots the norm of the residuals; `@optimplotstepsize`plots the step size; `@optimplotfirstorderopt`plots the first-order of optimality.
`TolFun`	Termination tolerance on the function value.
`TolX`	Termination tolerance on `x`.
`TypicalX`	Typical x values.

Large-Scale Algorithm Only

These options are used only by the large-scale algorithm:

`JacobMult`

Function handle for Jacobian multiply function. For large-scale structured problems, this function computes the Jacobian matrix product `J*Y`, `J'*Y`, or `J'*(J*Y)` without actually forming `J`. The function is of the form

```
W = jmfun(Jinfo,Y,flag,p1,p2,...)
```

where `Jinfo` and the additional parameters `p1,p2,...` contain the matrices used to compute `J*Y` (or `J'*Y`, or `J'*(J*Y)`). The first argument `Jinfo` must be the same as the second argument returned by the objective function fun, for example by

```
[F,Jinfo] = fun(x)
```

Y is a matrix that has the same number of rows as there are dimensions in the problem. `flag` determines which product to compute:

- If `flag == 0` then `W = J'*(J*Y)`.
- If `flag > 0` then `W = J*Y`.
- If `flag < 0` then `W = J'*Y`.

In each case, `J` is not formed explicitly. `fsolve` uses `Jinfo` to compute the preconditioner. The optional parameters `p1`, `p2`, `...` can be any additional parameters needed by `jmfun`. See Passing Extra Parameters for information on how to supply values for these parameters.

> **Note** `'Jacobian'` must be set to `'on'` for `Jinfo` to be passed from `fun` to `jmfun`.

See Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints for a similar example.

`JacobPattern`

Sparsity pattern of the Jacobian for finite differencing. If it is not convenient to compute the Jacobian matrix `J` in `fun`, `lsqnonlin` can approximate `J` via sparse finite differences provided the structure of `J`—i.e., locations of the nonzeros—is supplied as the value for `JacobPattern`. In the worst case, if the structure is unknown, you can set `JacobPattern` to be a dense matrix and a full finite-difference approximation is computed in each iteration (this is the default if `JacobPattern` is not set). 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 Algorithm).

`PrecondBandWidth`	The default `PrecondBandWidth`is `'Inf'`, which means a direct factorization (Cholesky) is used rather than the conjugate gradients (CG). The direct factorization is computationally more expensive than CG, but produces a better quality step towards the solution. Set `PrecondBandWidth`to 0 for diagonal preconditioning (upper bandwidth of 0). For some problems, an intermediate 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:

`NonlEqnAlgorithm`	Specify one of the following algorithms for solving nonlinear equations: • `'dogleg'` — Trust-region dogleg algorithm (default) • `'lm'` — Levenberg-Marquardt • `'gn'` — Gauss-Newton
`LineSearchType`	Line search algorithm choice. This option applies to the `'lm'` (Levenberg-Marquardt) and `'gn'` (Gauss-Netwton) algorithms.

Examples

Example 1

This example finds a zero of the system of two equations and two unknowns:

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

You 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]`.

First, write an M-file that computes `F`, the values of the equations at `x`.

```
function F = myfun(x)
F = [2*x(1) - x(2) - exp(-x(1));
      -x(1) + 2*x(2) - exp(-x(2))];
```

Next, call an optimization routine.

```
x0 = [-5; -5];           % Make a starting guess at the solution
options=optimset('Display','iter');   % Option to display output
[x,fval] = fsolve(@myfun,x0,options)  % Call optimizer
```

After 33 function evaluations, a zero is found.

```
                                    Norm of  First-order Trust-region
Iteration Func-count    f(x)          step   optimality       radius
    0        3       23535.6                 2.29e+004        1
    1        6       6001.72           1     5.75e+003        1
    2        9       1573.51           1     1.47e+003        1
    3       12       427.226           1           388        1
    4       15       119.763           1           107        1
    5       18       33.5206           1          30.8        1
    6       21       8.35208           1          9.05        1
    7       24       1.21394           1          2.26        1
    8       27      0.016329    0.759511         0.206      2.5
    9       30  3.51575e-006    0.111927       0.00294      2.5
   10       33  1.64763e-013  0.00169132     6.36e-007      2.5
Optimization terminated successfully:
 First-order optimality is less than options.TolFun

x =
    0.5671
    0.5671

fval =
  1.0e-006 *
      -0.4059
      -0.4059
```

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]`.

First, write an M-file that computes the equations to be solved.

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

Next, invoke an optimization routine.

```
x0 = ones(2,2);  % Make a starting guess at the solution
options = optimset('Display','off');  % Turn off Display
[x,Fval,exitflag] = fsolve(@myfun,x0,options)
```

The solution is

```
x =
    -0.1291    0.8602
     1.2903    1.1612

Fval =
```

```
  1.0e-009 *
    -0.1619    0.0776
     0.1161   -0.0469

exitflag =
     1
```

and the residual is close to zero.

```
sum(sum(Fval.*Fval))
ans =
     4.7915e-020
```

Notes

If the system of equations is linear, use `\` (matrix left division) for better speed and accuracy. For example, to find the solution to the following linear system of equations:

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

You can formulate and solve the problem 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 Gauss-Newton, Levenberg-Marquardt, and large-scale methods are based on the nonlinear least-squares algorithms also used in `lsqnonlin`. Use one of these methods if the system may 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 might converge to a point that is not a solution of the system of equations (see Limitations and Diagnostics following).

Large-Scale Optimization

`fsolve`, with the `LargeScale` option set to `'on'` with `optimset`, uses the large-scale algorithm if possible. This algorithm is a subspace trust-region method and is based on the interior-reflective Newton method described in [1] and [2]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See Trust-Region Methods for Nonlinear Minimization and Preconditioned Conjugate Gradients.

Medium-Scale Optimization

By default `fsolve` chooses the medium-scale algorithm and uses the trust-region dogleg method. The algorithm is a variant of the Powell dogleg method described in [8]. It is similar in nature to the algorithm implemented in [7].

Alternatively, you can select a Gauss-Newton method [3] with line-search, or a

Levenberg-Marquardt method [4], [5], and [6] with line-search. Use `optimset` to set `NonlEqnAlgorithm` option to `'dogleg'` (default), `'lm'`, or `'gn'`.

The default line search algorithm for the Levenberg-Marquardt and Gauss-Newton methods, i.e., the `LineSearchType` option, is `'quadcubic'`. This is a safeguarded mixed quadratic and cubic polynomial interpolation and extrapolation method. A safeguarded cubic polynomial method can be selected by setting `LineSearchType` to `'cubicpoly'`. 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 Standard Algorithms .

Diagnostics

Medium and Large-Scale Optimization

`fsolve` may converge to a nonzero point and give this message:

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

In this case, run `fsolve` again with other starting values.

Medium-Scale Optimization

For the trust-region dogleg method, `fsolve` stops if the step size becomes too small and it can make no more progress. `fsolve` gives this message:

```
The optimization algorithm can make no further progress:
 Trust region radius less than 10*eps
```

In this case, run `fsolve` again with other starting values.

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.

`fsolve` only handles real variables. When x has complex variables, the variables must be split into real and imaginary parts.

Large-Scale Optimization

The preconditioner computation used in the preconditioned conjugate gradient part of the large-scale method forms J^TJ (where J is the Jacobian matrix) before computing the preconditioner; therefore, a row of J with many nonzeros, which results in a nearly dense product J^TJ, might lead to a costly solution process for large problems.

Medium-Scale Optimization

The default trust-region dogleg method can only be used when the system of equations is square, i.e., the number of equations equals the number of unknowns. For the Levenberg-Marquardt and Gauss-Newton methods, the system of equations need not be square.

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

[4] Levenberg, K., "A Method for the Solution of Certain Problems in Least-Squares," *Quarterly Applied Mathematics 2* , pp. 164-168, 1944.

[5] Marquardt, D., "An Algorithm for Least-squares Estimation of Nonlinear Parameters," *SIAM Journal Applied Mathematics* , Vol. 11, pp. 431-441, 1963.

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

[7] Moré, J. J., B. S. Garbow, and K. E. Hillstrom, *User Guide for MINPACK 1*, Argonne National Laboratory, Rept. ANL-80-74, 1980.

[8] Powell, M. J. D., "A Fortran Subroutine for Solving Systems of Nonlinear Algebraic Equations," *Numerical Methods for Nonlinear Algebraic Equations* , P. Rabinowitz, ed., Ch.7, 1970.

See Also

`@ (function_handle)`, `\` (matrix left division), `lsqcurvefit`, `lsqnonlin`, `optimset`, `optimtool`, anonymous functions

Provide feedback about this page

fseminf fzero

fsolve

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.

Syntax

```
x = fsolve(fun,x0)
x = fsolve(fun,x0,options)
[x,fval] = fsolve(...)
[x,fval,exitflag] = fsolve(...)
[x,fval,exitflag,output] = fsolve(...)
[x,fval,exitflag,output,jacobian] = fsolve(...)
```

Description

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

`x = fsolve(fun,x0)` starts at `x0` and tries to solve the equations described in `fun`.

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

`[x,fval] = fsolve(fun,x0)` returns the value of the objective function `fun` at the solution `x`.

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

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

`[x,fval,exitflag,output,jacobian] = fsolve(...)` returns the Jacobian of `fun` at the solution `x`

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 `fsolve`. This section provides function-specific details for `fun` and `options`:

`fun` The nonlinear system of equations to solve. `fun` is a function that accepts a vector `x` and returns a vector `F`, the nonlinear equations evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fsolve(@myfun,x0)
```

where `myfun` is a MATLAB function such as

```
function F = myfun(x)
F = ...            % Compute function values at x
```

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

```
x = fsolve(@(x)sin(x.*x),x0);
```

If the Jacobian can also be computed *and* the `Jacobian` option is `'on'`, set by

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

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

```
function [F,J] = myfun(x)
F = ...          % objective function values at x
if nargout > 1   % two output arguments
   J = ...   % Jacobian of the function evaluated at x
end
```

If `fun` returns a vector (matrix) of `m` components and `x` has length `n`, where `n` is the length of `x0`, then the Jacobian `J` is an m-by-n matrix where `J(i,j)` is the partial derivative of`F(i)` with respect to `x(j)`. (Note that the Jacobian `J` is the transpose of the gradient of `F`.)

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

Output Arguments

Function Arguments contains general descriptions of arguments returned by `fsolve`. 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	Function converged to a solution `x`.
2	Change in `x` was smaller than the specified tolerance.
3	Change in the residual was smaller than the specified tolerance.

	`4`	Magnitude of search direction was smaller than the specified tolerance.
	`0`	Number of iterations exceeded `options.MaxIter` or number of function evaluations exceeded `options.FunEvals`.
	`-1`	Algorithm was terminated by the output function.
	`-2`	Algorithm appears to be converging to a point that is not a root.
	`-3`	Trust radius became too small.
	-4	Line search cannot sufficiently decrease the residual along the current search direction.
`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 (large-scale algorithm only) For large-scale problems, the first-order optimality is the infinity norm of the gradient $g = J^T F$ (see Nonlinear Least-Squares).

Options

Optimization options used by `fsolve`. Some options apply to all algorithms, some are only relevant when using the large-scale algorithm, and others are only relevant when 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 `fsolve`, the nonlinear system of equations cannot be underdetermined; that is, the number of equations (the number of elements of `F` returned by `fun`) must be at least as many as the length of `x` or else the medium-scale algorithm is used:

`LargeScale`	Use large-scale algorithm if possible when set to `'on'`. Use medium-scale algorithm when set to `'off'`. The default for `fsolve` is `'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 (Jacobian) to finite-differencing derivatives.
`Diagnostics`	Display diagnostic information about the function to be minimized.
`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` (default) displays just the final output.
`Jacobian`	If `'on'`, `fsolve` uses a user-defined Jacobian (defined in `fun`), or Jacobian information (when using `JacobMult`), for the objective function. If `'off'`, `fsolve` approximates the Jacobian using finite differences.
`MaxFunEvals`	Maximum number of function evaluations allowed.
`MaxIter`	Maximum number of iterations allowed.
`TolFun`	Termination tolerance on the function value.
`TolX`	Termination tolerance on `x`.
`TypicalX`	Typical `x` values.

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

`JacobMult`	Function handle for Jacobian multiply function. For large-scale structured problems, this function computes the Jacobian matrix product `J*Y`, `J'*Y`, or `J'*(J*Y)` without actually forming `J`. The function is of the form `W = jmfun(Jinfo,Y,flag,p1,p2,...)`

where `Jinfo` and the additional parameters `p1,p2,...` contain the matrices used to compute `J*Y` (or `J'*Y`, or `J'*(J*Y)`). The first argument `Jinfo` must be the same as the second argument returned by the objective function `fun`, for example by

```
[F,Jinfo] = fun(x)
```

`Y` is a matrix that has the same number of rows as there are dimensions in the problem. `flag` determines which product to compute:

- If `flag == 0` then `W = J'*(J*Y)`.
- If `flag > 0` then `W = J*Y`.
- If `flag < 0` then `W = J'*Y`.

In each case, `J` is not formed explicitly. `fsolve` uses `Jinfo` to compute the preconditioner. The optional parameters `p1`, `p2`, ... can be any additional parameters needed by `jmfun`. See Avoiding Global Variables via Anonymous and Nested Functions for information on how to supply values for these parameters.

> **Note** `'Jacobian'` must be set to `'on'` for `Jinfo` to be passed from `fun` to `jmfun`.

See Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints for a similar example.

`JacobPattern` — Sparsity pattern of the Jacobian for finite differencing. If it is not convenient to compute the Jacobian matrix `J` in `fun`, `lsqnonlin` can approximate `J` via sparse finite differences provided the structure of`J` -- i.e., locations of the nonzeros -- is supplied as the value for `JacobPattern`. In the worst case, if the structure is unknown, you can set `JacobPattern` to be a dense matrix and a full finite-difference approximation is computed in each iteration (this is the default if `JacobPattern` is not set). 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 Algorithm).

`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:

`DiffMaxChange`	Maximum change in variables for finite differencing.
`DiffMinChange`	Minimum change in variables for finite differencing.
`NonlEqnAlgorithm`	Choose Levenberg-Marquardt or Gauss-Newton over the trust region dogleg algorithm.
`LineSearchType`	Line search algorithm choice.

Examples

Example 1. This example finds a zero of the system of two equations and two unknowns:

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

You 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]`.

First, write an M-file that computes `F`, the values of the equations at `x`.

```
function F = myfun(x)
F = [2*x(1) - x(2) - exp(-x(1));
      -x(1) + 2*x(2) - exp(-x(2))];
```

Next, call an optimization routine.

```
x0 = [-5; -5];           % Make a starting guess at the solution
options=optimset('Display','iter');   % Option to display output
[x,fval] = fsolve(@myfun,x0,options)  % Call optimizer
```

After 33 function evaluations, a zero is found.

```
                                        Norm of  First-order Trust-region
Iteration Func-count    f(x)              step   optimality       radius
    0        3       23535.6                    2.29e+004        1
    1        6       6001.72           1        5.75e+003        1
    2        9       1573.51           1        1.47e+003        1
    3       12       427.226           1             388        1
    4       15       119.763           1             107        1
    5       18       33.5206           1            30.8        1
    6       21       8.35208           1            9.05        1
    7       24       1.21394           1            2.26        1
    8       27      0.016329    0.759511           0.206      2.5
    9       30  3.51575e-006    0.111927         0.00294      2.5
   10       33  1.64763e-013  0.00169132        6.36e-007      2.5
Optimization terminated successfully:
 First-order optimality is less than options.TolFun

x =
    0.5671
    0.5671

fval =
  1.0e-006 *
      -0.4059
      -0.4059
```

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]`.

First, write an M-file that computes the equations to be solved.

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

Next, invoke an optimization routine.

```
x0 = ones(2,2);  % Make a starting guess at the solution
options = optimset('Display','off');  % Turn off Display
[x,Fval,exitflag] = fsolve(@myfun,x0,options)
```

The solution is

```
x =
    -0.1291    0.8602
     1.2903    1.1612

Fval =
  1.0e-009 *
    -0.1619    0.0776
     0.1161   -0.0469

exitflag =
     1
```

and the residual is close to zero.

```
sum(sum(Fval.*Fval))
ans =
     4.7915e-020
```

Notes

If the system of equations is linear, use the `\` (the backslash operator; see `help slash`) for better speed and accuracy. For example, to find the solution to the following linear system of equations:

$$\begin{aligned} 3x_1 + 11x_2 - 2x_3 &= 7 \\ x_1 + x_2 - 2x_3 &= 4 \\ x_1 - x_2 + x_3 &= 19 \end{aligned}$$

You can formulate and solve the problem 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 Gauss-Newton, Levenberg-Marquardt, and large-scale methods are based on the nonlinear least-squares algorithms also used in `lsqnonlin`. Use one of these methods if the system may 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 might converge to a point that is not a solution of the system of equations (see Limitations and Diagnostics following).

Large-Scale Optimization. `fsolve`, with the `LargeScale` option set to `'on'` with `optimset`, uses the large-scale algorithm if possible. This 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 Trust-Region Methods for Nonlinear Minimization and Preconditioned Conjugate Gradients.

Medium-Scale Optimization. By default `fsolve` chooses the medium-scale algorithm and uses the trust-region dogleg method. The algorithm is a variant of the Powell dogleg method described in [8]. It is similar in nature to the algorithm implemented in [7].

Alternatively, you can select a Gauss-Newton method [3] with line-search, or a Levenberg-Marquardt method [4], [5], [6] with line-search. The choice of algorithm is made by setting the `NonlEqnAlgorithm` option to `'dogleg'` (default), `'lm'`, or `'gn'`.

The default line search algorithm for the Levenberg-Marquardt and Gauss-Newton methods, i.e., the `LineSearchType` option set to `'quadcubic'`, is a safeguarded mixed quadratic and cubic polynomial interpolation and extrapolation method. A safeguarded cubic polynomial method can be selected by setting `LineSearchType` to `'cubicpoly'`. 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 Standard Algorithms chapter.

Diagnostics

Medium and Large-Scale Optimization. `fsolve` may converge to a nonzero point and give this message:

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

In this case, run `fsolve` again with other starting values.

Medium-Scale Optimization. For the trust region dogleg method, `fsolve` stops if the step size becomes too small and it can make no more progress. `fsolve` gives this message:

```
The optimization algorithm can make no further progress:
 Trust region radius less than 10*eps
```

In this case, run `fsolve` again with other starting values.

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.

`fsolve` only handles real variables. When *x* has complex variables, the variables must be split into real and imaginary parts.

Large-Scale Optimization. The preconditioner computation used in the preconditioned conjugate gradient part of the large-scale method forms J^TJ (where J is the Jacobian matrix) before computing the preconditioner; therefore, a row of J with many nonzeros, which results in a nearly dense product J^TJ, might lead to a costly solution process for large problems.

Medium-Scale Optimization. The default trust region dogleg method can only be used when the system of equations is square, i.e., the number of equations equals the number of unknowns. For the Levenberg-Marquardt and Gauss-Newton methods, the system of equations need not be square.

See Also

@ (function_handle), \, lsqcurvefit, lsqnonlin, optimset, anonymous functions

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

[4] Levenberg, K., "A Method for the Solution of Certain Problems in Least-Squares," *Quarterly Applied Mathematics 2*, pp. 164-168, 1944.

[5] Marquardt, D., "An Algorithm for Least-squares Estimation of Nonlinear Parameters," *SIAM Journal Applied Mathematics*, Vol. 11, pp. 431-441, 1963.

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

[7] Moré, J. J., B. S. Garbow, and K. E. Hillstrom, *User Guide for MINPACK 1*, Argonne National Laboratory, Rept. ANL-80-74, 1980.

[8] Powell, M. J. D., "A Fortran Subroutine for Solving Systems of Nonlinear Algebraic Equations," *Numerical Methods for Nonlinear Algebraic Equations*, P. Rabinowitz, ed., Ch.7, 1970.

Linear Programming with Equalities and Inequalities

The problem is

$$\min f^T x \qquad \text{such that} \qquad \begin{matrix} Aeq \cdot x = beq \\ A \cdot x \le b \\ x \ge 0 \end{matrix}$$

and you can load the matrices and vectors `A`, `Aeq`, `b`, `beq`, `f`, and the lower bounds `lb` into the MATLAB workspace with

```
load sc50b
```

This problem in `sc50b.mat` has 48 variables, 30 inequalities, and 20 equalities.

You can use `linprog` to solve the problem:

```
[x,fval,exitflag,output] = ...
    linprog(f,A,b,Aeq,beq,lb,[],[],optimset('Display','iter'));
```

Because the iterative display was set using `optimset`, the results displayed are

```
  Residuals:   Primal      Dual      Duality     Total
               Infeas     Infeas       Gap        Rel
               A*x-b     A'*y+z-f     x'*z       Error
  ------------------------------------------------------
  Iter    0:  1.50e+003  2.19e+001  1.91e+004  1.00e+002
  Iter    1:  1.15e+002  2.94e-015  3.62e+003  9.90e-001
  Iter    2:  1.16e-012  2.21e-015  4.32e+002  9.48e-001
  Iter    3:  3.23e-012  5.16e-015  7.78e+001  6.88e-001
  Iter    4:  5.78e-011  7.61e-016  2.38e+001  2.69e-001
  Iter    5:  9.31e-011  1.84e-015  5.05e+000  6.89e-002
  Iter    6:  2.96e-011  1.62e-016  1.64e-001  2.34e-003
  Iter    7:  1.51e-011  2.74e-016  1.09e-005  1.55e-007
  Iter    8:  1.51e-012  2.37e-016  1.09e-011  1.51e-013

Optimization terminated successfully.
```

For this problem, the large-scale linear programming algorithm quickly reduces the scaled residuals below the default tolerance of `1e-08`.

The `exitflag` value is positive, telling you `linprog` converged. You can also get the final function value in `fval` and the number of iterations in `output.iterations`:

```
exitflag =
     1
fval =
 -70.0000
output =
      iterations: 8
    cgiterations: 0
       algorithm: 'lipsol'
```

Linear Least-Squares with Bound Constraints

Linear Programming with Dense Columns in the Equalities

Nonlinear Inequality Constrained Example

If inequality constraints are added to Eq. 2-1, the resulting problem can be solved by the `fmincon` function. For example, find *x* that solves

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

subject to the constraints

$$x_1x_2 - x_1 - x_2 \le -1.5$$
$$x_1x_2 \ge -10$$

Because neither of the constraints is linear, you cannot pass the constraints to `fmincon` at the command line. Instead you can create a second M-file, `confun.m`, that returns the value at both constraints at the current `x` in a vector `c`. The constrained optimizer, `fmincon`, is then invoked. Because `fmincon` expects the constraints to be written in the form $c(x) \le 0$, you must rewrite your constraints in the form

$$x_1x_2 - x_1 - x_2 + 1.5 \le 0 \qquad \textbf{(2-3)}$$
$$-x_1x_2 - 10 \le 0$$

Step 1: Write an M-file confun.m for the constraints.

```
function [c, ceq] = confun(x)
% Nonlinear inequality constraints
c = [1.5 + x(1)*x(2) - x(1) - x(2);
     -x(1)*x(2) - 10];
% Nonlinear equality constraints
ceq = [];
```

Step 2: Invoke constrained optimization routine.

```
x0 = [-1,1];     % Make a starting guess at the solution
options = optimset('LargeScale','off');
[x, fval] = ...
fmincon(@objfun,x0,[],[],[],[],[],[],@confun,options)
```

After 38 function calls, the solution `x` produced with function value `fval` is

```
x =
   -9.5474  1.0474
fval =
    0.0236
```

You can evaluate the constraints at the solution by entering

```
[c,ceq] = confun(x)
```

This returns

```
c=
    1.0e-14 *
    0.1110
   -0.1776

ceq =
     []
```

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

Unconstrained Minimization Example Constrained Example with Bounds

W.R. Wilcox, Clarkson University
Last revised May 4, 2006

SOLUTION OF EQUATIONS USING MATLAB

See also:

- **Tutorial on symbolic mathematics using MATLAB**
- **Relationships between mathematical, MATLAB and Excel expressions**
- **MATLAB tutorials on analysis and plotting of data**
- **Common conversion factors for units**
- **Introductory MATLAB tutorials: See MATLAB help, MATLAB, getting started. The following sites also have useful tutorials: Mathworks, Florida, Utah, Louisiana, New Hampshire, Carnegie Mellon**

The present tutorial deals exclusively with numerical methods of solving algebraic and trigonometric equations, both single and several simultaneously, both linear and non-linear. Analytical solutions can be obtained using the methods described in the symbolic tutorial. When both symbolic and numerical methods work, sometimes one is simpler and sometimes the other.

In order to benefit from the following material one must copy and paste the indicated commands into MATLAB and execute them. These commands are shown in the format >> help plot, for example. The user should look up each command in MATLAB's help in order to understand the command and the different possible syntaxes that can be used with it.

Graphical solution of one non-linear equation or two non-linear equations in explicit form

When a single equation or a pair of equations is to be solved, it is good practice to first make a plot. In this way, it can be seen immediately what the approximate solution is.

In order to plot an equation using the "plot" command it must be in explicit form, i.e. y = f(x). If the equation can be written only in implicit form, i.e. f(x,y) = 0, then the "ezplot" command must be used as described in the symbolic tutorial.

We consider first a single equation, i.e. f(x) = 0. As an example we will use $\sin^2(x)\, e^{-x/2} - 1 = 0$. Do the following steps:

1. **While optional, it's a good idea to clear the Workspace memory in order to avoid the chance of MATLAB using a previous value for a variable:**
   >> clear
2. **Guess the range of x values over which you expect a solution and generate values for the x vector over this range. Select an increment small enough to yield a smooth curve. For example:**
   >> x = -10:0.01:10;
   gives x from -10 to +10 in increments of 0.01.
3. **Generate the y = f(x) vector corresponding to these x values:**
   >> y = sin(x).^2.*exp(-x/2) -1;
   Notice the dots before ^ and *. These dots are essential. Because x is a vector, we must use array operations rather than scalar operations (without the dot).
4. **Also create a line for y = 0. The intersection of this line with the curve above gives the solution (or solutions, as in this example). To do this we create a vector with the same number of values as x, with each value being 0.**

```
>> y0 = zeros(1,length(x));
```

5. **Make the plot:**

```
>> plot(x,y,x,y0);
```

6. **Locate the solution(s). If the two lines do not intersect, repeat with another range for x. In our example, the lines intersect more than once, and we surmise, in fact, that there are infinitely many solutions for negative x. Let us select the solution at about x = -1. Enlarge this area of the intersection in the Figure window by Tools / Zoom In, and then clicking near the intersection once or more. Then go to Tools / Data Cursor. Click as close as you can to the intersection. Write the result down to compare with that obtained by using the "fzero" command below.**

Next let us consider a pair of explicit equations, $y = \sin^2(x)\, e^{-x/2} - 1$ and $y = 10\sin(x)/x$. Proceeding as above:

```
>> clear;  x = -10:0.01:10;   y1 = 10*sin(x)./x;   y2 = sin(x).^2.*exp(-x/2) - 1;  plot(x,y1,x,y2);
```

Again we see that there are many solutions (intersections), both positive and negative. (Note that when MATLAB calculated sin(0)/0 it gave a warning, but nonetheless completed the other calculations and the plot.) As above, find the approximate solution at x near 3.5. (I get x = 3.49, y = 0.9796.) Write your result down to compare with those to be found later. Alternately, we can equate these two equations (since both = y), move everything to the left-hand side, and find the values of x when this is 0. For the value of x found, the corresponding value for y is determined by substituting this back into either one of the original equations.

```
>> clear;  x = -10:0.01:10;   yeq = 10*sin(x)./x - sin(x).^2.*exp(-x/2)+1;
>> yeq0 = zeros(1,length(x));   plot(x,yeq,x,yeq0);
```

Using the same method as above, I again got x = 3.49 and yeq = 0. Note that yeq is not y. To find the y we substitute x back into both of the simultaneous equations, which gives us two results. (Use your own value of x rather than my value of 3.49.)

```
>> clear;  x = 3.49;  y1 = 10*sin(x)/x,   y2 = sin(x)^2*exp(-x/2) - 1
```

The closer these are y1 and y2, the better our solution.

Numerical solution of one or two non-linear equations in explicit form (y = f(x))

We use the "fzero" command, which finds the value of x for f(x) = 0. We will use the same examples as above. To display the result to 14 significant figures we first change the format to long. For the single equation, $\sin^2(x)\, e^{-x/2} - 1 = 0$:

```
>> clear;  format long;  x = fzero('sin(x)^2*exp(-x/2)-1', -1)
```

(Notice that no dot is required now before ^ or *, although using the dot causes no problem.) How does this result compare to the approximate result you obtained by the graphical method above?

Now we again consider the pair of equations, $y = \sin^2(x)\, e^{-x/2} - 1$ and $y = 10\sin(x)/x$:

```
>> clear;  x = fzero('10*sin(x)/x - sin(x)^2*exp(-x/2) + 1', 3.5)
>> y1 = 10*sin(x)/x,   y2 = sin(x)^2*exp(-x/2) - 1
```

Again, note that the numerical method is more accurate. So why bother with the graphical method at all? To see about where you need to have "fzero" seek a solution!

Numerical solution of two or more equations in implicit forms (e.g., f1(x,y)=0, f2(x,y)=0)

We use the "fminsearch" command. Consider as an example the following two implicit equations (already in MATLAB format):

```
0.5 = (200+3*x+4*y)^2 / ((20+2*x+3*y)^2 * x)
10 = (20+2*x+3*y)*y / x
```

Use the following steps:

1. **Move everything to the left-hand side in all equations.**
2. **Using MATLAB's edit window, create a function that calculates the deviations from 0 when arbitrary values are used for the variables. The final output of the function is the sum of the squares of these deviations. Following is the code for our example, and would be saved as meth_reac.m in MATLAB's Current Directory, which must also be in the Path (File/Set Path).**

```
function out=meth_reac(guesses)
% function to calculate the values of x and y
% satisfying the two equibiliubrium relations
% for a methanol reactor, CO + 2H2 = CH3OH
% CO2 + 3H2 = CH3OH + H2O (coal to chems case study)
% x is the molar flow rate of CO, y of CO2 (kmol/h)
% After saving: >> fminsearch('meth_reac',[25,3]); x=ans(1), y=ans(2)
% Any positive values work as initial guesses in this example
x=guesses(1); y=guesses(2);
eq1 = 0.5 -(200+3*x+4*y)^2/(20+2*x+3*y)^2/x;
eq2 = 10 -(20+2*x+3*y)*y/x;
out=eq1^2+eq2^2;
end
```

3. **In the Command window, type "fminsearch('function name',[guessed values]). For our example,**

```
>> fminsearch('meth_reac',[25,3]); x=ans(1), y=ans(2)
```

You can check your result by using the symbolic "solve" command, as follows for our example:

```
>> clear, syms x y
>> eq1='0.5=(200+3*x+4*y)^2/(20+2*x+3*y)^2/x'
>> eq2='10=(20+2*x+3*y)*y/x'
>> [x y]=solve(eq1,eq2,x,y)
```

Note that several solutions are produced, from which you have to select the physically reasonable one. If this is the third solution, then to obtain double-precision numerical variables we use:

```
>> x=double(x(3)), y=double(y(3))
```

(The equations in this example are based on the equilibrium relationships for a chemical reactor making methanol from CO, CO_2 and H_2 as part of an existing plant that produces a variety chemicals starting with the gasification of coal, i.e. coal reacting with water at high temperature.)

Numerical solution of simultaneous linear equations using the matrix backslash (\) method

MATLAB uses Gaussian elimination to solve systems of linear equations via the backslash matrix command (\). As an example, consider the following three equations:

2x-3y+4z=5
y+4z+x=10
-2z+3x+4y=0

The procedure is as follows:

1. **Each equation must first have each variable in the same order on the left-hand side, with the constants on the right-hand side. This should be done using pencil and paper. Students commonly assume they can do this while entering the information in MATLAB, and frequently make mistakes doing so. For our example, we write:**

   2x - 3y + 4z = 5
   x + y + 4z = 10
   3x + 4y - 2z = 0

2. **In MATLAB, create matrices of coefficients for the variables and constants on the right-hand sides. Thus**

for our example:

>> clear, C = [2,-3,4; 1,1,4; 3,4,-2], B = [5; 10; 0]

Thus, in matrix notation our system of equations is represented by C*A=B where A = [x; y; z],

or: $\begin{bmatrix} 2 & -3 & 4 \\ 1 & 1 & 4 \\ 4 & 4 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 5 \\ 10 \\ 0 \end{bmatrix}$

3. **Solve for x, y and z using:**

>> A = C\B, x = A(1), y = A(2), z = A(3)

Check your results using: >> C*A - B

(Note that these are matrix operations; you MUST NOT use a dot (.) before * or \)

Compare this result with that obtained using MATLAB symbolic mathematics for the same system of equations.

Another application of the backslash matrix operator is to fit data to a non-linear equation. We use an example with the following y versus data:

>> t = [0, .3, .8, 1.1, 1.6, 2.3]'; y = [0.5, 0.82, 1.14, 1.25, 1.35, 1.40]'; plot(t,y,'o')

Notice that this does not give a straight line plot. Instead we will try a quadratic fit (without using the Basic Fitting tool on the graph window or the "polyfit" command). That is, we want to find the values of the coefficients (a_0, a_1, a_2) that best fit these data to the form $y = a_0 + a_1 t + a_2 t^2$. We have 6 values each of t and y, from which we generate 6 equations in the format $a_0 + ta_1 + t^2 a_2 = y$ matching that required for solution of simultaneous linear equations:

$a_0 = 0.5$

$a_0 + 0.3a_1 + 0.3^2 a_2 = 0.82$

$a_0 + 0.8a_1 + 0.8^2 a_2 = 1.14$

$a_0 + 1.1a_1 + 1.1^2 a_2 = 1.25$

$a_0 + 1.6a_1 + 1.6^2 a_2 = 1.35$

$a_0 + 2.3a_1 + 2.3^2 a_2 = 1.40$

Thus we have six simultaneous equations, but only 3 unknowns. We cannot expect to get values for a_0, a_1 and a_2 that would exactly satisfy all 6 equations. The backslash operator gives the least squares values instead. We must put these 6 equations in the form C*A=B, as follows (assuming t and y have already been entered, as above):

```
>> C(:,1) = ones(length(t),1)
>> C(:,2)=t
>> C(:,3)=t.^2
>> B = y
>> A = C\B
>> a0 = A(1), a1 = A(2), a2 = A(3)
>> tfit=0:0.1:2.3; yfit = a0 + a1*tfit + a2*tfit.^2;
>> plot(t,y,'o',tfit,yfit);
```

This is a general method and not confined to fitting data to polynomials. See fitting data to non-polynomial equations.

Find a solution of equation

E.2.2 Solving a System of Algebraic Equations: Example 6-7

In this example a system of three algebraic equations

$$0 = C_H - C_{H_0} + \left(k_1 C_H^{1/2} C_M + k_2 C_H^{1/2} C_X\right)\tau$$
$$0 = C_M - C_{M_0} + k_1 C_H^{1/2} C_M \tau$$
$$0 = \left(k_1 C_H^{1/2} C_M + k_2 C_H^{1/2} C_X\right)\tau - C_X$$

is solved for three variables, C_H, C_M, and C_X.

Step 1: To solve these equations using MATLAB the constants are declared to be symbolic, the values for the constants are entered in the equations and the equations are entered as eq1, eq2, and eq3 in the following form: *(eq1=symbolic equation).*

$h = C_H$	$C_{H0} = .021$
$k_1 = 55.2$	
$m = C_M$	$C_{M0} = .0105$
$k_2 = 30.2$	
$x = C_X$	$t = 0.5$

```
syms h m x;

eq1=h-.021+(55.2*m*h^0.5+30.2*x*h^0.5)*0.5;
eq2=m-0.0105+(55.2*m*h^0.5)*0.5;
eq3=(55.2*m*h^0.5-30.2*x*h^0.5)*0.5-x;
```

Step 2: Next, to solve this system of equations, we type

```
S=solve(eq1,eq2,eq3);
```

The answers can be displayed by typing the following commands:

```
S.h
ans = .894358044991691397750649762302 42e-2

S.m
 ans = .290846967571707015075384932598 10e-2

S.x
ans = .312664109848277367599879897106 24e-2
```

Therefore, C_H=0.00894, C_M=0.00291, and C_X=0.00313.

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

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

starting at `x0 = [-5 -5]`.

```
function F = myfun(x)
F = [2*x(1) - x(2) - exp(-x(1));
      -x(1) + 2*x(2) - exp(-x(2))];
```

Linear Programming with Equalities and Inequaliti

$$\min f^T x \quad \text{such that} \quad \begin{matrix} Aeq \cdot x = beq \\ A \cdot x \le b \\ x \ge 0 \end{matrix}$$

```
load sc50b
[x,fval,exitflag,output] = ...
    linprog(f,A,b,Aeq,beq,lb,[],[],optimset('Display','iter'));
```

Tutorial for the Optimization Toolbox.

This is a demonstration for the medium-scale algorithms in the Optimization Toolbox. It closely follows the Tutorial section of the users' guide.

All the principles outlined in this demonstration apply to the other nonlinear solvers: FGOALATTAIN, FMINIMAX, LSQNONLIN, FSOLVE.

The routines differ from the Tutorial Section examples in the User's Guide only in that some objectives are anonymous functions instead of M-file functions.

Contents

Unconstrained optimization example

Consider initially the problem of finding a minimum of the function:

$$f(x) = \exp(x(1)) \cdot (4x(1)^2 + 2x(2)^2 + 4x(1).x(2) + 2x(2) + 1)$$

Define the objective to be minimized as an anonymous function:

```
fun = @(x) exp(x(1)) * (4*x(1)^2 + 2*x(2)^2 + 4*x(1)*x(2) + 2*x(2) + 1)
```

```
fun =

    @(x) exp(x(1)) * (4*x(1)^2 + 2*x(2)^2 + 4*x(1)*x(2) + 2*x(2) + 1)
```

Take a guess at the solution:

```
x0 = [-1; 1];
```

Set optimization options: turn off the large-scale algorithms (the default):

```
options = optimset( 'LargeScale','off');
```

Call the unconstrained minimization function:

```
[x, fval, exitflag, output] = fminunc(fun, x0, options);
```

```
Optimization terminated: relative infinity-norm of gradient less than optio
```

The optimizer has found a solution at:

```
    x
```

```
x =

    0.5000
   -1.0000
```

The function value at the solution is:

```
    fval
```

```
fval =

  1.0983e-015
```

The total number of function evaluations was:

```
    output.funcCount
```

```
ans =

    66
```

Constrained optimization example: inequalities

Consider the above problem with two additional constraints:

$$\text{minimize} \quad f(x) = \exp(x(1)) \,.\, (4x(1)^2 + 2x(2)^2 + 4x(1).x(2) + 2x(2) + 1)$$

$$\text{subject to} \quad 1.5 + x(1).x(2) - x(1) - x(2) \leq 0$$
$$- x(1).x(2) \leq 10$$

The objective function this time is contained in an M-file, objfun.m:

```
    type objfun
```

```
function f = objfun(x)
% Objective function

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.5.4.2 $  $Date: 2004/04/06 01:10:28 $

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

The constraints are also defined in an M-file, confun.m:

```
type confun
```

```
function [c, ceq] = confun(x)
% Nonlinear inequality constraints:

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.6.4.2 $  $Date: 2004/04/06 01:10:16 $

c = [1.5 + x(1)*x(2) - x(1) - x(2);
     -x(1)*x(2) - 10];
% No nonlinear equality constraints:
ceq = [];
```

Take a guess at the solution:

```
x0 = [-1 1];
```

Set optimization options: turn off the large-scale algorithms (the default) and turn on the display of results at each iteration:

```
options = optimset( 'LargeScale','off','Display','iter');
```

Call the optimization algorithm. We have no linear equalities or inequalities or bounds, so we pass [] for those arguments:

```
[x,fval,exitflag,output] = fmincon(@objfun,x0,[],[],[],[],[],[],@confun
```

```
                                max                  Directional   First-or
 Iter F-count        f(x)   constraint    Step-size   derivative   optimali
    0      3       1.8394          0.5
    1      7      1.85127     -0.09197            1       -0.027        0.7
    2     11     0.300167         9.33            1       -0.825        0.3
    3     15     0.529835       0.9209            1        0.302        0.2
    4     20     0.186965       -1.517          0.5       -0.437         0.
    5     24    0.0729085       0.3313            1      -0.0715        0.0
    6     28    0.0353323     -0.03303            1       -0.026       0.02
    7     32    0.0235566     0.003184            1     -0.00963      0.005
    8     36    0.0235504   9.032e-008            1   -6.22e-006    8.51e-0
Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
                                       1
                                       2
```

A solution to this problem has been found at:

```
    x
```

```
x =

   -9.5474    1.0474
```

The function value at the solution is:

```
    fval
```

```
fval =

    0.0236
```

Both inequality constraints are satisfied (and active) at the solution:

```
    [c, ceq] = confun(x)
```

```
c =

  1.0e-007 *

   -0.9032
    0.9032


ceq =

     []
```

The total number of function evaluations was:

```
    output.funcCount
```

```
ans =

    36
```

Constrained optimization example: inequalities and bounds

Consider the previous problem with additional bound constraints:

$$\text{minimize} \quad f(x) = \exp(x(1)) \cdot (4x(1)^2 + 2x(2)^2 + 4x(1).x(2) + 2x(2) + 1)$$

```
 subject to  1.5 + x(1).x(2) - x(1) - x(2) <= 0
                 - x(1).x(2)               <= 10

        and                     x(1)        >= 0
                                       x(2) >= 0
```

As in the previous example, the objective and constraint functions are defined in M-files. The file objfun.m contains the objective:

```
type objfun
```

```
function f = objfun(x)
% Objective function

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.5.4.2 $  $Date: 2004/04/06 01:10:28 $

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

The file confun.m contains the constraints:

```
type confun
```

```
function [c, ceq] = confun(x)
% Nonlinear inequality constraints:

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.6.4.2 $  $Date: 2004/04/06 01:10:16 $

c = [1.5 + x(1)*x(2) - x(1) - x(2);
     -x(1)*x(2) - 10];
% No nonlinear equality constraints:
ceq = [];
```

Set the bounds on the variables:

```
lb = zeros(1,2); % Lower bounds x >= 0
ub = [];         % No upper bounds
```

Again, make a guess at the solution:

```
x0 = [-1 1];
```

Set optimization options: turn off the large-scale algorithms (the default). This time we do not turn on the Display option.

```
options = optimset( 'LargeScale','off');
```

Run the optimization algorithm:

```
[x,fval,exitflag,output] = fmincon(@objfun,x0,[],[],[],[],lb,ub,@confun
```

```
Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
    1                                1
```

The solution to this problem has been found at:

```
x
```

```
x =

         0    1.5000
```

The function value at the solution is:

```
fval
```

```
fval =

    8.5000
```

The constraint values at the solution are:

```
[c, ceq] = confun(x)
```

```
c =

     0
   -10


ceq =

     []
```

The total number of function evaluations was:

```
output.funcCount
```

```
ans =

    15
```

Constrained optimization example: user-supplied gradients

Optimization problems can be solved more efficiently and accurately if gradients are supplied by the user. This demo shows how this may be performed. We again solve the inequality-constrained problem

```
                                      2        2
 minimize  f(x) = exp(x(1)) . (4x(1)  + 2x(2)  + 4x(1).x(2) + 2x(2) + 1)

 subject to  1.5 + x(1).x(2) - x(1) - x(2) <= 0
                 - x(1).x(2)               <= 10
```

The objective function and its gradient are defined in the M-file objfungrad.m:

```
    type objfungrad
```

```
function [f, G] = objfungrad(x)
% objective function:

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.6.4.1 $  $Date: 2004/02/07 19:13:23 $

f =exp(x(1)) * (4*x(1)^2 + 2*x(2)^2 + 4*x(1)*x(2) + 2*x(2) + 1);
% gradient (partial derivatives) 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);
G = [ t + exp(x(1)) * (8*x(1) + 4*x(2)),
      exp(x(1))*(4*x(1)+4*x(2)+2)];
```

The constraints and their partial derivatives are contained in the M-file confungrad:

```
    type confungrad
```

```
function [c, ceq, dc, dceq] = confungrad(x)
% Nonlinear inequality constraints:

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.5.4.1 $  $Date: 2004/02/07 19:12:54 $

c = [1.5 + x(1)*x(2) - x(1) - x(2);
     -x(1)*x(2) - 10];
% Gradient (partial derivatives) of nonlinear inequality constraints:
dc = [x(2)-1, -x(2);
      x(1)-1, -x(1)];
% no nonlinear equality constraints (and gradients)
ceq = [];
dceq = [];
```

Make a guess at the solution:

```
x0 = [-1; 1];
```

Set optimization options: since we are supplying the gradients, we have the choice to use either the medium- or the large-scale algorithm; we will continue to use the same algorithm for comparison purposes.

```
options = optimset( 'LargeScale','off');
```

We also set options to use the gradient information in the objective and constraint functions. Note: these options MUST be turned on or the gradient information will be ignored.

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

Call the optimization algorithm:

```
[x,fval,exitflag,output] = fmincon(@objfungrad,x0,[],[],[],[],[],[],
                                   @confungrad,options);
```

```
Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
                                      1
                                      2
```

As before, the solution to this problem has been found at:

```
x
```

```
x =

   -9.5474
    1.0474
```

The function value at the solution is:

```
fval
```

```
fval =

    0.0236
```

Both inequality constraints are active at the solution:

```
[c, ceq] = confungrad(x)
```

```
c =

  1.0e-007 *

   -0.9032
    0.9032


ceq =

     []
```

The total number of function evaluations was:

```
output.funcCount
```

```
ans =

    18
```

Constrained optimization example: equality constraints

Consider the above problem with an additional equality constraint:

$$\begin{aligned} \text{minimize} \quad & f(x) = \exp(x(1)) \,.\, (4x(1)^2 + 2x(2)^2 + 4x(1).x(2) + 2x(2) + 1) \\ \text{subject to} \quad & x(1)^2 + x(2) = 1 \\ & -x(1).x(2) \leq 10 \end{aligned}$$

Like in previous examples, the objective function is defined in the M-file objfun.m:

```
type objfun
```

```
function f = objfun(x)
% Objective function

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.5.4.2 $  $Date: 2004/04/06 01:10:28 $

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

The M-file confuneq.m contains the equality and inequality constraints:

```
type confuneq
```

```
function [c, ceq] = confuneq(x)
% Nonlinear inequality constraints:

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.7.4.1 $  $Date: 2004/02/07 19:12:53 $

c = -x(1)*x(2) - 10;
% Nonlinear equality constraint:
ceq = x(1)^2 + x(2) - 1;
```

Again, make a guess at the solution:

```
x0 = [-1 1];
```

Set optimization options: turn off the large-scale algorithms (the default):

```
options = optimset( 'LargeScale','off');
```

Call the optimization algorithm:

```
[x,fval,exitflag,output] = fmincon(@objfun,x0,[],[],[],[],[],[],@confun
```

```
Optimization terminated: first-order optimality measure less than options.T
 and maximum constraint violation is less than options.TolCon.
No active inequalities
```

The solution to this problem has been found at:

```
x
```

```
x =

   -0.7529    0.4332
```

The function value at the solution is:

```
    fval
fval =

    1.5093
```

The constraint values at the solution are:

```
    [c, ceq] = confuneq(x)
c =

   -9.6739


ceq =

  6.3038e-009
```

The total number of function evaluations was:

```
    output.funcCount
ans =

    27
```

Changing the default termination tolerances

Consider the original unconstrained problem we solved first:

$$\text{minimize} \quad f(x) = \exp(x(1)) \cdot (4x(1)^2 + 2x(2)^2 + 4x(1).x(2) + 2x(2) + 1)$$

This time we will solve it more accurately by overriding the default termination criteria (options.TolX and options.TolFun).

Create an anonymous function of the objective to be minimized:

```
    fun = @(x) exp(x(1)) * (4*x(1)^2 + 2*x(2)^2 + 4*x(1)*x(2) + 2*x(2) + 1)
```

```
fun =

    @(x) exp(x(1)) * (4*x(1)^2 + 2*x(2)^2 + 4*x(1)*x(2) + 2*x(2) + 1)
```

Again, make a guess at the solution:

```
x0 = [-1; 1];
```

Set optimization options: turn off the large-scale algorithms (the default):

```
options = optimset( 'LargeScale','off');
```

Override the default termination criteria:

```
% Termination tolerance on X and f.
options = optimset(options, 'TolX',1e-3,'TolFun',1e-3);
```

Call the optimization algorithm:

```
[x, fval, exitflag, output] = fminunc(fun, x0, options);
```

```
Optimization terminated: relative infinity-norm of gradient less than optio
```

The optimizer has found a solution at:

```
x
```

```
x =

    0.4998
   -0.9997
```

The function value at the solution is:

```
fval
```

```
fval =

  2.0368e-007
```

The total number of function evaluations was:

```
output.funcCount
```

```
ans =

    60
```

Set optimization options: turn off the large-scale algorithms (the default):

```
options = optimset( 'LargeScale','off');
```

If we want a tabular display of each iteration we can set options.Display = 'iter' as follows:

```
options = optimset(options,  'Display','iter');

[x, fval, exitflag, output] = fminunc(fun, x0, options);
```

```
                                                  Gradient's
 Iteration  Func-count       f(x)        Step-size      infinity-norm
     0           3           1.8394                        0.736
     1           9          1.72428       0.368157           0.257
     2          27        0.0845289        22.5704           0.923
     3          51         0.072564      0.0012394            1.05
     4          54       0.00450951              1            0.29
     5          57     1.15036e-005              1           0.014
     6          60     2.03677e-007              1         0.00107
     7          63     3.47346e-012              1       9.34e-006
     8          66     1.09827e-015              1       7.37e-008
Optimization terminated: relative infinity-norm of gradient less than optio
```

At each major iteration the table displayed consists of: (i) number of function evaluations, (ii) function value, (iii) step length used in the line search, (iv) gradient in the direction of search.

Tutorial for the Optimization Toolbox.

This is a demonstration for the medium-scale algorithms in the Optimization Toolbox. It closely follows the Tutorial section of the users' guide.

All the principles outlined in this demonstration apply to the other nonlinear solvers: FGOALATTAIN, FMINIMAX, LSQNONLIN, FSOLVE.

The routines differ from the Tutorial Section examples in the User's Guide only in that some objectives are anonymous functions instead of M-file functions.

Contents

Unconstrained optimization example

Consider initially the problem of finding a minimum of the function:

$$f(x) = \exp(x(1)) \cdot (4x(1)^2 + 2x(2)^2 + 4x(1).x(2) + 2x(2) + 1)$$

Define the objective to be minimized as an anonymous function:

```
fun = @(x) exp(x(1)) * (4*x(1)^2 + 2*x(2)^2 + 4*x(1)*x(2) + 2*x(2) + 1)
```

```
fun =

    @(x) exp(x(1)) * (4*x(1)^2 + 2*x(2)^2 + 4*x(1)*x(2) + 2*x(2) + 1)
```

Take a guess at the solution:

```
x0 = [-1; 1];
```

Set optimization options: turn off the large-scale algorithms (the default):

```
options = optimset( 'LargeScale','off');
```

Call the unconstrained minimization function:

```
[x, fval, exitflag, output] = fminunc(fun, x0, options);
```

```
Optimization terminated: relative infinity-norm of gradient less than optio
```

The optimizer has found a solution at:

```
    x
```

```
x =

    0.5000
   -1.0000
```

The function value at the solution is:

```
    fval
```

```
fval =

  1.0983e-015
```

The total number of function evaluations was:

```
    output.funcCount
```

```
ans =

    66
```

Constrained optimization example: inequalities

Consider the above problem with two additional constraints:

$$\text{minimize} \quad f(x) = \exp(x(1)) \,.\, (4x(1)^2 + 2x(2)^2 + 4x(1).x(2) + 2x(2) + 1)$$

$$\text{subject to} \quad 1.5 + x(1).x(2) - x(1) - x(2) \leq 0$$
$$- x(1).x(2) \leq 10$$

The objective function this time is contained in an M-file, objfun.m:

```
    type objfun
```

```
function f = objfun(x)
% Objective function

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.5.4.2 $  $Date: 2004/04/06 01:10:28 $

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

The constraints are also defined in an M-file, confun.m:

```
type confun
```

```
function [c, ceq] = confun(x)
% Nonlinear inequality constraints:

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.6.4.2 $  $Date: 2004/04/06 01:10:16 $

c = [1.5 + x(1)*x(2) - x(1) - x(2);
     -x(1)*x(2) - 10];
% No nonlinear equality constraints:
ceq = [];
```

Take a guess at the solution:

```
x0 = [-1 1];
```

Set optimization options: turn off the large-scale algorithms (the default) and turn on the display of results at each iteration:

```
options = optimset( 'LargeScale','off','Display','iter');
```

Call the optimization algorithm. We have no linear equalities or inequalities or bounds, so we pass [] for those arguments:

```
[x,fval,exitflag,output] = fmincon(@objfun,x0,[],[],[],[],[],[],@confun
```

```
                                max                  Directional   First-or
 Iter F-count        f(x)   constraint    Step-size   derivative   optimali
    0      3       1.8394          0.5
    1      7      1.85127     -0.09197            1       -0.027        0.7
    2     11     0.300167         9.33            1       -0.825        0.3
    3     15     0.529835       0.9209            1        0.302        0.2
    4     20     0.186965       -1.517          0.5       -0.437         0.
    5     24    0.0729085       0.3313            1      -0.0715        0.0
    6     28    0.0353323     -0.03303            1       -0.026       0.02
    7     32    0.0235566     0.003184            1     -0.00963      0.005
    8     36    0.0235504   9.032e-008            1   -6.22e-006    8.51e-0
Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
                                       1
                                       2
```

A solution to this problem has been found at:

```
    x
```

```
x =

   -9.5474    1.0474
```

The function value at the solution is:

```
    fval
```

```
fval =

    0.0236
```

Both inequality constraints are satisfied (and active) at the solution:

```
    [c, ceq] = confun(x)
```

```
c =

  1.0e-007 *

   -0.9032
    0.9032


ceq =

     []
```

The total number of function evaluations was:

```
    output.funcCount
```

```
ans =

    36
```

Constrained optimization example: inequalities and bounds

Consider the previous problem with additional bound constraints:

$$\text{minimize} \quad f(x) = \exp(x(1)) \cdot (4x(1)^2 + 2x(2)^2 + 4x(1).x(2) + 2x(2) + 1)$$

```
 subject to  1.5 + x(1).x(2) - x(1) - x(2) <= 0
                 - x(1).x(2)               <= 10

        and                     x(1)        >= 0
                                       x(2) >= 0
```

As in the previous example, the objective and constraint functions are defined in M-files. The file objfun.m contains the objective:

```
type objfun
```

```
function f = objfun(x)
% Objective function

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.5.4.2 $  $Date: 2004/04/06 01:10:28 $

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

The file confun.m contains the constraints:

```
type confun
```

```
function [c, ceq] = confun(x)
% Nonlinear inequality constraints:

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.6.4.2 $  $Date: 2004/04/06 01:10:16 $

c = [1.5 + x(1)*x(2) - x(1) - x(2);
     -x(1)*x(2) - 10];
% No nonlinear equality constraints:
ceq = [];
```

Set the bounds on the variables:

```
lb = zeros(1,2); % Lower bounds x >= 0
ub = [];         % No upper bounds
```

Again, make a guess at the solution:

```
x0 = [-1 1];
```

Set optimization options: turn off the large-scale algorithms (the default). This time we do not turn on the Display option.

```
options = optimset( 'LargeScale','off');
```

Run the optimization algorithm:

```
[x,fval,exitflag,output] = fmincon(@objfun,x0,[],[],[],[],lb,ub,@confun
```

```
Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
    1                                1
```

The solution to this problem has been found at:

```
x
```

```
x =

         0    1.5000
```

The function value at the solution is:

```
fval
```

```
fval =

    8.5000
```

The constraint values at the solution are:

```
[c, ceq] = confun(x)
```

```
c =

     0
   -10


ceq =

     []
```

The total number of function evaluations was:

```
output.funcCount
```

```
ans =

    15
```

Constrained optimization example: user-supplied gradients

Optimization problems can be solved more efficiently and accurately if gradients are supplied by the user. This demo shows how this may be performed. We again solve the inequality-constrained problem

```
                                      2        2
 minimize  f(x) = exp(x(1)) . (4x(1)  + 2x(2)  + 4x(1).x(2) + 2x(2) + 1)

 subject to  1.5 + x(1).x(2) - x(1) - x(2) <= 0
                 - x(1).x(2)               <= 10
```

The objective function and its gradient are defined in the M-file objfungrad.m:

```
    type objfungrad
```

```
function [f, G] = objfungrad(x)
% objective function:

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.6.4.1 $  $Date: 2004/02/07 19:13:23 $

f =exp(x(1)) * (4*x(1)^2 + 2*x(2)^2 + 4*x(1)*x(2) + 2*x(2) + 1);
% gradient (partial derivatives) 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);
G = [ t + exp(x(1)) * (8*x(1) + 4*x(2)),
      exp(x(1))*(4*x(1)+4*x(2)+2)];
```

The constraints and their partial derivatives are contained in the M-file confungrad:

```
    type confungrad
```

```
function [c, ceq, dc, dceq] = confungrad(x)
% Nonlinear inequality constraints:

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.5.4.1 $  $Date: 2004/02/07 19:12:54 $

c = [1.5 + x(1)*x(2) - x(1) - x(2);
     -x(1)*x(2) - 10];
% Gradient (partial derivatives) of nonlinear inequality constraints:
dc = [x(2)-1, -x(2);
      x(1)-1, -x(1)];
% no nonlinear equality constraints (and gradients)
ceq = [];
dceq = [];
```

Make a guess at the solution:

```
x0 = [-1; 1];
```

Set optimization options: since we are supplying the gradients, we have the choice to use either the medium- or the large-scale algorithm; we will continue to use the same algorithm for comparison purposes.

```
options = optimset( 'LargeScale','off');
```

We also set options to use the gradient information in the objective and constraint functions. Note: these options MUST be turned on or the gradient information will be ignored.

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

Call the optimization algorithm:

```
[x,fval,exitflag,output] = fmincon(@objfungrad,x0,[],[],[],[],[],[],
                                   @confungrad,options);
```

```
Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
                                      1
                                      2
```

As before, the solution to this problem has been found at:

```
x
```

```
x =

   -9.5474
    1.0474
```

The function value at the solution is:

```
fval
```

```
fval =

    0.0236
```

Both inequality constraints are active at the solution:

```
[c, ceq] = confungrad(x)
```

```
c =

  1.0e-007 *

   -0.9032
    0.9032


ceq =

     []
```

The total number of function evaluations was:

```
output.funcCount
```

```
ans =

    18
```

Constrained optimization example: equality constraints

Consider the above problem with an additional equality constraint:

$$\begin{aligned} \text{minimize} \quad & f(x) = \exp(x(1)) \,.\, (4x(1)^2 + 2x(2)^2 + 4x(1).x(2) + 2x(2) + 1) \\ \text{subject to} \quad & x(1)^2 + x(2) = 1 \\ & -x(1).x(2) \leq 10 \end{aligned}$$

Like in previous examples, the objective function is defined in the M-file objfun.m:

```
type objfun
```

```
function f = objfun(x)
% Objective function

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.5.4.2 $  $Date: 2004/04/06 01:10:28 $

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

The M-file confuneq.m contains the equality and inequality constraints:

```
type confuneq
```

```
function [c, ceq] = confuneq(x)
% Nonlinear inequality constraints:

%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.7.4.1 $  $Date: 2004/02/07 19:12:53 $

c = -x(1)*x(2) - 10;
% Nonlinear equality constraint:
ceq = x(1)^2 + x(2) - 1;
```

Again, make a guess at the solution:

```
x0 = [-1 1];
```

Set optimization options: turn off the large-scale algorithms (the default):

```
options = optimset( 'LargeScale','off');
```

Call the optimization algorithm:

```
[x,fval,exitflag,output] = fmincon(@objfun,x0,[],[],[],[],[],[],@confun
```

```
Optimization terminated: first-order optimality measure less than options.T
 and maximum constraint violation is less than options.TolCon.
No active inequalities
```

The solution to this problem has been found at:

```
x
```

```
x =

   -0.7529    0.4332
```

The function value at the solution is:

```
    fval
fval =

    1.5093
```

The constraint values at the solution are:

```
    [c, ceq] = confuneq(x)
c =

   -9.6739


ceq =

  6.3038e-009
```

The total number of function evaluations was:

```
    output.funcCount
ans =

    27
```

Changing the default termination tolerances

Consider the original unconstrained problem we solved first:

$$\text{minimize} \quad f(x) = \exp(x(1)) \cdot (4x(1)^2 + 2x(2)^2 + 4x(1).x(2) + 2x(2) + 1)$$

This time we will solve it more accurately by overriding the default termination criteria (options.TolX and options.TolFun).

Create an anonymous function of the objective to be minimized:

```
    fun = @(x) exp(x(1)) * (4*x(1)^2 + 2*x(2)^2 + 4*x(1)*x(2) + 2*x(2) + 1)
```

```
fun =

    @(x) exp(x(1)) * (4*x(1)^2 + 2*x(2)^2 + 4*x(1)*x(2) + 2*x(2) + 1)
```

Again, make a guess at the solution:

```
x0 = [-1; 1];
```

Set optimization options: turn off the large-scale algorithms (the default):

```
options = optimset( 'LargeScale','off');
```

Override the default termination criteria:

```
% Termination tolerance on X and f.
options = optimset(options, 'TolX',1e-3,'TolFun',1e-3);
```

Call the optimization algorithm:

```
[x, fval, exitflag, output] = fminunc(fun, x0, options);
```

```
Optimization terminated: relative infinity-norm of gradient less than optio
```

The optimizer has found a solution at:

```
x
```

```
x =

    0.4998
   -0.9997
```

The function value at the solution is:

```
fval
```

```
fval =

  2.0368e-007
```

The total number of function evaluations was:

```
output.funcCount
```

```
ans =

    60
```

Set optimization options: turn off the large-scale algorithms (the default):

```
options = optimset( 'LargeScale','off');
```

If we want a tabular display of each iteration we can set options.Display = 'iter' as follows:

```
options = optimset(options,  'Display','iter');

[x, fval, exitflag, output] = fminunc(fun, x0, options);
```

```
                                                  Gradient's
 Iteration  Func-count       f(x)        Step-size      infinity-norm
     0           3           1.8394                        0.736
     1           9          1.72428       0.368157           0.257
     2          27        0.0845289        22.5704           0.923
     3          51         0.072564      0.0012394            1.05
     4          54       0.00450951              1            0.29
     5          57     1.15036e-005              1           0.014
     6          60     2.03677e-007              1         0.00107
     7          63     3.47346e-012              1       9.34e-006
     8          66     1.09827e-015              1       7.37e-008
Optimization terminated: relative infinity-norm of gradient less than optio
```

At each major iteration the table displayed consists of: (i) number of function evaluations, (ii) function value, (iii) step length used in the line search, (iv) gradient in the direction of search.