IC-32 **Optimal Control** **Winter 2006**

Benoît Chachuat
ME C2 401, Ph: 33844, benoit.chachuat@epfl.ch

Problem Set #3

1. Consider the following NLP problem:

$$\min_{\mathbf{x}\in\mathbb{R}^3} f(\mathbf{x}) := x_1^2 + x_1x_2 + 2x_2^2 - 6x_1 - 2x_2 - 12x_3 \tag{1}$$

$$\begin{aligned} \text{s.t.}\quad & g_1(\mathbf{x}) := 2x_1^2 + x_2^2 \leq 15 \\ & g_2(\mathbf{x}) := x_1 - 2x_2 - x_3 \geq -3 \\ & x_1, x_2, x_3 \geq 0. \end{aligned} \tag{2}$$

(a) Find an optimal solution $\mathbf{x}^\star$ to this problem using the function `fmincon` in MATLAB®'s Optimization Toolbox:

- Make sure that the medium scale SQP algorithm, with Quasi-Newton update and line-search, is the selected solver in `fmincon`;
- Set the solution point tolerance, function tolerance and constraint tolerance in `fmincon` to 10^{-7};
- Specify the initial guess as $\mathbf{x}^0 = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^\mathsf{T}$;
- Make sure that the inequality constraint g_2 is treated as a *linear* constraint by `fmincon`;
- Solve the NLP by using a finite-difference approximation of the gradients of the objective function and constraints first. Then, resolve the problem by providing explicit expressions for the gradients of the objective function and constraints (ask `fmincon` to check for the gradients before running the optimization in this latter case);
- Make sure that the solver terminated successfully in each case.

(b) Repeat the numerical optimization from a different starting point, e.g., $\mathbf{x}^0 = \begin{pmatrix} 4 & 0 & 0 \end{pmatrix}^\mathsf{T}$. Does `fmincon` converge to the same solution point? Could this be expected?

(c) Get the values $\boldsymbol{\nu}^\star$ of the Lagrange multipliers at $\mathbf{x}^\star$, as well as the gradients of the objective function and constraints. Check that the optimal solution point is (i) a regular point and (ii) a KKT point.

(d) Consider the perturbed problems

$$\min_{\mathbf{x}\in\mathbb{R}^3} f(\mathbf{x}) := x_1^2 + x_1x_2 + 2x_2^2 - 6x_1 - 2x_2 - 12x_3 \tag{3}$$

$$\begin{aligned} \text{s.t.}\quad & g_1^\theta(\mathbf{x}) := 2x_1^2 + x_2^2 \leq \theta \\ & g_2(\mathbf{x}) := x_1 - 2x_2 - x_3 \geq 3 \\ & x_1, x_2, x_3 \geq 0, \end{aligned}$$

where θ stands for the perturbation parameter. Solve (3) for N equally spaced values of θ in the range $[0, 30]$ (e.g., use a resolution of 0.5 for θ). Let us denote the optimal solution point as $\boldsymbol{\xi}^\star(\theta)$ and the optimal value of the Lagrange multiplier associated to the perturbed constraint as $\omega^\star(\theta)$.

i. Plot $f(\boldsymbol{\xi}^\star(\theta))$ versus θ, and estimate the slope of this curve at $\theta = 15$. What does the corresponding value represent?

ii. Plot $\omega^\star(\theta)$ versus θ. Comment this plot and, in particular, explain the behavior at $\theta = 0$. What is be the slope of the curve $f(\boldsymbol{\xi}^\star(\theta))$ versus θ at $\theta = 0$?

fmincon

Find the minimum of a constrained nonlinear multivariable function

$$\min_x f(x) \quad \text{subject to} \quad \begin{aligned} &c(x) \le 0 \\ &ceq(x) = 0 \\ &A \cdot x \le b \\ &Aeq \cdot x \le beq \\ &lb \le x \le ub \end{aligned}$$

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

Syntax

```
x = fmincon(fun,x0,A,b)
x = fmincon(fun,x0,A,b,Aeq,beq)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options,P1,P2, ...)
[x,fval] = fmincon(...)
[x,fval,exitflag] = fmincon(...)
[x,fval,exitflag,output] = fmincon(...)
[x,fval,exitflag,output,lambda] = fmincon(...)
[x,fval,exitflag,output,lambda,grad] = fmincon(...)
[x,fval,exitflag,output,lambda,grad,hessian] = fmincon(...)
```

Description

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

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

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

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

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

`x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)` minimizes with the optimization parameters specified in the structure `options`.

`x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options,P1,P2,...)` passes the problem-dependent parameters `P1`, `P2`, etc., directly to the functions `fun` and `nonlcon`. Pass empty matrices as placeholders for `A`, `b`, `Aeq`, `beq`, `lb`, `ub`, `nonlcon`, and `options` if these arguments are not needed.

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

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

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

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

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

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

Arguments

The arguments passed into the function are described in Table 1-1. The arguments returned by the function are described in Table 1-2. Details relevant to `fmincon` are included below for `fun`, `nonlcon`, `options`, `exitflag`, `lambda`, and `output`.

<table>
<tr><td><code>fun</code></td><td>The function to be minimized. <code>fun</code> takes a vector <code>x</code> and returns a scalar value <code>f</code> of the objective function evaluated at <code>x</code>. You can specify <code>fun</code> to be an inline object. For example,
<pre>fun = inline('sin(x''*x)');</pre>Alternatively, <code>fun</code> can be a string containing the name of a function (an M-file, a built-in function, or a MEX-file). If <code>fun='myfun'</code> then the M-file function <code>myfun.m</code> would have the form
<pre>function f = myfun(x)
f = ...   % Compute function value at x</pre></td></tr>
<tr><td></td><td>If the gradient of <code>fun</code> can also be computed and <code>options.GradObj</code> is <code>'on'</code>, as set by
<pre>options = optimset('GradObj','on')</pre>then the function <code>fun</code> must return, in the second output argument, the gradient value <code>g</code>, a vector, at <code>x</code>. Note that by checking the value of <code>nargout</code> the function can avoid computing <code>g</code> when <code>fun</code> is called with only one output argument (in the case where the optimization algorithm only needs the value of <code>f</code> but not <code>g</code>):
<pre>function [f,g] = myfun(x)
f = ...   % compute the function value at x
if nargout > 1   % fun called with two output arguments
   g = ...   % compute the gradient evaluated at x
end</pre></td></tr>
</table>

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

	If the Hessian matrix can also be computed *and* `options.Hessian` is `'on'`, i.e., `options = optimset('Hessian','on')`, then the function `fun` must return the Hessian value `H`, a symmetric matrix, at `x` in a third output argument. Note that by checking the value of `nargout` we can avoid computing `H` when `fun` is called with only one or two output arguments (in the case where the optimization algorithm only needs the values of `f` and `g` but not `H`):
	`function [f,g,H] = myfun(x)`
	`f = ...     % Compute the objective function value at x`
	`if nargout > 1   % fun called with two output arguments`
	`   g = ...    % gradient of the function evaluated at x`
	`   if nargout > 2`
	`   H = ...    % Hessian evaluated at x`
	`end`
	The Hessian matrix is the second partial derivatives matrix of `f` at the point `x`. That is, the (`i`th,`j`th) component of `H` is the second partial derivative of `f` with respect to x_i and x_j, $\partial^2 f/\partial x_i \partial x_j$. The Hessian is by definition a symmetric matrix.

`nonlcon`	The function that computes the nonlinear inequality constraints `c(x)<=0` and nonlinear equality constraints `ceq(x)=0`. `nonlcon` is a string containing the name of a function (an M-file, a built-in, or a MEX-file). `nonlcon` takes a vector `x` and returns two arguments, a vector `c` of the nonlinear inequalities evaluated at `x` and a vector `ceq` of the nonlinear equalities evaluated at `x`. For example, if `nonlcon='mycon'` then the M-file `mycon.m` would have the form
	`function [c,ceq] = mycon(x)`
	`c = ...     % Compute nonlinear inequalities at x`
	`ceq = ...   % Compute the nonlinear equalities at x`

	If the gradients of the constraints can also be computed *and* `options.GradConstr` is `'on'`, as set by `options = optimset('GradConstr','on')` then the function `nonlcon` must also return, in the third and fourth output arguments, `GC`, the gradient of `c(x)`, and `GCeq`, the gradient of `ceq(x)`. Note that by checking the value of `nargout` the function can avoid computing `GC` and `GCeq` when `nonlcon` is called with only two output arguments (in the case where the optimization algorithm only needs the values of `c` and `ceq` but not `GC` and `GCeq`):

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

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

	where `GCeq(i,j)` is the partial derivative of `ceq(j)` with respect to `x(i)` (i.e., the `j`th column of `GCeq` is the gradient of the `j`th equality constraint `ceq(j)`).
`options`	Optimization parameter options. You can set or change the values of these parameters using the `optimset` function. Some parameters 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. We start by describing the `LargeScale` option since it states a *preference* for which algorithm to use. It is only a preference since certain conditions must be met to use the large-scale algorithm. For `fmincon`, the *gradient must be provided (*see the description of `fun` above to see how) or else the medium-scale algorithm will be used. • `LargeScale` - Use large-scale algorithm if possible when set to `'on'`. Use medium-scale algorithm when set to `'off'`.
	Parameters used by both the large-scale and medium-scale algorithms: • `Diagnostics` - Print diagnostic information about the function to be minimized. • `Display` - Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output. • `GradObj` - Gradient for the objective function defined by user. See the description of `fun` under the *Arguments* section above to see how to define the gradient in `fun`. The gradient *must* be provided to use the large-scale method. It is optional for the medium-scale method.
	• `MaxFunEvals` - Maximum number of function evaluations allowed. • `MaxIter` - Maximum number of iterations allowed. • `TolFun` - Termination tolerance on the function value. • `TolCon` - Termination tolerance on the constraint violation. • `TolX` - Termination tolerance on `x`.
	Parameters used by the large-scale algorithm only: • `Hessian` - Hessian for the objective function defined by user. See the description of `fun` under the *Arguments* section above to see how to define the Hessian in `fun`. • `HessPattern` - Sparsity pattern of the Hessian for finite-differencing. If it is not convenient to compute the sparse Hessian matrix `H` in `fun`, the large-scale method in `fmincon` can approximate `H` via sparse finite-differences (of the gradient) provided the *sparsity structure* of `H` -- i.e., locations of the nonzeros -- is supplied as the value for `HessPattern`. In the worst case, if the structure is unknown, you can set `HessPattern` to be a dense matrix and a full finite-difference approximation will be computed at each iteration (this is the default). This can be very expensive for large problems so it is usually worth the effort to determine the sparsity structure.
	• `MaxPCGIter` - Maximum number of PCG (preconditioned conjugate gradient) iterations (see the *Algorithm* section below). • `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. • `TypicalX` - Typical `x` values.
	Parameters used by the medium-scale algorithm only: • `DerivativeCheck` - Compare user-supplied derivatives (gradients of the objective

	and constraints) to finite-differencing derivatives. • `DiffMaxChange` - Maximum change in variables for finite-difference gradients. • `DiffMinChange` - Minimum change in variables for finite-difference gradients. • `LineSearchType` - Line search algorithm choice.
`exitflag`	Describes the exit condition: • `> 0` indicates that the function converged to a solution `x`. • `0` indicates that the maximum number of function evaluations or iterations was reached. • `< 0` indicates that the function did not converge to a solution.
`lambda`	A structure containing the Lagrange multipliers at the solution `x` (separated by constraint type): • `lambda.lower` for the lower bounds `lb`. • `lambda.upper` for the upper bounds `ub`. • `lambda.ineqlin` for the linear inequalities. • `lambda.eqlin` for the linear equalities. • `lambda.ineqnonlin` for the nonlinear inequalities. • `lambda.eqnonlin` for the nonlinear equalities.
`output`	A structure whose fields contain information about the optimization: • `output.iterations` - The number of iterations taken. • `output.funcCount` - The number of function evaluations. • `output.algorithm` - The algorithm used. • `output.cgiterations` - The number of PCG iterations (large-scale algorithm only). • `output.stepsize` - The final step size taken (medium-scale algorithm only). • `output.firstorderopt` - A measure of first-order optimality (large-scale algorithm only).

Examples

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

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

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

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

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

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

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

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

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

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

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

After 66 function evaluations, the solution is

```
x =
    24.0000
    12.0000
    12.0000
```

where the function value is

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

and linear inequality constraints evaluate to be `<= 0`

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

Notes

Large-scale optimization. To use the large-scale method, the gradient must be provided in `fun` (and `options.GradObj` set to `'on'`). A warning is given if no gradient is provided and `options.LargeScale` is not `'off'`. `fmincon` permits $g(x)$ to be an approximate gradient but this option is not recommended: the numerical behavior of most optimization codes is considerably more robust when the true gradient is used.

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

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

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

Several aspects of linearly constrained minimization should be noted:

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

  $$B = Aeq^T R^{-T}$$

  where `R`T is the Cholesky factor of the preconditioner. Therefore, there is a potential conflict between choosing an effective preconditioner and minimizing fill in B.

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

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

Algorithm

Large-scale optimization. By default `fmincon` will choose the large-scale algorithm *if* the user supplies the gradient in `fun` (and `GradObj` is `'on'` in `options`) *and* if *only* upper and lower bounds exists *or only* linear equality constraints exist. This algorithm is a subspace trust region method and is based on the interior-reflective Newton method described in [5],[6]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See the trust-region and preconditioned conjugate gradient method descriptions in the *Large-Scale Algorithms* chapter.

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

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

See also the SQP implementation section in the *Introduction to Algorithms* chapter for more details on the algorithm used.

Diagnostics

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

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

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

Limitations

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

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

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

Large-scale optimization. To use the large-scale algorithm, the user must supply the gradient in `fun` (and `GradObj` must be set `'on'` in `options`) , and only upper and lower bounds constraints may be specified, *or only* linear equality constraints must exist and `Aeq` cannot have more rows than columns. `Aeq` is typically sparse. See Table 1-4 for more information on what problem formulations are covered and what information must be provided.

Currently, if the analytical gradient is provided in `fun`, the `options` parameter `DerivativeCheck` cannot be used with the large-scale method to compare the analytic gradient to the finite-difference gradient. Instead, use the medium-scale method to check the derivative with `options` parameter `MaxIter` set to 0 iterations. Then run the problem with the large-scale method.

References

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

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

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

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

[5] 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.

[6] 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.

See Also

`fminbnd`, `fminsearch`, `fminunc`, `optimset`

[Previous | Help Desk | Next]