Constrained Nonlinear Optimization Examples

On this page…

Example: Nonlinear Inequality Constraints

If inequality constraints are added to Equation 6-16, the resulting problem can be solved by the `fmincon` function. For example, find x that solves

$$\min_x f(x) = e^{x_1}\left(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1\right). \tag{6-57}$$

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 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

$$\begin{aligned} &x_1x_2 - x_1 - x_2 + 1.5 \le 0, \\ &-x_1x_2 - 10 \le 0. \end{aligned} \tag{6-58}$$

Step 1: Write a file objfun.m for the objective function.

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

Step 2: Write a 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 3: Invoke constrained optimization routine.

```
x0 = [-1,1];     % Make a starting guess at the solution
```

```
options = optimset('Algorithm','active-set');
[x,fval] = ...
fmincon(@objfun,x0,[],[],[],[],[],[],@confun,options);
```

`fmincon` produces the solution `x` with function value `fval`:

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

You can evaluate the constraints at the solution by entering

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

This returns numbers close to zero, such as

```
c =
  1.0e-007 *
   -0.9032
    0.9032

ceq =
     []
```

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

Back to Top

Example: Bound Constraints

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

```
x = fmincon(@objfun,x0,[],[],[],[],lb,ub,@confun,options);
```

limits `x` to be within the range `lb ≤ x ≤ ub`.

To restrict `x` in Equation 6-57 to be greater than 0 (i.e., $x_1 \ge 0$, $x_1 \ge 0$), use the commands

```
x0 = [-1,1];          % Make a starting guess at the solution
lb = [0,0];           % Set lower bounds
ub = [ ];             % No upper bounds
options = optimset('Algorithm','active-set');
[x,fval] = ...
fmincon(@objfun,x0,[],[],[],[],lb,ub,@confun,options);
```

Note that to pass in the lower bounds as the seventh argument to `fmincon`, you must specify values for the third through sixth arguments. In this example, we specified `[]` for these arguments since there are no linear inequalities or linear equalities.

fmincon produces the following results:

```
x,fval
x =
        0   1.5000
fval =
```

```
        8.5000
```

When `lb` or `ub` contains fewer elements than `x`, only the first corresponding elements in `x` are bounded. Alternatively, if only some of the variables are bounded, then use `-inf` in `lb` for unbounded below variables and `inf` in `ub` for unbounded above variables. For example,

```
lb = [-inf 0];
ub = [10 inf];
```

bounds $x_1 \leq 10$, $x_2 \geq 0$. x_1 has no lower bound, and x_2 has no upper bound. Using `inf` and `-inf` give better numerical results than using a very large positive number or a very large negative number to imply lack of bounds.

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

Back to Top

Example: Constraints With Gradients

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

To solve Equation 6-57 using analytically determined gradients, do the following.

Step 1: Write a file for the objective function and gradient.

```
function [f,G] = objfungrad(x)
f = exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
% Gradient of the objective function
if nargout  > 1
    G = [ f + exp(x(1)) * (8*x(1) + 4*x(2)),
    exp(x(1))*(4*x(1)+4*x(2)+2)];
end
```

Step 2: Write a file for the nonlinear constraints and the gradients of the nonlinear constraints.

```
function [c,ceq,DC,DCeq] = confungrad(x)
c(1) = 1.5 + x(1) * x(2) - x(1) - x(2); %Inequality constraints
c(2) = -x(1) * x(2)-10;
% No nonlinear equality constraints
ceq=[];
% Gradient of the constraints
if nargout > 2
    DC= [x(2)-1, -x(2);
        x(1)-1, -x(1)];
    DCeq = [];
end
```

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

$$\nabla f = \begin{bmatrix} e^{x_1}\left(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1\right) + e^{x_1}\left(8x_1 + 4x_2\right) \\ e^{x_1}\left(4x_1 + 4x_2 + 2\right) \end{bmatrix}. \tag{6-59}$$

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

$$\begin{bmatrix} \frac{\partial c_1}{\partial x_1} & \frac{\partial c_2}{\partial x_1} \\ \frac{\partial c_1}{\partial x_2} & \frac{\partial c_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} x_2 - 1 & -x_2 \\ x_1 - 1 & -x_1 \end{bmatrix}. \tag{6-60}$$

Since you are providing the gradient of the objective in objfungrad.m and the gradient of the constraints in confungrad.m, you *must* tell fmincon that these files contain this additional information. Use optimset to turn the options GradObj and GradConstr to 'on' in the example's existing options structure:

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

If you do not set these options to 'on' in the options structure, fmincon does not use the analytic gradients.

The arguments lb and ub place lower and upper bounds on the independent variables in x. In this example, there are no bound constraints and so they are both set to [].

Step 3: Invoke the constrained optimization routine.

```
x0 = [-1,1];            % Starting guess
options = optimset('Algorithm','active-set');
options = optimset(options,'GradObj','on','GradConstr','on');
lb = [ ]; ub = [ ];   % No upper or lower bounds
[x,fval] = fmincon(@objfungrad,x0,[],[],[],[],lb,ub,...
   @confungrad,options);
```

The results:

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

[c,ceq] = confungrad(x) % Check the constraint values at x
c =
   1.0e-007 *
   -0.9032
    0.9032
ceq =
     []
```

Back to Top

Example: Constrained Minimization Using fmincon's Interior-Point Algorithm with Analytic Hessian

The fmincon interior-point algorithm can accept a Hessian function as an input. When you supply a Hessian, you may obtain a faster, more accurate solution to a constrained minimization problem.

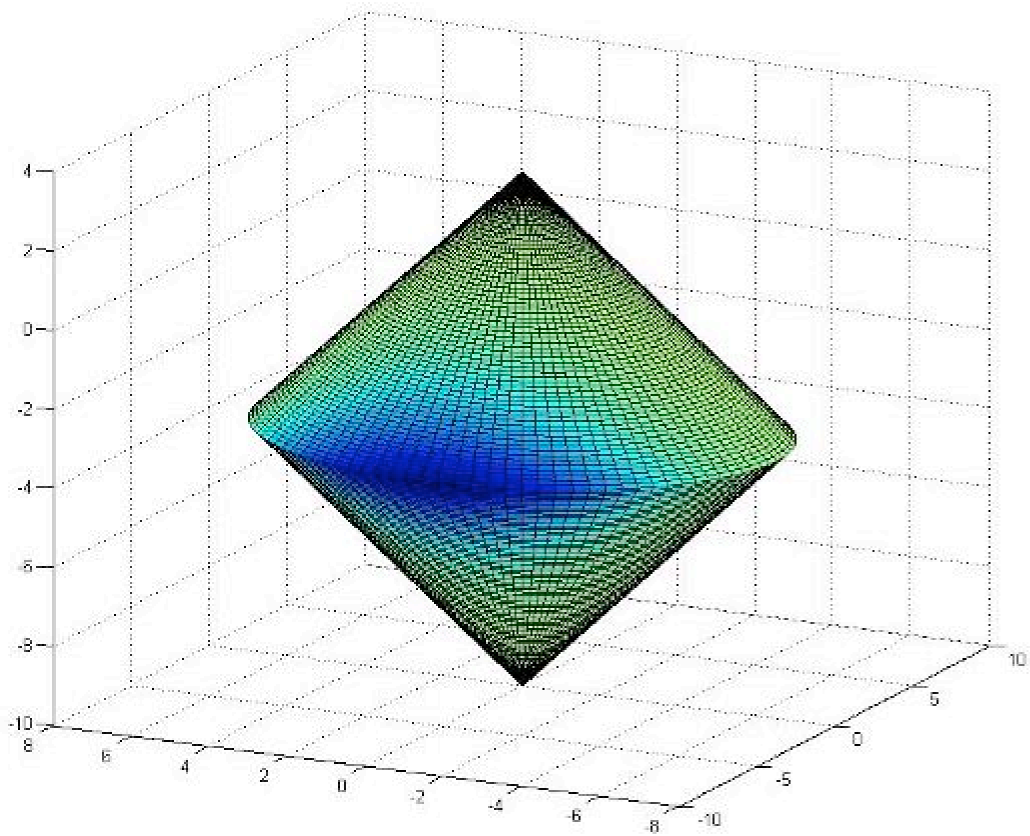
The constraint set for this example is the intersection of the interior of two cones—one pointing up, and one pointing down. The constraint function `c` is a two-component vector, one component for each cone. Since this is a three-dimensional example, the gradient of the constraint `c` is a 3-by-2 matrix.

```
function [c ceq gradc gradceq] = twocone(x)
% This constraint is two cones, z > -10 + r
% and z < 3 - r

ceq = [];
r = sqrt(x(1)^2 + x(2)^2);
c = [-10+r-x(3);
    x(3)-3+r];

if nargout > 2

    gradceq = [];
    gradc = [x(1)/r,x(1)/r;
       x(2)/r,x(2)/r;
       -1,1];

end
```

The objective function grows rapidly negative as the `x(1)` coordinate becomes negative. Its gradient is a three-element vector.

```
function [f gradf] = bigtoleft(x)
% This is a simple function that grows rapidly negative
% as x(1) gets negative
%
f=10*x(1)^3+x(1)*x(2)^2+x(3)*(x(1)^2+x(2)^2);

if nargout > 1

   gradf=[30*x(1)^2+x(2)^2+2*x(3)*x(1);
       2*x(1)*x(2)+2*x(3)*x(2);
       (x(1)^2+x(2)^2)];

end
```

Here is a plot of the problem. The shading represents the value of the objective function. You can see that the objective function is minimized near `x = [-6.5,0,-3.5]`:



▸ Code for generating the figure

The Hessian of the Lagrangian is given by the equation:

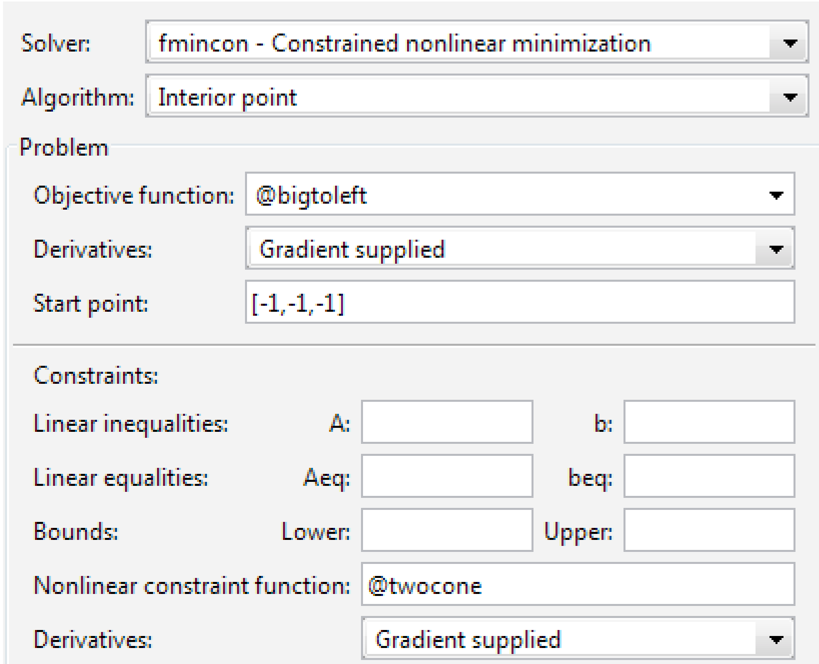
$$\nabla_{xx}^2 L(x,\lambda) = \nabla^2 f(x) + \sum \lambda_i \nabla^2 c_i(x) + \sum \lambda_i \nabla^2 ceq_i(x).$$

The following function computes the Hessian at a point `x` with Lagrange multiplier structure `lambda`:

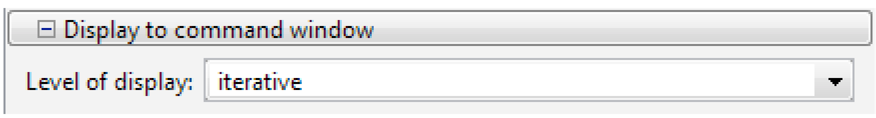
```
function h = hessinterior(x,lambda)

h = [60*x(1)+2*x(3),2*x(2),2*x(1);
    2*x(2),2*(x(1)+x(3)),2*x(2);
    2*x(1),2*x(2),0];% Hessian of f
r = sqrt(x(1)^2+x(2)^2);% radius
rinv3 = 1/r^3;
hessc = [(x(2))^2*rinv3,-x(1)*x(2)*rinv3,0;
    -x(1)*x(2)*rinv3,x(1)^2*rinv3,0;
    0,0,0];% Hessian of both c(1) and c(2)
h = h + lambda.ineqnonlin(1)*hessc + lambda.ineqnonlin(2)*hessc;
```

Run this problem using the interior-point algorithm in `fmincon`. To do this using the Optimization Tool:

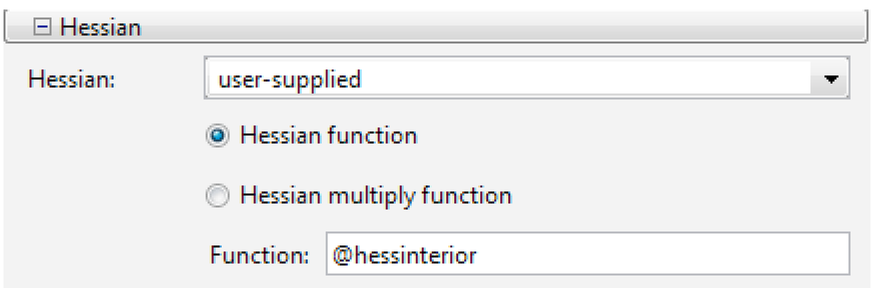
1. Set the problem as in the following figure.



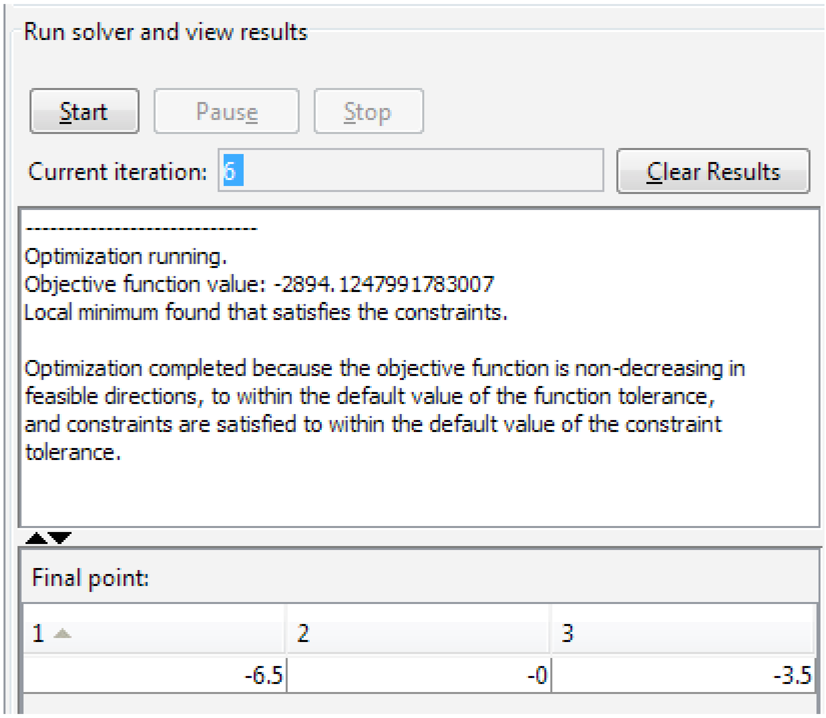
2. For iterative output, scroll to the bottom of the **Options** pane and select **Level of display**, `iterative`.



3. In the **Options** pane, give the analytic Hessian function handle.



4. Under **Run solver and view results**, click **Start**.



To perform the minimization at the command line:

1. Set `options` as follows:

```
options = optimset('Algorithm','interior-point',...
          'Display','iter','GradObj','on','GradConstr','on',...
          'Hessian','user-supplied','HessFcn',@hessinterior);
```

2. Run `fmincon` with starting point [–1,–1,–1], using the `options` structure:

```
[x fval mflag output]=fmincon(@bigtoleft,[-1,-1,-1],...
           [],[],[],[],[],[],@twocone,options)
```

The output is:

```
                                            First-order      Norm of
 Iter F-count            f(x)  Feasibility   optimality         step
    0       1   -1.300000e+001   0.000e+000   3.067e+001
    1       2   -2.011543e+002   0.000e+000   1.739e+002   1.677e+000
    2       3   -1.270471e+003   9.844e-002   3.378e+002   2.410e+000
    3       4   -2.881667e+003   1.937e-002   1.079e+002   2.206e+000
    4       5   -2.931003e+003   2.798e-002   5.813e+000   6.006e-001
    5       6   -2.894085e+003   0.000e+000   2.352e-002   2.800e-002
    6       7   -2.894125e+003   0.000e+000   5.981e-005   3.048e-005
 Local minimum found that satisfies the constraints.

 Optimization completed because the objective function is non-decreasing in
 feasible directions, to within the default value of the function tolerance,
 and constraints are satisfied to within the default value of the constraint tolera

 x =
    -6.5000   -0.0000   -3.5000

 fval =
```

```
 -2.8941e+003

mflag =
     1

output =
         iterations: 6
          funcCount: 7
     constrviolation: 0
           stepsize: 3.0479e-005
          algorithm: 'interior-point'
       firstorderopt: 5.9812e-005
        cgiterations: 3
            message: [1x782 char]
```

If you do not use a Hessian function, `fmincon` takes 9 iterations to converge, instead of 6:

```
options = optimset('Algorithm','interior-point',...
        'Display','iter','GradObj','on','GradConstr','on');
[x fval mflag output]=fmincon(@bigtoleft,[-1,-1,-1],...
           [],[],[],[],[],[],@twocone,options)

                                            First-order      Norm of
Iter F-count            f(x)  Feasibility   optimality         step
  0      1  -1.300000e+001   0.000e+000   3.067e+001
  1      2  -7.259551e+003   2.495e+000   2.414e+003   8.344e+000
  2      3  -7.361301e+003   2.529e+000   2.767e+001   5.253e-002
  3      4  -2.978165e+003   9.392e-002   1.069e+003   2.462e+000
  4      8  -3.033486e+003   1.050e-001   8.282e+002   6.749e-001
  5      9  -2.893740e+003   0.000e+000   4.186e+001   1.053e-001
  6     10  -2.894074e+003   0.000e+000   2.637e-001   3.565e-004
  7     11  -2.894124e+003   0.000e+000   2.340e-001   1.680e-004
  8     12  -2.894125e+003   2.830e-008   1.180e-001   6.374e-004
  9     13  -2.894125e+003   2.939e-008   1.423e-004   6.484e-004
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the default value of the function tolerance,
and constraints are satisfied to within the default value of the constraint tolera

x =
   -6.5000   -0.0000   -3.5000

fval =
 -2.8941e+003

mflag =
     1

output =
         iterations: 9
          funcCount: 13
     constrviolation: 2.9391e-008
           stepsize: 6.4842e-004
          algorithm: 'interior-point'
       firstorderopt: 1.4235e-004
```

```
        cgiterations: 0
             message: [1x782 char]
```

Both runs lead to similar solutions, but the F-count and number of iterations are lower when using an analytic Hessian.

Back to Top

Example: Equality and Inequality Constraints

For routines that permit equality constraints, nonlinear equality constraints must be computed with the nonlinear inequality constraints. For linear equalities, the coefficients of the equalities are passed in through the matrix `Aeq` and the right-hand-side vector `beq`.

For example, if you have the nonlinear equality constraint $x_1^2 + x_2 = 1$ and the nonlinear inequality constraint $x_1 x_2 \geq -10$, rewrite them as

$$x_1^2 + x_2 - 1 = 0,$$
$$-x_1 x_2 - 10 \leq 0,$$

and then solve the problem using the following steps.

Step 1: Write a file objfun.m.

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

Step 2: Write a file confuneq.m for the nonlinear constraints.

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

Step 3: Invoke constrained optimization routine.

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

After 21 function evaluations, the solution produced is

```
x,fval
x =
   -0.7529    0.4332
fval =
    1.5093

[c,ceq] = confuneq(x) % Check the constraint values at x

c =
    -9.6739
ceq =
  6.3038e-009
```

Note that `ceq` is equal to 0 within the default tolerance on the constraints of `1.0e-006` and that `c` is less than or equal to 0 as desired.

Back to Top

Example: Nonlinear Minimization with Bound Constraints and Banded Preconditioner

The goal in this problem is to minimize the nonlinear function

$$f(x) = 1 + \sum_{i=1}^{n} \left|(3 - 2x_i)x_i - x_{i-1} - x_{i+1} + 1\right|^p + \sum_{i=1}^{n/2} \left|x_i + x_{i+n/2}\right|^p,$$

such that $-10.0 \le x_i \le 10.0$, where n is 800 (n should be a multiple of 4), $p = 7/3$, and $x_0 = x_{n+1} = 0$.

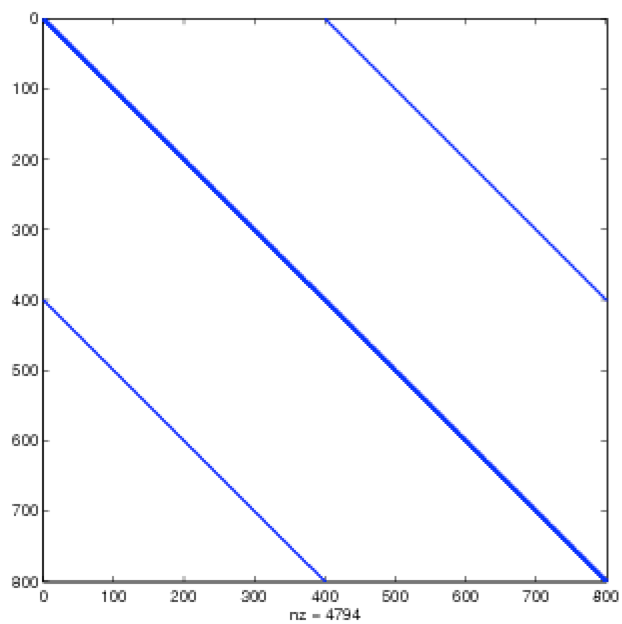
Step 1: Write a file tbroyfg.m that computes the objective function and the gradient of the objective

The `tbroyfg.m` file computes the function value and gradient. This file is long and is not included here. You can see the code for this function using the command

```
type tbroyfg
```

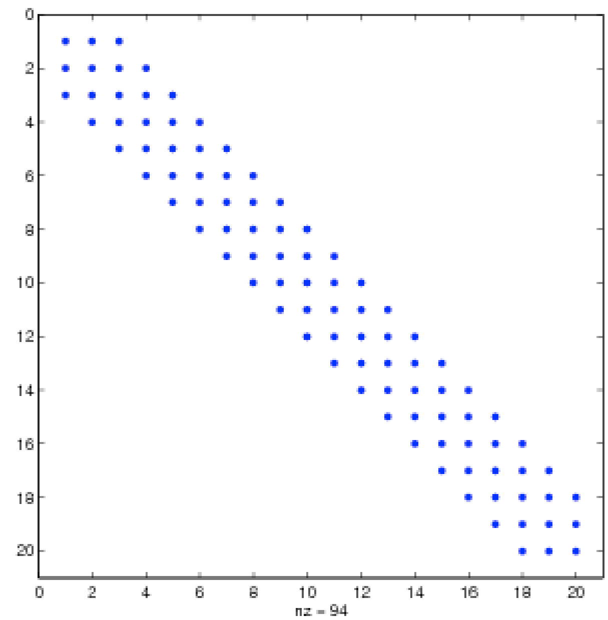
The sparsity pattern of the Hessian matrix has been predetermined and stored in the file `tbroyhstr.mat`. The sparsity structure for the Hessian of this problem is banded, as you can see in the following spy plot.

```
load tbroyhstr
spy(Hstr)
```



In this plot, the center stripe is itself a five-banded matrix. The following plot shows the matrix more clearly:

```
spy(Hstr(1:20,1:20))
```



Use optimset to set the HessPattern parameter to Hstr. When a problem as large as this has obvious sparsity structure, not setting the HessPattern parameter requires a huge amount of unnecessary memory and computation. This is because fmincon attempts to use finite differencing on a full Hessian matrix of 640,000 nonzero entries.

You must also set the GradObj parameter to 'on' using optimset, since the gradient is computed in tbroyfg.m. Then execute fmincon as shown in Step 2.

Step 2: Call a nonlinear minimization routine with a starting point xstart.

```
fun = @tbroyfg;
load tbroyhstr            % Get Hstr, structure of the Hessian
n = 800;
xstart = -ones(n,1); xstart(2:2:n) = 1;
lb = -10*ones(n,1); ub = -lb;
options = optimset('GradObj','on','HessPattern',Hstr);

[x,fval,exitflag,output] = ...
   fmincon(fun,xstart,[],[],[],[],lb,ub,[],options);
```

After seven iterations, the exitflag, fval, and output values are

```
exitflag =
     3

fval =
  270.4790

output =
         iterations: 7
          funcCount: 8
       cgiterations: 18
      firstorderopt: 0.0163
          algorithm: 'trust-region-reflective'
```

```
              message: [1x471 char]
       constrviolation: 0
```

For bound constrained problems, the first-order optimality is the infinity norm of `v.*g`, where `v` is defined as in [Box Constraints](), and `g` is the gradient.

Because of the five-banded center stripe, you can improve the solution by using a five-banded preconditioner instead of the default diagonal preconditioner. Using the [`optimset`]() function, reset the `PrecondBandWidth` parameter to `2` and solve the problem again. (The bandwidth is the number of upper (or lower) diagonals, not counting the main diagonal.)

```
fun = @tbroyfg;
load tbroyhstr          % Get Hstr, structure of the Hessian
n = 800;
xstart = -ones(n,1); xstart(2:2:n,1) = 1;
lb = -10*ones(n,1); ub = -lb;
options = optimset('GradObj','on','HessPattern',Hstr, ...
                   'PrecondBandWidth',2);
[x,fval,exitflag,output] = ...
   fmincon(fun,xstart,[],[],[],[],lb,ub,[],options);
```

The number of iterations actually goes up by two; however the total number of CG iterations drops from 18 to 15. The first-order optimality measure is reduced by a factor of `1e-3`:

```
exitflag =
     3

fval =
  270.4790

output =
         iterations: 9
          funcCount: 10
        cgiterations: 15
      firstorderopt: 7.5340e-005
          algorithm: 'trust-region-reflective'
            message: [1x471 char]
     constrviolation: 0
```

[Back to Top]()

Example: Nonlinear Minimization with Equality Constraints

The trust-region reflective method for [`fmincon`]() can handle equality constraints if no other constraints exist. Suppose you want to minimize the same objective as in [Equation 6-17](), which is coded in the function `brownfgh.m`, where `n = 1000`, such that $Aeq \cdot x = beq$ for *Aeq* that has 100 equations (so *Aeq* is a 100-by-1000 matrix).

Step 1: Write a file brownfgh.m that computes the objective function, the gradient of the objective, and the sparse tridiagonal Hessian matrix.

The file is lengthy so is not included here. View the code with the command

```
type brownfgh
```

Because `brownfgh` computes the gradient and Hessian values as well as the objective function, you need to use [`optimset`]() to indicate that this information is available in `brownfgh`, using the `GradObj` and

`Hessian` options.

The sparse matrix `Aeq` and vector `beq` are available in the file `browneq.mat`:

```
load browneq
```

The linear constraint system is 100-by-1000, has unstructured sparsity (use [spy](Aeq) `(Aeq)` to view the sparsity structure), and is not too badly ill-conditioned:

```
condest(Aeq*Aeq')
ans =
  2.9310e+006
```

Step 2: Call a nonlinear minimization routine with a starting point xstart.

```
fun = @brownfgh;
load browneq          % Get Aeq and beq, the linear equalities
n = 1000;
xstart = -ones(n,1); xstart(2:2:n) = 1;
options = optimset('GradObj','on','Hessian','on');
[x,fval,exitflag,output] = ...
   fmincon(fun,xstart,[],[],Aeq,beq,[],[],[],options);
```

fmincon prints the following exit message:

```
Local minimum possible.

fmincon stopped because the final change in function value relative to
its initial value is less than the default value of the function tolerance.
```

The `exitflag` value of `3` also indicates that the algorithm terminated because the change in the objective function value was less than the tolerance `TolFun`. The final function value is given by `fval`.

```
exitflag,fval,output

exitflag =
     3

fval =
  205.9313

output =
         iterations: 22
          funcCount: 23
       cgiterations: 30
      firstorderopt: 0.0027
          algorithm: 'trust-region-reflective'
            message: [1x471 char]
    constrviolation: 2.2249e-013
```

The linear equalities are satisfied at `x`.

```
norm(Aeq*x-beq)

ans =
  1.1957e-012
```

[Back to Top](#)

Example: Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints

The `fmincon` interior-point and trust-region reflective algorithms, and the `fminunc` large-scale algorithm can solve problems where the Hessian is dense but structured. For these problems, `fmincon` and `fminunc` do not compute $H*Y$ with the Hessian H directly, because forming H would be memory-intensive. Instead, you must provide `fmincon` or `fminunc` with a function that, given a matrix Y and information about H, computes $W = H*Y$.

In this example, the objective function is nonlinear and linear equalities exist so `fmincon` is used. The description applies to the trust-region reflective algorithm; the `fminunc` large-scale algorithm is similar. For the interior-point algorithm, see the `'HessMult'` option in [Hessian](#). The objective function has the structure

$$f(x) = \hat{f}(x) - \frac{1}{2}x^T V V^T x,$$

where V is a 1000-by-2 matrix. The Hessian of f is dense, but the Hessian of $\hat{f}$ is sparse. If the Hessian of $\hat{f}$ is $\hat{H}$, then H, the Hessian of f, is

$$H = \hat{H} - VV^T.$$

To avoid excessive memory usage that could happen by working with H directly, the example provides a Hessian multiply function, `hmfleq1`. This function, when passed a matrix `Y`, uses sparse matrices `Hinfo`, which corresponds to $\hat{H}$, and `V` to compute the Hessian matrix product

```
W = H*Y = (Hinfo - V*V')*Y
```

In this example, the Hessian multiply function needs $\hat{H}$ and `V` to compute the Hessian matrix product. `V` is a constant, so you can capture `V` in a function handle to an anonymous function.

However, $\hat{H}$ is not a constant and must be computed at the current `x`. You can do this by computing $\hat{H}$ in the objective function and returning $\hat{H}$ as `Hinfo` in the third output argument. By using `optimset` to set the `'Hessian'` options to `'on'`, `fmincon` knows to get the `Hinfo` value from the objective function and pass it to the Hessian multiply function `hmfleq1`.

Step 1: Write a file brownvv.m that computes the objective function, the gradient, and the sparse part of the Hessian.

The example passes `brownvv` to `fmincon` as the objective function. The [`brownvv.m`](#) file is long and is not included here. You can view the code with the command

```
type brownvv
```

Because `brownvv` computes the gradient and part of the Hessian as well as the objective function, the example ([Step 3](#)) uses [`optimset`](#) to set the `GradObj` and `Hessian` options to `'on'`.

Step 2: Write a function to compute Hessian-matrix products for H given a matrix Y.

Now, define a function `hmfleq1` that uses `Hinfo`, which is computed in `brownvv`, and `V`, which you can capture in a function handle to an anonymous function, to compute the Hessian matrix product `W` where `W = H*Y = (Hinfo - V*V')*Y`. This function must have the form

```
W = hmfleq1(Hinfo,Y)
```

The first argument must be the same as the third argument returned by the objective function `brownvv`. The second argument to the Hessian multiply function is the matrix `Y` (of `W = H*Y`).

Because `fmincon` expects the second argument `Y` to be used to form the Hessian matrix product, `Y` is always a matrix with `n` rows where `n` is the number of dimensions in the problem. The number of columns in `Y` can vary. Finally, you can use a function handle to an anonymous function to capture V, so V can be the third argument to `'hmfleqq'`.

```
function W = hmfleq1(Hinfo,Y,V);
%HMFLEQ1 Hessian-matrix product function for BROWNVV objective.
%   W = hmfleq1(Hinfo,Y,V) computes W = (Hinfo-V*V')*Y
%   where Hinfo is a sparse matrix computed by BROWNVV
%   and V is a 2 column matrix.
W = Hinfo*Y - V*(V'*Y);
```

Note The function `hmfleq1` is available in the `optimdemos` folder as `hmfleq1.m`.

Step 3: Call a nonlinear minimization routine with a starting point and linear equality constraints.

Load the problem parameter, `V`, and the sparse equality constraint matrices, `Aeq` and `beq`, from `fleq1.mat`, which is available in the `optimdemos` folder. Use `optimset` to set the `GradObj` and `Hessian` options to `'on'` and to set the `HessMult` option to a function handle that points to `hmfleq1`. Call `fmincon` with objective function `brownvv` and with `V` as an additional parameter:

```
function [fval, exitflag, output, x] = runfleq1
% RUNFLEQ1 demonstrates 'HessMult' option for
% FMINCON with linear equalities.

%   Copyright 1984-2006 The MathWorks, Inc.
%   $Revision: 1.1.6.29.2.1 $  $Date: 2011/12/02 18:22:02 $

problem = load('fleq1');   % Get V, Aeq, beq
V = problem.V; Aeq = problem.Aeq; beq = problem.beq;
n = 1000;              % problem dimension
xstart = -ones(n,1); xstart(2:2:n,1) = ones(length(2:2:n),1);
% starting point
options = optimset('GradObj','on','Hessian','on','HessMult',...
@(Hinfo,Y)hmfleq1(Hinfo,Y,V) ,'Display','iter','TolFun',1e-9);
[x,fval,exitflag,output] = fmincon(@(x)brownvv(x,V),...
xstart,[],[],Aeq,beq,[],[], [],options);
```

To run the preceding code, enter

```
[fval,exitflag,output,x] = runfleq1;
```

Because the iterative display was set using `optimset`, this command generates the following iterative display:

```
                                 Norm of      First-order
 Iteration        f(x)           step          optimality   CG-iterations
     0            1997.07                          916
     1            1072.57         6.31716          465            1
     2            480.247         8.19711          201            2
     3            136.982         10.3039         78.1            2
```

```
   4          44.416        9.04685         16.7          2
   5          44.416            100         16.7          2
   6          44.416             25         16.7          0
   7        -9.05631           6.25         52.9          0
   8        -317.437           12.5         91.7          1
   9        -405.381           12.5    1.11e+003          1
  10        -451.161          3.125          327          4
  11        -482.688        0.78125          303          5
  12        -547.427         1.5625          187          5
  13         -610.42         1.5625          251          7
  14        -711.522         1.5625          143          3
  15         -802.98          3.125          165          3
  16        -820.431        1.13329         32.9          3
  17        -822.996       0.492813         7.61          2
  18        -823.236       0.223154         1.68          3
  19        -823.245       0.056205        0.529          3
  20        -823.246      0.0150139       0.0342          5
  21        -823.246     0.00479085       0.0152          7
  22        -823.246     0.00353697      0.00828          9
  23        -823.246    0.000884242        0.005          9
  24        -823.246      0.0012715      0.00125          9
  25        -823.246    0.000317876       0.0025          9

Local minimum possible.

fmincon stopped because the final change in function value relative to
its initial value is less than the selected value of the function tolerance.
```

Convergence is rapid for a problem of this size with the PCG iteration cost increasing modestly as the optimization progresses. Feasibility of the equality constraints is maintained at the solution.

```
problem = load('fleq1');   % Get V, Aeq, beq
V = problem.V; Aeq = problem.Aeq; beq = problem.beq;
norm(Aeq*x-beq,inf)

ans =
  2.4869e-014
```

Preconditioning

In this example, `fmincon` cannot use `H` to compute a preconditioner because `H` only exists implicitly. Instead of `H`, `fmincon` uses `Hinfo`, the third argument returned by `brownvv`, to compute a preconditioner. `Hinfo` is a good choice because it is the same size as `H` and approximates `H` to some degree. If `Hinfo` were not the same size as `H`, `fmincon` would compute a preconditioner based on some diagonal scaling matrices determined from the algorithm. Typically, this would not perform as well.

Back to Top

Example: Using Symbolic Math Toolbox Functions to Calculate Gradients and Hessians

If you have a license for Symbolic Math Toolbox functions, you can use these functions to calculate analytic gradients and Hessians for objective and constraint functions. There are two relevant Symbolic Math Toolbox functions:

- `jacobian` generates the gradient of a scalar function, and generates a matrix of the partial derivatives of a vector function. So, for example, you can obtain the Hessian matrix, the second derivatives of the objective function, by applying `jacobian` to the gradient. Part of this example shows how to use `jacobian` to generate symbolic gradients and Hessians of objective and constraint functions.

- `matlabFunction` generates either an anonymous function or a file that calculates the values of a symbolic expression. This example shows how to use `matlabFunction` to generate files that evaluate the objective and constraint function and their derivatives at arbitrary points.

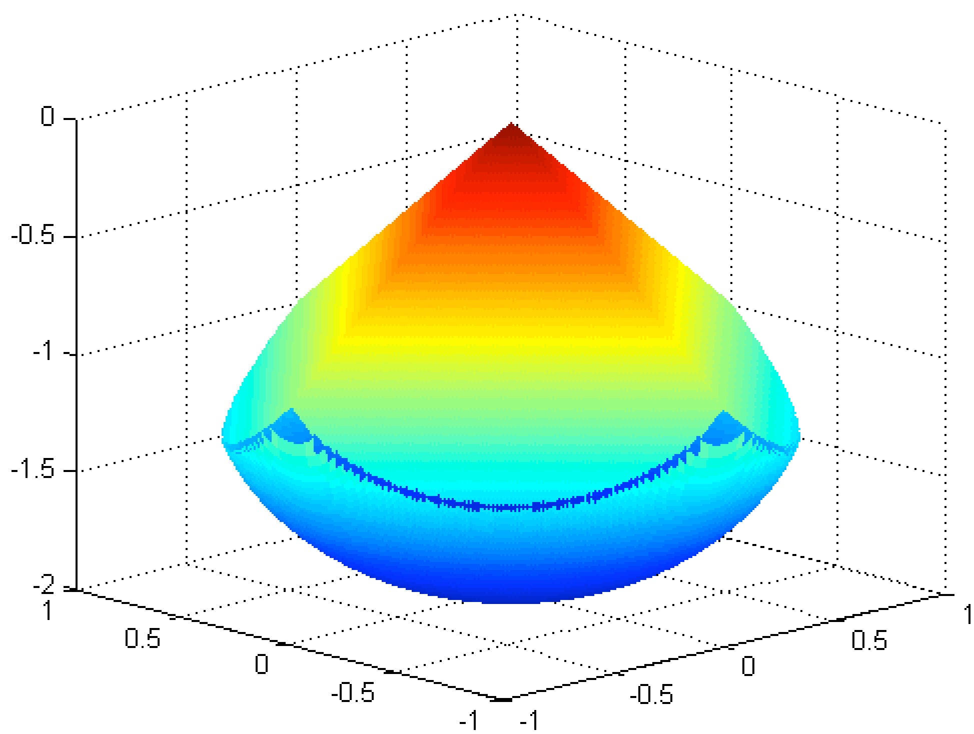
Consider the electrostatics problem of placing 10 electrons in a conducting body. The electrons will arrange themselves so as to minimize their total potential energy, subject to the constraint of lying inside the body. It is well known that all the electrons will be on the boundary of the body at a minimum. The electrons are indistinguishable, so there is no unique minimum for this problem (permuting the electrons in one solution gives another valid solution). This example was inspired by Dolan, Moré, and Munson [58].

This example is a conducting body defined by the following inequalities:

$$z \le -|x| - |y| \quad \textbf{(6-61)}$$

$$x^2 + y^2 + (z+1)^2 \le 1. \quad \textbf{(6-62)}$$

This body looks like a pyramid on a sphere.



Code for Generating the Figure

There is a slight gap between the upper and lower surfaces of the figure. This is an artifact of the general plotting routine used to create the figure. This routine erases any rectangular patch on one surface that touches the other surface.

The syntax and structures of the two sets of toolbox functions differ. In particular, symbolic variables are real or complex scalars, but Optimization Toolbox functions pass vector arguments. So there are several steps to take to generate symbolically the objective function, constraints, and all their requisite derivatives, in a form suitable for the interior-point algorithm of `fmincon`:

1. Create the Variables
2. Include the Linear Constraints
3. Create the Nonlinear Constraints, Their Gradients and Hessians
4. Create the Objective Function, Its Gradient and Hessian
5. Create the Objective Function File
6. Create the Constraint Function File
7. Generate the Hessian Files
8. Run the Optimization
9. Clear the Symbolic Variable Assumptions

To see the efficiency in using gradients and Hessians, see Compare to Optimization Without Gradients and Hessians.

Create the Variables

Generate a symbolic vector `x` as a 30-by-1 vector composed of real symbolic variables `xij`, `i` between 1 and 10, and `j` between 1 and 3. These variables represent the three coordinates of electron `i`: `xi1` corresponds to the *x* coordinate, `xi2` corresponds to the *y* coordinate, and `xi3` corresponds to the *z* coordinate.

```
x = cell(3, 10);
for i = 1:10
    for j = 1:3
        x{j,i} = sprintf('x%d%d',i,j);
    end
end
x = x(:); % now x is a 30-by-1 vector
x = sym(x, 'real');
```

The vector `x` is:

```
x
x =
  x11
  x12
  x13
  x21
  x22
  x23
  x31
  x32
  x33
  x41
  x42
  x43
  x51
  x52
  x53
  x61
  x62
  x63
  x71
  x72
  x73
  x81
  x82
```

```
 x83
 x91
 x92
 x93
x101
x102
x103
```

Include the Linear Constraints

Write the linear constraints in Equation 6-61,

$z \le -|x| - |y|$,

as a set of four linear inequalities for each electron:

xi3 - xi1 - xi2 ≤ 0
xi3 - xi1 + xi2 ≤ 0
xi3 + xi1 - xi2 ≤ 0
xi3 + xi1 + xi2 ≤ 0

Therefore there are a total of 40 linear inequalities for this problem.

Write the inequalities in a structured way:

```
B = [1,1,1;-1,1,1;1,-1,1;-1,-1,1];

A = zeros(40,30);
for i=1:10
    A(4*i-3:4*i,3*i-2:3*i) = B;
end

b = zeros(40,1);
```

You can see that `A*x ≤ b` represents the inequalities:

```
A*x

ans =
    x11 + x12 + x13
    x12 - x11 + x13
    x11 - x12 + x13
    x13 - x12 - x11
    x21 + x22 + x23
    x22 - x21 + x23
    x21 - x22 + x23
    x23 - x22 - x21
    x31 + x32 + x33
    x32 - x31 + x33
    x31 - x32 + x33
    x33 - x32 - x31
    x41 + x42 + x43
    x42 - x41 + x43
    x41 - x42 + x43
    x43 - x42 - x41
    x51 + x52 + x53
    x52 - x51 + x53
    x51 - x52 + x53
    x53 - x52 - x51
```

```
    x61 + x62 + x63
    x62 - x61 + x63
    x61 - x62 + x63
    x63 - x62 - x61
    x71 + x72 + x73
    x72 - x71 + x73
    x71 - x72 + x73
    x73 - x72 - x71
    x81 + x82 + x83
    x82 - x81 + x83
    x81 - x82 + x83
    x83 - x82 - x81
    x91 + x92 + x93
    x92 - x91 + x93
    x91 - x92 + x93
    x93 - x92 - x91
 x101 + x102 + x103
 x102 - x101 + x103
 x101 - x102 + x103
 x103 - x102 - x101
```

Create the Nonlinear Constraints, Their Gradients and Hessians

The nonlinear constraints in Equation 6-62 ,

$$x^2 + y^2 + (z+1)^2 \le 1,$$

are also structured. Generate the constraints, their gradients, and Hessians as follows:

```
c = sym(zeros(1,10));
i = 1:10;
c = (x(3*i-2).^2 + x(3*i-1).^2 + (x(3*i)+1).^2 - 1).';

gradc = jacobian(c,x).'; % .' performs transpose

hessc = cell(1, 10);
for i = 1:10
    hessc{i} = jacobian(gradc(:,i),x);
end
```

The constraint vector `c` is a row vector, and the gradient of `c(i)` is represented in the `i`th column of the matrix `gradc`. This is the correct form, as described in Nonlinear Constraints.

The Hessian matrices, `hessc{1}`...`hessc{10}`, are square and symmetric. It is better to store them in a cell array, as is done here, than in separate variables such as `hessc1, ..., hesssc10`.

Use the `.'` syntax to transpose. The `'` syntax means conjugate transpose, which has different symbolic derivatives.

Create the Objective Function, Its Gradient and Hessian

The objective function, potential energy, is the sum of the inverses of the distances between each electron pair:

$$\text{energy} = \sum_{i<j} \frac{1}{|x_i - x_j|}.$$

The distance is the square root of the sum of the squares of the differences in the components of the vectors.

Calculate the energy, its gradient, and its Hessian as follows:

```
energy = sym(0);
for i = 1:3:25
    for j = i+3:3:28
        dist = x(i:i+2) - x(j:j+2);
        energy = energy + 1/sqrt(dist.'*dist);
    end
end

gradenergy = jacobian(energy,x).';

hessenergy = jacobian(gradenergy,x);
```

Create the Objective Function File

The objective function should have two outputs, `energy` and `gradenergy`. Put both functions in one vector when calling `matlabFunction` to reduce the number of subexpressions that `matlabFunction` generates, and to return the gradient only when the calling function (`fmincon` in this case) requests both outputs. This example shows placing the resulting files in your current folder. Of course, you can place them anywhere you like, as long as the folder is on the MATLAB path.

```
currdir = [pwd filesep]; % You may need to use currdir = pwd
filename = [currdir,'demoenergy.m'];
matlabFunction(energy,gradenergy,'file',filename,'vars',{x});
```

This syntax causes `matlabFunction` to return `energy` as the first output, and `gradenergy` as the second. It also takes a single input vector `{x}` instead of a list of inputs `x11`, ..., `x103`.

The resulting file `demoenergy.m` contains, in part, the following lines or similar ones:

```
function [energy,gradenergy] = demoenergy(in1)
%DEMOENERGY
%    [ENERGY,GRADENERGY] = DEMOENERGY(IN1)
...
x101 = in1(28,:);
...
energy = 1./t140.^(1./2) + ...;
if nargout > 1
    ...
    gradenergy = [(t174.*(t185 - 2.*x11))./2 - ...];
end
```

This function has the correct form for an objective function with a gradient; see Writing Scalar Objective Functions.

Create the Constraint Function File

Generate the nonlinear constraint function, and put it in the correct format.

```
filename = [currdir,'democonstr.m'];
matlabFunction(c,[],gradc,[],'file',filename,'vars',{x},...
    'outputs',{'c','ceq','gradc','gradceq'});
```

The resulting file `democonstr.m` contains, in part, the following lines or similar ones:

```
function [c,ceq,gradc,gradceq] = democonstr(in1)
```

```
%DEMOCONSTR
%    [C,CEQ,GRADC,GRADCEQ] = DEMOCONSTR(IN1)
...
x101 = in1(28,:);
...
c = [t417.^2 + ...];
if nargout > 1
    ceq = [];
end
if nargout > 2
    gradc = [2.*x11,...];
end
if nargout > 3
    gradceq = [];
end
```

This function has the correct form for a constraint function with a gradient; see Nonlinear Constraints.

Generate the Hessian Files

To generate the Hessian of the Lagrangian for the problem, first generate files for the energy Hessian and for the constraint Hessians.

The Hessian of the objective function, `hessenergy`, is a very large symbolic expression, containing over 150,000 symbols, as evaluating `size(char(hessenergy))` shows. So it takes a substantial amount of time to run `matlabFunction(hessenergy)`.

To generate a file `hessenergy.m`, run the following two lines:

```
filename = [currdir,'hessenergy.m'];
matlabFunction(hessenergy,'file',filename,'vars',{x});
```

In contrast, the Hessians of the constraint functions are small, and fast to compute:

```
for i = 1:10
    ii = num2str(i);
    thename = ['hessc',ii];
    filename = [currdir,thename,'.m'];
    matlabFunction(hessc{i},'file',filename,'vars',{x});
end
```

After generating all the files for the objective and constraints, put them together with the appropriate Lagrange multipliers in a file `hessfinal.m` as follows:

```
function H = hessfinal(X,lambda)
%
% Call the function hessenergy to start
H = hessenergy(X);

% Add the Lagrange multipliers * the constraint Hessians
H = H + hessc1(X) * lambda.ineqnonlin(1);
H = H + hessc2(X) * lambda.ineqnonlin(2);
H = H + hessc3(X) * lambda.ineqnonlin(3);
H = H + hessc4(X) * lambda.ineqnonlin(4);
H = H + hessc5(X) * lambda.ineqnonlin(5);
H = H + hessc6(X) * lambda.ineqnonlin(6);
H = H + hessc7(X) * lambda.ineqnonlin(7);
H = H + hessc8(X) * lambda.ineqnonlin(8);
```

```
H = H + hessc9(X) * lambda.ineqnonlin(9);
H = H + hessc10(X) * lambda.ineqnonlin(10);
```

Run the Optimization

Start the optimization with the electrons distributed randomly on a sphere of radius 1/2 centered at [0,0,-1]:

```
rng('default'); % for reproducibility
Xinitial = randn(3,10); % columns are normal 3-D vectors
for j=1:10
    Xinitial(:,j) = Xinitial(:,j)/norm(Xinitial(:,j))/2;
    % this normalizes to a 1/2-sphere
end
Xinitial(3,:) = Xinitial(3,:) - 1; % center at [0,0,-1]
Xinitial = Xinitial(:); % Convert to a column vector
```

Set the options to use the interior-point algorithm, and to use gradients and the Hessian:

```
options = optimset('Algorithm','interior-point','GradObj','on',...
    'GradConstr','on','Hessian','user-supplied',...
    'HessFcn',@hessfinal,'Display','final');
```

Call `fmincon`:

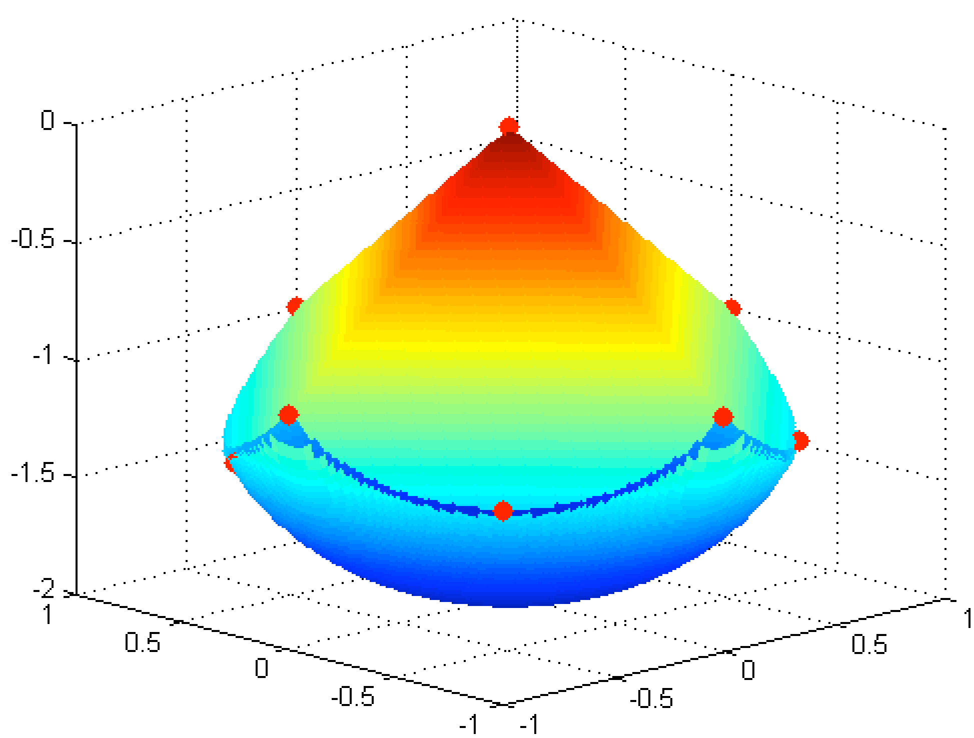
```
[xfinal fval exitflag output] = fmincon(@demoenergy,Xinitial,...
    A,b,[],[],[],[],@democonstr,options)
```

The output is as follows:

```
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the default value of the function tolerance,
and constraints are satisfied to within the default value of the constraint tolera

xfinal =
   -0.0317
    0.0317
   -1.9990
    0.6356
   -0.6356
   -1.4381
    0.0000
   -0.0000
   -0.0000
    0.0000
   -1.0000
   -1.0000
    1.0000
   -0.0000
   -1.0000
   -1.0000
   -0.0000
   -1.0000
    0.6689
    0.6644
   -1.3333
```

```
   -0.6667
    0.6667
   -1.3333
    0.0000
    1.0000
   -1.0000
   -0.6644
   -0.6689
   -1.3333

fval =
   34.1365

exitflag =
     1

output =
         iterations: 19
          funcCount: 28
    constrviolation: 0
           stepsize: 4.0372e-005
          algorithm: 'interior-point'
      firstorderopt: 4.0015e-007
       cgiterations: 55
            message: [1x777 char]
```

Even though the initial positions of the electrons were random, the final positions are nearly symmetric:



Code for Generating the Figure

Compare to Optimization Without Gradients and Hessians

The use of gradients and Hessians makes the optimization run faster and more accurately. To compare with the same optimization using no gradient or Hessian information, set the options not to use gradients and Hessians:

```
options = optimset('Algorithm','interior-point',...
    'Display','final');
[xfinal2 fval2 exitflag2 output2] = fmincon(@demoenergy,Xinitial,...
    A,b,[],[],[],[],@democonstr,options)
```

The output shows that `fmincon` found an equivalent minimum, but took more iterations and many more function evaluations to do so:

```
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the default value of the function tolerance,
and constraints are satisfied to within the default value of the constraint tolera

xfinal2 =
    0.0000
    1.0000
   -1.0000
    0.6690
   -0.6644
   -1.3333
   -0.6644
    0.6690
   -1.3333
    0.0000
   -1.0000
   -1.0000
    0.6356
    0.6356
   -1.4381
   -0.0317
   -0.0317
   -1.9990
    1.0000
    0.0000
   -1.0000
   -1.0000
    0.0000
   -1.0000
    0.0000
    0.0000
   -0.0000
   -0.6667
   -0.6667
   -1.3333

fval2 =
   34.1365

exitflag2 =
     1
```

```
output2 =
         iterations: 87
          funcCount: 2745
    constrviolation: 0
           stepsize: 1.4494e-06
          algorithm: 'interior-point'
      firstorderopt: 2.9480e-06
       cgiterations: 0
            message: [1x777 char]
```

In this run the number of function evaluations (in `output2.funcCount`) is 2745, compared to 28 (in `output.funcCount`) when using gradients and Hessian.

Clear the Symbolic Variable Assumptions

The symbolic variables in this example have the assumption, in the symbolic engine workspace, that they are real. To clear this assumption from the symbolic engine workspace, it is not sufficient to delete the variables. You must clear the variables using the syntax

```
syms x11 x12 x13 clear
```

or reset the symbolic engine using the command

```
reset(symengine)
```

After resetting the symbolic engine you should clear all symbolic variables from the MATLAB workspace with the `clear` command, or `clear` *`variable_list`*.

Back to Top

Example: One-Dimensional Semi-Infinite Constraints

Find values of *x* that minimize

$f(x) = (x_1 - 0.5)^2 + (x_2 - 0.5)^2 + (x_3 - 0.5)^2$

where

$$K_1(x,w_1) = \sin(w_1 x_1)\cos(w_1 x_2) - \frac{1}{1000}(w_1 - 50)^2 - \sin(w_1 x_3) - x_3 \le 1,$$
$$K_2(x,w_2) = \sin(w_2 x_2)\cos(w_2 x_1) - \frac{1}{1000}(w_2 - 50)^2 - \sin(w_2 x_3) - x_3 \le 1,$$

for all values of w_1 and w_2 over the ranges

$1 \le w_1 \le 100,$
$1 \le w_2 \le 100.$

Note that the semi-infinite constraints are one-dimensional, that is, vectors. Because the constraints must be in the form $K_i(x,w_i) \le 0$, you need to compute the constraints as

$$K_1(x,w_1) = \sin(w_1 x_1)\cos(w_1 x_2) - \frac{1}{1000}(w_1 - 50)^2 - \sin(w_1 x_3) - x_3 - 1 \le 0,$$
$$K_2(x,w_2) = \sin(w_2 x_2)\cos(w_2 x_1) - \frac{1}{1000}(w_2 - 50)^2 - \sin(w_2 x_3) - x_3 - 1 \le 0.$$

First, write a file that computes the objective function.

```
function f = myfun(x,s)
% Objective function
```

```
f = sum((x-0.5).^2);
```

Second, write a file `mycon.m` that computes the nonlinear equality and inequality constraints and the semi-infinite constraints.

```
function [c,ceq,K1,K2,s] = mycon(X,s)
% Initial sampling interval
if isnan(s(1,1)),
   s = [0.2 0; 0.2 0];
end
% Sample set
w1 = 1:s(1,1):100;
w2 = 1:s(2,1):100;

% Semi-infinite constraints
K1 = sin(w1*X(1)).*cos(w1*X(2)) - 1/1000*(w1-50).^2 -...
       sin(w1*X(3))-X(3)-1;
K2 = sin(w2*X(2)).*cos(w2*X(1)) - 1/1000*(w2-50).^2 -...
       sin(w2*X(3))-X(3)-1;

% No finite nonlinear constraints
c = []; ceq=[];

% Plot a graph of semi-infinite constraints
plot(w1,K1,'-',w2,K2,':')
title('Semi-infinite constraints')
drawnow
```

Then, invoke an optimization routine.

```
x0 = [0.5; 0.2; 0.3];      % Starting guess
[x,fval] = fseminf(@myfun,x0,2,@mycon);
```

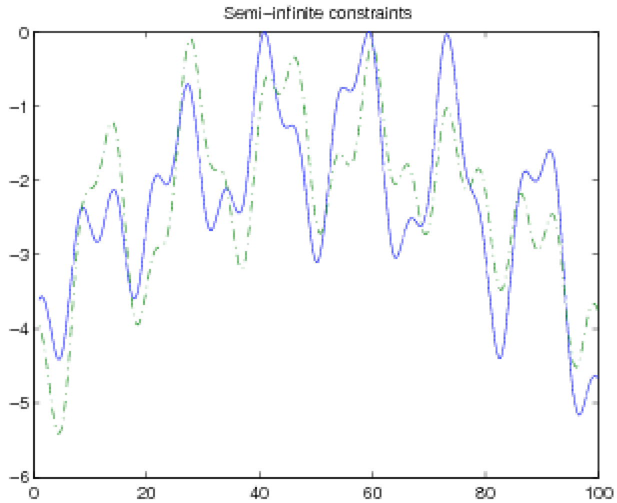
After eight iterations, the solution is

```
x
x =
    0.6675
    0.3012
    0.4022
```

The function value and the maximum values of the semi-infinite constraints at the solution `x` are

```
fval
fval =
    0.0771

[c,ceq,K1,K2] = mycon(x,NaN); % Initial sampling interval
max(K1)
ans =
   -0.0077
max(K2)
ans =
   -0.0812
```

A plot of the semi-infinite constraints is produced.



This plot shows how peaks in both constraints are on the constraint boundary.

The plot command inside `mycon.m` slows down the computation. Remove this line to improve the speed.

Back to Top

Example: Two-Dimensional Semi-Infinite Constraint

Find values of x that minimize

$$f(x) = (x_1 - 0.2)^2 + (x_2 - 0.2)^2 + (x_3 - 0.2)^2,$$

where

$$K_1(x,w) = \sin(w_1 x_1)\cos(w_2 x_2) - \frac{1}{1000}(w_1 - 50)^2 - \sin(w_1 x_3) - x_3 + \ldots$$
$$\sin(w_2 x_2)\cos(w_1 x_1) - \frac{1}{1000}(w_2 - 50)^2 - \sin(w_2 x_3) - x_3 \le 1.5,$$

for all values of w_1 and w_2 over the ranges

$$1 \le w_1 \le 100,$$
$$1 \le w_2 \le 100,$$

starting at the point $x = [0.25,0.25,0.25]$.

Note that the semi-infinite constraint is two-dimensional, that is, a matrix.

First, write a file that computes the objective function.

```
function f = myfun(x,s)
% Objective function
f = sum((x-0.2).^2);
```

Second, write a file for the constraints, called `mycon.m`. Include code to draw the surface plot of the semi-infinite constraint each time `mycon` is called. This enables you to see how the constraint changes as `X` is being minimized.

```
function [c,ceq,K1,s] = mycon(X,s)
% Initial sampling interval
if isnan(s(1,1)),
   s = [2 2];
```

```
end

% Sampling set
w1x = 1:s(1,1):100;
w1y = 1:s(1,2):100;
[wx,wy] = meshgrid(w1x,w1y);

% Semi-infinite constraint
K1 = sin(wx*X(1)).*cos(wx*X(2))-1/1000*(wx-50).^2 -...
       sin(wx*X(3))-X(3)+sin(wy*X(2)).*cos(wx*X(1))-...
       1/1000*(wy-50).^2-sin(wy*X(3))-X(3)-1.5;

% No finite nonlinear constraints
c = []; ceq=[];

% Mesh plot
m = surf(wx,wy,K1,'edgecolor','none','facecolor','interp');
camlight headlight
title('Semi-infinite constraint')
drawnow
```

Next, invoke an optimization routine.

```
x0 = [0.25, 0.25, 0.25];    % Starting guess
[x,fval] = fseminf(@myfun,x0,1,@mycon)
```

After nine iterations, the solution is

```
x
x =
    0.2522    0.1714    0.1936
```

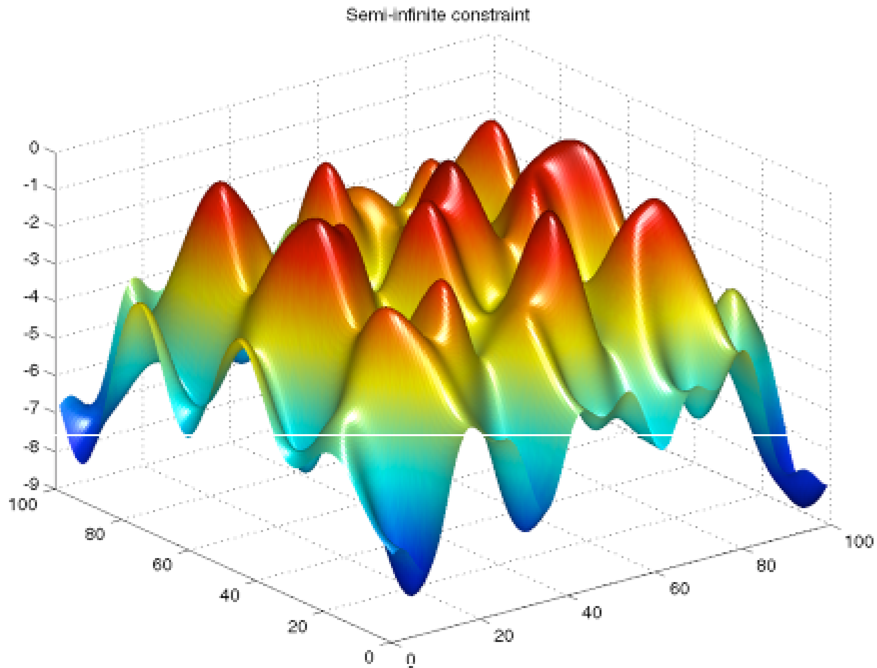
and the function value at the solution is

```
fval
fval =
    0.0036
```

The goal was to minimize the objective $f(x)$ such that the semi-infinite constraint satisfied $K_1(x,w) \leq 1.5$. Evaluating `mycon` at the solution `x` and looking at the maximum element of the matrix `K1` shows the constraint is easily satisfied.

```
[c,ceq,K1] = mycon(x,[0.5,0.5]);  % Sampling interval 0.5
max(max(K1))

ans =
   -0.0333
```

This call to `mycon` produces the following surf plot, which shows the semi-infinite constraint at `x`.



Back to Top

Was this topic helpful? Yes No

Constrained Nonlinear Optimization Algorithms Linear Programming Algorithms