



Optimization Using Symbolic Derivatives

By Alan Weiss

Most Optimization Toolbox™ solvers run faster and more accurately when your objective and constraint function files include derivative calculations. Some solvers also benefit from second derivatives, or Hessians. While calculating a derivative is straightforward, it is also quite tedious and error-prone. Calculating second derivatives is even more tedious and fraught with opportunities for error. How can you get your solver to run faster and more accurately without the pain of computing derivatives manually?

Products Used

- MATLAB®
- Optimization Toolbox™
- Symbolic Math Toolbox™

This article demonstrates how to ease the calculation and use of gradients using Symbolic Math Toolbox™. The techniques described here are applicable to almost any optimization problem where the objective or constraint functions can be defined analytically. This means that you can use them if your objective and constraint functions are not simulations or black-box functions.

Running a Symbolically Defined Optimization

Suppose we want to minimize the function $x + y + \cosh(x - 1.1y) + \sinh(z/4)$ over the region defined by the implicit equation $z^2 = \sin(z - x^2y^2)$, $-1 \leq x \leq 1$, $-1 \leq y \leq 1$, $0 \leq z \leq 1$.

The region is shown in Figure 1.

The `fmincon` solver from Optimization Toolbox solves nonlinear optimization

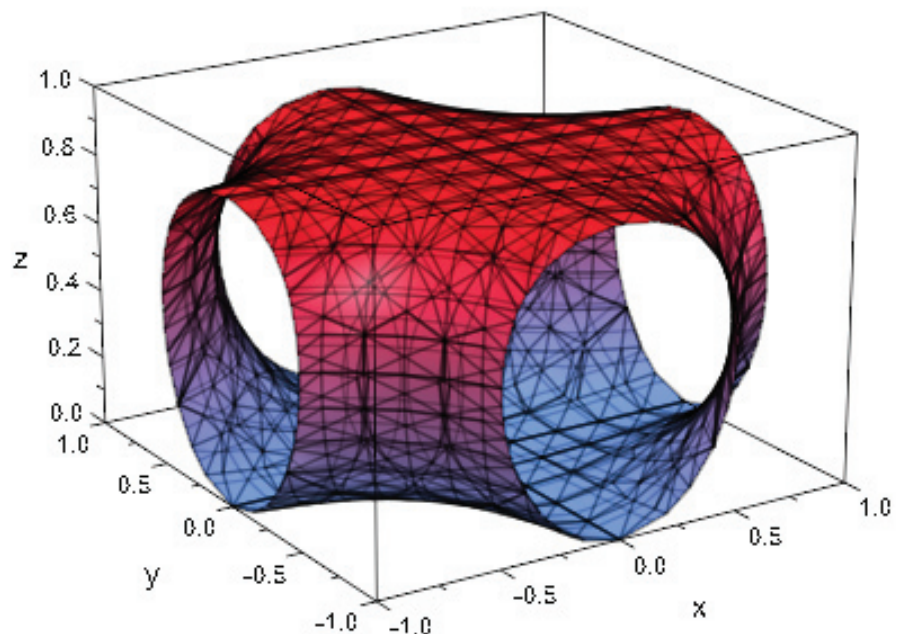


Figure 1. Surface plot created in the Symbolic Math Toolbox note book interface showing the region defined by the implicit equation $z^2 = \sin(z - x^2y^2)$, $-1 \leq x \leq 1$, $-1 \leq y \leq 1$, $0 \leq z \leq 1$.

problems with nonlinear constraints. To formulate our problem for `fmincon`, we first write the objective and constraint functions symbolically (Figure 2). We then generate function handles for numerical computation with `matlabFunction` from Symbolic Math Toolbox.

The returned output structure shows that it took `fmincon` 20 iterations and 99 function evaluations to solve the problem. The solution point `x` (the yellow sphere in the plot in Figure 3) is `[-0.8013;-0.6122;0.4077]`.

Solving the Problem with Gradients

To include derivatives of the objective and constraint functions in the calculation, we simply perform three steps:

1. Compute the derivatives using the Symbolic Math Toolbox `jacobian` function.
2. Generate objective and constraint functions that include the derivatives with `matlabFunction`
3. Set `fmincon` options to use the derivatives.

Figure 4 shows how to include gradients for the example.

Notice that the `jacobian` function is followed by `.'`. This transpose ensures that `gradw` and `gradobj` are column vectors, the preferred orientation for Optimization Toolbox solvers. `matlabFunction` creates a function handle for evaluating both the function and its gradient. Notice, too, that we were able to calculate the gradient of the constraint function even though the function is implicit.

The output structure shows that `fmincon` computed the solution in 20 iterations, just as it did without gradients. `fmincon` with gradients evaluated the nonlinear functions at 36 points, compared to 99 points without gradients.

```
>> syms x y z % define symbolic variables
t = [x;y;z]; % the symbolic vector
w = z^2 - sin(z - x^2*y^2); % the constraint is w = 0
obj = x + y + cosh(x - (11*y)/10) + sinh(z/4); % the objective
objfun = matlabFunction(obj,'vars',{t}); % function handle to objective
% Generate function handle to constraint:
confun = matlabFunction([],w,'vars',{t},'outputs',{'c','ceq'});
opts = optimset('Algorithm','interior-point','Display','off');
[x fval eflag output] = fmincon(objfun,[0;0;1],...
    [],[],[],[-1;-1;0],[1;1;1],confun,opts);
```

Figure 2. Generating and running an optimization problem from symbolic expressions.

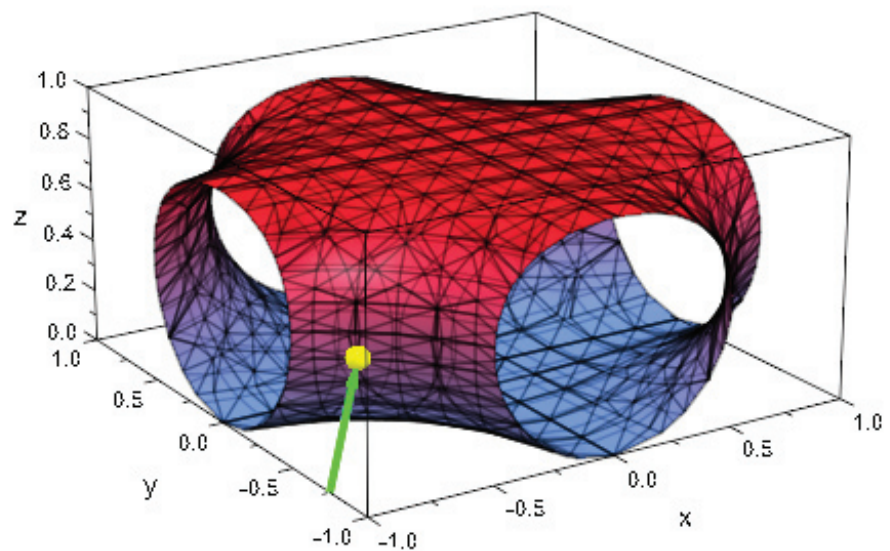


Figure 3. Constraint set and solution point.

```
>> gradw = jacobian(w,t).'; % the gradient as column vector
gradobj = jacobian(obj,t).'; % gradient of objective, column
% Generate function handle to objective
% with gradient as second output of objective function:
objfun = matlabFunction(obj,gradobj,'vars',{t},...
    'outputs',{'f','gradf'});
% Generate function handle to constraints, including gradient:
confun = matlabFunction([],w,[],gradw,'vars',{t},...
    'outputs',{'c','ceq','gradc','gradceq'});
% Set options to use the gradients of objective and constraint:
opts = optimset(opts,'GradObj','on','GradConstr','on');
[x fval eflag output] = fmincon(objfun,[0;0;1],...
    [],[],[],[-1;-1;0],[1;1;1],confun,opts);
```

Figure 4. Using `jacobian` and `matlabFunction` to compute the value and gradient of the objective and constraint functions.

```

>> hessw = jacobian(gradw,t); % Hessian of constraint
hw = matlabFunction(hessw,'vars',{t}); % constraint handle
hessobj = jacobian(gradobj,t); % Hessian of objective
hoh = matlabFunction(hessobj,'vars',{t}); % objective handle
% Hessian of Lagrangian with one equality constraint:
% add hoh and a Lagrange multiplier times hw
hfcn = @(x,lambda)(hoh(x) + lambda.eqnonlin(1)*hw(x));
opts = optimset(opts,'Hessian','user-supplied',...
    'HessFcn',hfcn); % Use the Hessian
[x fval eflag output] = fmincon(objfun,[0;0;1],...
    [],[],[],[-1;-1;0],[1;1;1],confun,opts);

```

Figure 5. Including the Hessian function and running the optimization.

Including the Hessian

A Hessian function lets us solve the problem even more efficiently. For the interior-point algorithm, we write a function that is the Hessian of the Lagrangian. This means that if f is the objective function, c is the vector of nonlinear inequality constraints, ceq is the vector of nonlinear equality constraints, and λ is the vector of associated Lagrange multipliers, the Hessian H is

$$H = \nabla^2 f(x) + \sum_i \lambda_i \nabla^2 c_i(x) + \sum_j \lambda_j \nabla^2 ceq_j(x).$$

$\nabla^2 u$ represents the matrix of second derivatives with respect to x of the function u .

`fmincon` generates the Lagrange multipliers in a MATLAB® structure. The relevant multipliers are `lambda.ineqnonlin` and `lambda.eqnonlin`, corresponding to indices i and j in the equation for H . We include multipliers in the Hessian function (Figure 5), and then run the optimization¹.

The output structure shows that including a Hessian results in fewer iterations (10 instead of 20), a lower function count (11 instead of 36), and a better first-order optimality measure ($2e-8$ instead of $8e-8$). ■

For More Information

- Example: Using Symbolic Math Toolbox Functions to Calculate Gradients and Hessians
www.mathworks.com/symbolic-math-functions
- Demo: Using Symbolic Mathematics with Optimization Toolbox Solvers
www.mathworks.com/optimization-solvers

Resources

VISIT

www.mathworks.com

TECHNICAL SUPPORT

www.mathworks.com/support

ONLINE USER COMMUNITY

www.mathworks.com/matlabcentral

DEMOS

www.mathworks.com/demos

TRAINING SERVICES

www.mathworks.com/training

THIRD-PARTY PRODUCTS AND SERVICES

www.mathworks.com/connections

Worldwide CONTACTS

www.mathworks.com/contact

E-MAIL

info@mathworks.com

© 2010 The MathWorks, Inc. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

91801v00 3/10

¹ For nonlinear equality constraints in Optimization Toolbox version 9b or earlier, you must subtract, not add, the Lagrange multiplier. See <http://www.mathworks.com/support/bugreports/566464>