sqp

Purpose Find the minimum of a constrained nonlinear multivariable function,

$$\min_{x} f(x) \quad \text{such that} \quad G(x) \leq 0$$

where *x* is a vector, *f(x)* is a function that returns a scalar, and *G(x)* is a function that returns a vector. Both *f(x)* and *G(x)* can be nonlinear functions. *G(x)* can define both equality and inequality constraints. Equality constraints precede inequality constraints. Number of equality constraints is set in `options(13)`.

Synopsis

```
x = sqp('fun',x0)
x = sqp('fun',x0,options)
x = sqp('fun',x0,options,vlb,vub,'grad')
x = sqp(@fun, x0,options,vlb,vub,@grad, p1,p2, ...)
x = sqp(problem)
[x,options] = sqp('fun',x0, ... )
[x,options,lambda] = sqp(@fun,x0, ... )
[x,options,lambda,hess,status] = sqp(@fun,x0, ... )
```

Description `sqp` 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*. It was originally written with a function signature based on the *MATLAB Optimization Toolbox Version 1* `constr.m` function, and remains backward compatible. Subsequently, it has been updated to employ function handles (`@fun` and `@grad`) for the user-supplied objective and constraint evaluation functions, as well as to use the `optimset` data structure for the `options` input variable for `fmincon` in place of the `options` from the obsolete `foptions` utility function.

`x = sqp('fun',x0)` starts at the point `x0` and finds a minimum of the function and constraints defined in the m-file named `fun.m`.

`x = sqp('fun',x0,options)` uses the parameter values in the vector options rather than the default option values.

`x = sqp('fun',x,options,vlb,vub)` defines a set of lower and upper bounds on x through the matrices vlb and vub. This restricts the solution to the range `vlb <= x <= vub`.

`x = sqp('fun',x0,options,vlb,vub,'grad')` uses the gradient information calculated by the function grad, defined in the M-file grad.m, rather than the default of approximating the partial derivatives via finite differencing.

`x = sqp('fun',x0,options,vlb,vub,'grad',p1,p2,...)` passes the problem-dependent parameters p1, p2, etc., directly to the functions fun and grad.

`[x,options] = sqp('fun',x0)` returns the parameters used in the optimization method. For example, `options(10)` contains the number of function evaluations used.

`[x,options,lambda] = sqp('fun',x0)` returns the vector lambda of the Lagrange multipliers at the solution `x`.
`[x,options,lambda,hess] = sqp('fun',x0)` also returns the approximation to the Hessian at the final iteration.

Arguments `fun` A string (or function handle) containing the name of the function that computes the objective function to be minimized and the constraint functions at the point *x.* The function fun returns two arguments: a scalar value of *f* and a vector of constraint values *g*,

```
[f,g] = fun(x)
```

When inequality constraints are present, the objective function f is minimized such that `g <= zeros(size(g))`.

Equality constraints, when present, are placed in the first elements of `g` and `options(13)` must be set to the number of equality constraints.

Alternatively, a string expression can be used with *x* representing the independent variables and with *f* and *g* representing the function and constraints.

```
x = sqp('f = fun(x); g = cstr(x);',x0)
```

Finally, `fun` may be replaced by a `fmincon` problem structure whose fields are the input argument variables (see Optimization Toolbox help):

```
x = sqp(problem)
```

`x0` Starting vector.

`vlb, vub` Upper and lower bound vectors. The variables, `vlb` and `vub`, are normally the same size as `x`. However, if `vlb` has *n* elements and fewer elements than `x`, then only the first *n* elements in `x` have lower bounds; upper bounds in `vub` are defined similarly.

`options` A vector of control parameters. Of the 18 elements of `options`, the input options used by `sqp` are: 1, 2, 3, 4, 9, 13, 14, 16, 17. When options is an output parameter, the `options` used by `sqp` to return values are: 8, 10, 11, 14.

- `options(1)` controls display. Setting this to a value of 1 produces a tabular display of intermediate results.
- `options(2)` controls the accuracy of x at the solution.
- `options(3)` controls the accuracy of f at the solution.
- `options(4)` sets the maximum constraint violation that is acceptable.

The termination criteria involving `options(2)`, `options(3)`, and `options(4)` must all hold true for the algorithm to terminate.

- `options(9)` Set to 1 if you want to check user-supplied gradients.
- `options(13)` Maximum number of function evaluations. (Default is 100*number of variables).
- `options(14)` Set to 1 if you want to check user-supplied gradients.
- `options(16)` Minimum change in variables for finite difference gradients.
- `options(17)` Maximum change in variables for finite difference gradients.

Alternatively, a structure following the new `fmincon` `options` (`optimset`). In addition to the usual `optimset` options, `options` may contain:

`options.foptions` – vector (<=18 length) of old-style foptions

`options.ComplexStep` – Set to `'on'` to use complex step in place of finite difference derivatives. `Fun` and `Grad` code must be able to accept and return complex values.

`options.LagrangeMultipliers` – initial Lagrange multiplier estimate

`options.HessMatrix` – initial positive-definite Hessian estimate

`options.HessFun` – user-supplied Hessian function handle: `H=Hessian(x,LagrangeMultipliers)`

`grad` A string (or function handle) containing the name of the function that computes the gradients of the objective and constraints at the point *x*. This function has the form

```
[df,dg] = grad(x)
```

The variable `df` is a vector that contains the partial derivatives of *f* with respect to *x*. The variable `dg` is a matrix where the columns of `dg` contain the partial derivatives for each of the constraints respectively, (i.e., the `i`th column of `dg` corresponds to the partial derivative of the `i`th constraint with respect to each of the elements in `x`).

Output Arguments

`x` Final design variable vector.

`options` A vector (or structure) summarizing results

- `options(8)` = value of the objective function at the solution (`options.fval`)
- `options(10)` = number of function evaluations
- `options(11)` = number of gradient evaluations
- `options(14)` = number of iterations

`lambda` A vector that returns the set of Lagrange multipliers at the solution. The length of lambda is `length(g)+length(vlb)+length(vub)` and the Lagrange multipliers are given in the corresponding order: first the multipliers for `g`, then `vlb`, then `vub`.

`hess` Modified Powell BFGS Quasi-Newton approximation to the Hessian matrix at the final iteration.

`status` Termination status: 0=converged.

Example Find values of *x* that minimize *f(x)*, starting at the point *x0*, and subject to the constraints, *g(x)*.

Step 1: Write an M-file:

```
function [f,g] = fun(x)
f = -x(1) * x(2) * x(3);
g(1) = -x(1) - 2 * x(2) - 2 * x(3); % Evaluate Constraints
g(2) = x(1) + 2 * x(2) + 2 * x(3) - 72;
```

Step 2: Invoke an optimization routine:

```
x0 = [10,10,10]; % Starting guess at the solution
x = sqp(@fun,x0) % Invoke optimizer
```

p1, p2,... Additional arguments to be passed to fun, that is, when *sqp* calls *fun*, and *grad* when it exists, the calls are

```
     [f,g]   = fun(x,p1,p2, ...)
     [df,dg] = grad(x,p1,p2, ...)
```

Using this feature, the same m-file can solve a number of similar problems with different parameters while avoiding the need to use global variables. Since all the arguments preceding *p1*, *p2*, etc., in the call to `sqp.m` must be defined, empty matrices may be passed in for options, `vlb`, `vub`, etc. to indicate that default arguments are to be used, as in

```
x = sqp(@fun,x0,[],[],[],@grad,p1,p2, ...)
```

Algorithm `sqp` 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 Powell's modified BFGS formula. A line search is performed using a merit function. It faithfully implements Schittkowski's SQP algorithm.

References

Schittkowski (1985). "NLPQL: A FORTRAN Subroutine Solving Constrained Nonlinear Programming Problems." *Annals Op. Res.* 5: 485-500.

Powell, M. J. D. (1978). "A Fast Algorithm for Nonlinearly Constrained Optimization Calculations." *Numerical Analysis*. G. A. Watson. New York, Springer-Verlag. 630: 144-157.

Squire, W. and G. Trapp (1998). "Using complex variables to estimate derivatives of real functions." *SIAM Review* 40(1): 110-112.

Limitations The objective function and constraints must both be continuous. SQP may only give only a local minimum. When the problem is infeasible, sqp attempts to minimize the maximum constraint value. The objective function and constraint functions must be real-valued, (cannot return complex values for real inputs). To use the *Complex Step* option, the functions must be analytic.

Features The authors, Mark Spillman and Robert A. Canfield, originally developed `sqp.m` function to overcome limitations of `constr.m` in Version 1 of the MATLAB Optimization Toolbox. They found that Schittkowski's NLPQL algorithm in the IMSL Fortran math library at the time, often outperformed the SQP implementation in MATLAB. Although the lastest versions of the Optimization Toolbox have added algorithms that are more robust than the original `constr.m` SQP algorithm, now known as the `active-set` algorithm in `fmincon`, the Spillamn-Canfield `sqp` m-file still offers unique benefits, some inherited from the rigor of Schittkowski's algorithm. Only some features were adopted in later versions of the Optimization Toolbox.

- *Complex-step* derivatives compute objective and constraint graidents with machine-precision accuracy [Squire, Trapp 1998] in the absence of user-supplied a `grad` m-file function. This avoids convergence studies to choose optimal step-size(s) for the forward finite difference option, and without the additional function evaluations for central difference derivatives. For compatible user-supplied code, it is simple as setting `options.ComplexStep='on'`. The user may still verify compmlex step with finite difference deriviatives by also setting `options.DerivativeCheck='on'` to confirm that the user's MATLAB code conforms to the analytic function requirement.

- **Separate** *fun* and *grad functions* often more naturally simplify programming, because one analysis tool often generates both the objective and the constraint function values, which can be assigned in a single m-file function. MATLAB changed the interface to divide the objective function and gradients from the constraint objective and gradients.

- **Lagrange Multiplier vector, *v*,** and **Hessian matrix, H,** are returned as output arguments. Then they may be re-used as input arguments to retain good starting values from a previous run, in place of otherwise arbitrarily initialized values. We improved efficiency and reliability by using this feature when generating Pareto optimal fronts by calling `sqp` in a loop, re-using last *v* and *H*.

- **Declare `ACTIVE_CONSTRAINTS`** as a global variable in the user's `grad` m-file to avoid computing gradients of inactive constraints.

- **Set** `opts(5)` to scale the design variables, x, and/or f and g. Although SQP is quadratically convergent near the optimum, properly scaling the variables and functions may accelerate convergence early in the iteration history, especially for highly nonlinear problems and starting far from the optimum.

- **Compatible** with `optimset options` to set tolerances and algorithm parameters conveniently, instead of using the foptions vector. A user may override (reasonable) defaults easily by specifying `options.TolX`, `options.TolFun`, `options.TolCon`, and `options.MaxIter` to control convergence and `options.Display` to control displayed output.

- **Compatible** with `optimset problem` structure input argument for compatibility with `fmincon` and `optimtool` problem export.

slp_trust

Purpose Use Sequential Linear Programming (SLP) in a Trust Region to minimize a constrained nonlinear multivariable function,

$$\min_{x} f(x) \quad \text{such that} \quad G(x) \leq 0$$

where *x* is a vector, *f(x)* is a function that returns a scalar, and *G(x)* is a function that returns a vector. Both *f(x)* and *G(x)* can be nonlinear functions.

Synopsis

```
x = slp_trust(@fun,x0,options,vlb,vub,@grad,p1,p2,…)
x = slp_trust (problem)
[x,f,exitflag,output,lambda] = trust(@fun,x0, ... )
```

Description `slp_trust` finds the constrained minimum of a scalar function of several variables starting at an initial estimate, complementing `sqp.m` for large numbers of variables. Although it is a first-order method, its redesign step uses an efficient, large-scale linear programming (LP) algorithm in concert with a trust region strategy. It's input arguments are identical to `sqp`. It typically takes more iterations to converge than SQP, but for large number of design variables, the optimization step at each iteration can be much faster. It has an optional active set strategy to avoid computing inactive constraint gradients (and finite differnce for inacctive constraints) with the following calling sequence:

```
[df,dg] = grad(x,active)
 dg(:,active) = ...
```

University of Illinois/NCSA Open Source License