NLPy Documentation

Release 0.2

Dominique Orban

September 03, 2010

CONTENTS

Release 0.2

Date September 03, 2010

Warning: Much remains to be done in terms of documentation. For now, the paper and inline documentation are all there is. With time, these pages will grow. Inline documentation is reproduced below in much more readable form. Please alert me of all errors or insufficient documentation strings.

Part I

The NLPy Library

CHAPTER

ONE

INTRODUCTION

NLPy is a Python package for numerical optimization. It aims to provide a toolbox for solving linear and nonlinear programming problems that is both easy to use and extensible.

NLPy combines the capabilities of the mature AMPL modeling language with the high-quality numerical resources and object-oriented power of the Python programming language. This combination makes NLPy an excellent tool to prototype and test optimization algorithms, and also a great teaching aid for optimization.

NLPy can read optimization problems coded in the AMPL modeling language. All aspects of a problem can be examined and the problem solved. Individual objective and constraint functions, together with their derivatives (if they exist) can be accessed transparently through an object-oriented framework.

NLPy is extensible and new algorithms can be built by assembling the supplied building blocks, or by creating new building blocks. Existing building blocks include procedures for solving symmetric (possibly indefinite) linear systems via symmetric factorizations or preconditioned iterative solvers, iterative methods for linear least-squares problems, minimization methods for constrained and unconstrained optimization, and much more.

1.1 Overview

NLPy is a collection of tools and interfaces for implementing and prototyping optimization algorithms. It is a set of Python modules and classes that support sparse matrices, nonlinear optimization methods, and the efficient solution of large sparse linear systems, especially those occuring in the course of optimization algorithms (e.g., symmetric indefinite systems).

The purpose of NLPy is to offer an environment in which implementing, testing, prototyping, experimenting with, and modifying and creating innovative optimization algorithms for large-scale constrained problems is a moderately easy task. We feel that the environment should be useful, simple, and intuitive enough that programmers need only concentrate on the logic of the algorithm instead of the intricacies of the programming language. We believe that NLPy is appropriate for teaching, for learning, and also for bleeding edge research in large-scale numerical optimization. This is achieved by providing the heavy-duty number-crunching procedures in fast, low-level languages such as C, Fortran 77, and Fortran 90/95.

NLPy aims to

- represent the bleeding edge of research in numerical (differentiable or not) optimization,
- provide a number of low-level tools upon which algorithms may be built, with the intent of solving potentially large problems,
- use sparse matrix data structures to do so,
- provide tools that are useful in the assessment of the performance of optimization algorithms.

1.2 Structure of NLPy

NLPy is designed as an object-oriented Python layer over lower-level subprograms. The subprograms, written in C, Fortran 77, and Fortran 90/95, implement the most numerically-intensive tasks.

These low-level subprograms include MA27 and MA57 for the multifrontal solution of symmetric linear systems, ICFS for preconditioned conjugate gradients with limited-memory Cholesky factorization, and GLTR for the solution of trust-region subproblems, to name a few.

All the sparse matrix capabilities in NLPy are based on the Pysparse package.

The main purpose of NLPy is to facilitate access and manipulation of optimization problems. Therefore, an interface to the AMPL modeling language was designed. It allows access to components of models and to take advantage of the automatic differentiation abilities of AMPL.

The above and the design of the Python language combine with interfaces written in C and in which only pointers are exchanged, leads to an environment suitable for the efficient solution of large-scale problems.

CHAPTER

TWO

INSTALLING NLPY

2.1 Requirements

- Numpy
- PySparse
- LibAmpl
- Several packages from the Harwell Subroutine Library. All required packages are available free of charge to academics. License terms apply. Please see the website for details.
- The incomplete Cholesky factorization with limited memory ICFS
- GALAHAD

Part II

Reference

CHAPTER

THREE

MODELING IN NLPY

3.1 Using Models in the AMPL Modeling Language

3.1.1 The `amplpy` Module

Python interface to the AMPL modeling language

class `amplpy.`**`AmplModel`**(*model*, ***kwargs*)

Bases: `nlpy.model.nlp.NLPModel`

AmplModel creates an instance of an AMPL model. If the *nl* file is already available, simply call *AmplModel(stub)* where the string *stub* is the name of the model. For instance: *AmplModel('elec')*. If only the *.mod* file is available, set the positional parameter *neednl* to *True* so AMPL generates the *nl* file, as in *AmplModel('elec.mod', data='elec.dat', neednl=True)*.

`A`(**args*)

Evaluate sparse Jacobian of the linear part of the constraints. Useful to obtain constraint matrix when problem is a linear programming problem.

`AtOptimality`(*x*, *y*, *z*, ***kwargs*)

See `OptimalityResiduals()` for a description of the arguments.

Returns

res tuple of residuals, as returned by `OptimalityResiduals()`,

optimal *True* if the residuals fall below the thresholds specified by `self.stop_d`, `self.stop_c` and `self.stop_p`.

`OptimalityResiduals`(*x*, *y*, *z*, ***kwargs*)

Evaluate the KKT or Fritz-John residuals at (x, y, z). The sign of the objective gradient is adjusted in this method as if the problem were a minimization problem.

Parameters

x Numpy array of length `n` giving the vector of primal variables,

y Numpy array of length `m` + `nrangeC` giving the vector of Lagrange multipliers for general constraints (see below),

z Numpy array of length `nbounds` + `nrangeB` giving the vector of Lagrange multipliers for simple bounds (see below).

Keywords

c constraints vector, if known. Must be in appropriate order (see below).

g objective gradient, if known.

J constraints Jacobian, if known. Must be in appropriate order (see below).

Returns vectors of residuals (dual_feas, compl, bnd_compl, primal_feas, bnd_feas)

The multipliers *y* associated to general constraints must be ordered as follows:

$c_i(x) = c_i^E$ (*i* in *equalC*): *y[:nequalC]*

$c_i(x) \geq c_i^L$ (*i* in *lowerC*): *y[nequalC:nequalC+nlowerC]*

$c_i(x) \leq c_i^U$ (*i* in *upperC*): *y[nequalC+nlowerC:nequalC+nlowerC+nupperC]*

$c_i(x) \geq c_i^L$ (*i* in *rangeC*): *y[nlowerC+nupperC:m]*

$c_i(x) \leq c_i^U$ (*i* in *rangeC*): *y[m:]*

For inequality constraints, the sign of each *y[i]* should be as if it corresponded to a nonnegativity constraint, i.e., $c_i^U - c_i(x) \geq 0$ instead of $c_i(x) \leq c_i^U$.

For equality constraints, the sign of each *y[i]* should be so the Lagrangian may be written:

```
L(x,y,z) = f(x) - <y, c_E(x)> - ...
```

Similarly, the multipliers *z* associated to bound constraints must be ordered as follows:

1.*x_i = L_i* (*i* in *fixedB*) : *z[:nfixedB]*

2.*x_i geq L_i* (*i* in *lowerB*) : *z[nfixedB:nfixedB+nlowerB]*

3.*x_i leq U_i* (*i* in *upperB*) : *z[nfixedB+nlowerB:nfixedB+nlowerB+nupperB]*

4.*x_i geq L_i* (*i* in *rangeB*) : *z[nfixedB+nlowerB+nupperB:nfixedB+nlowerB+nupperB+nrangeB]*

5.*x_i leq U_i* (*i* in *rangeB*) : *z[nfixedB+nlowerB+nupper+nrangeB:]*

The sign of each *z[i]* should be as if it corresponded to a nonnegativity constraint (except for fixed variables), i.e., those *z[i]* should be nonnegative.

It is possible to check the Fritz-John conditions via the keyword *FJ*. If *FJ* is present, it should be assigned the multiplier value that will be applied to the gradient of the objective function.

Example: *OptimalityResiduals(x, y, z, FJ=1.0e-5)*

If *FJ* has value *0.0*, the gradient of the objective will not be included in the residuals (it will not be computed).

ResetCounters()
Reset the *feval*, *geval*, *Heval*, *Hprod*, *ceval*, *Jeval* and *Jprod* counters of the current instance to zero.

close()

cons(*x*)
Evaluate vector of constraints at x. Returns a Numpy array.

The constraints appear in natural order. To order them as follows

1.equalities

2.lower bound only

3.upper bound only

4.range constraints,

use the *permC* permutation vector.

consPos(*x*)
Convenience function to return the vector of constraints reformulated as

ci(x) - ai = 0 for i in equalC ci(x) - Li >= 0 for i in lowerC + rangeC Ui - ci(x) >= 0 for i in upperC + rangeC.

The constraints appear in natural order, except for the fact that the 'upper side' of range constraints is appended to the list.

cost()
Evaluate sparse cost vector. Useful when problem is a linear program. Return a sparse vector. This method changes the sign of the cost vector if the problem is a maximization problem.

display_basic_info()
Display vital statistics about the current model.

grad(*x*)
Evaluate objective gradient at x. Returns a Numpy array. This method changes the sign of the objective gradient if the problem is a maximization problem.

hess(*x*, *z*, **args*)
Evaluate sparse lower triangular Hessian at (x, z). Returns a sparse matrix in format self.mformat (0 = compressed sparse row, 1 = linked list).

Note that the sign of the Hessian matrix of the objective function appears as if the problem were a minimization problem.

hprod(*z*, *v*, ***kwargs*)
Evaluate matrix-vector product H(x,z) * v. Returns a Numpy array.

Note that the sign of the Hessian matrix of the objective function appears as if the problem were a minimization problem.

icons(*i*, *x*)
Evaluate value of i-th constraint at x. Returns a floating-point number.

igrad(*i*, *x*)
Evaluate dense gradient of i-th constraint at x. Returns a Numpy array.

irow(*i*)
Evaluate sparse gradient of the linear part of the i-th constraint. Useful to obtain constraint rows when problem is a linear programming problem.

islp()
Determines whether problem is a linear programming problem.

jac(*x*, **args*)
Evaluate sparse Jacobian of constraints at x. Returns a sparse matrix in format self.mformat (0 = compressed sparse row, 1 = linked list).

The constraints appear in the following order:

1.equalities

2.lower bound only

3.upper bound only

4.range constraints.

jacPos(*x*)
Convenience function to evaluate the Jacobian matrix of the constraints reformulated as

ci(x) = ai for i in equalC

ci(x) - Li >= 0 for i in lowerC

ci(x) - Li >= 0 for i in rangeC

Ui - ci(x) >= 0 for i in upperC

Ui - ci(x) >= 0 for i in rangeC.

The gradients of the general constraints appear in 'natural' order, i.e., in the order in which they appear in the problem. The gradients of range constraints appear in two places: first in the 'natural' location and again after all other general constraints, with a flipped sign to account for the upper bound on those constraints.

The overall Jacobian of the new constraints thus has the form

[J] [-JR]

This is a *m* + *nrangeC* by *n* matrix, where *J* is the Jacobian of the general constraints in the order above in which the sign of the 'less than' constraints is flipped, and *JR* is the Jacobian of the 'less than' side of range constraints.

obj(*x*)
Evaluate objective function value at x. Returns a floating-point number. This method changes the sign of the objective value if the problem is a maximization problem.

set_x(*x*)
Set *x* as current value for subsequent calls to `obj()`, `grad()`, `jac()`, etc. If several of `obj()`, `grad()`, `jac()`, ..., will be called with the same argument *x*, it may be more efficient to first call *set_x(x)*. In AMPL, `obj()`, `grad()`, etc., normally check whether their argument has changed since the last call. Calling *set_x()* skips this check.

See also `unset_x()`.

sgrad(*x*)
Evaluate sparse objective gradient at x. Returns a sparse vector. This method changes the sign of the objective gradient if the problem is a maximization problem.

sigrad(*i*, *x*)
Evaluate sparse gradient of i-th constraint at x. Returns a sparse vector representing the sparse gradient in coordinate format.

unset_x()
Reinstates the default behavior of `obj()`, `grad()`, *jac*, etc., which is to check whether their argument has changed since the last call.

See also `set_x()`.

writesol(*x*, *z*, *msg*)
Write primal-dual solution and message msg to stub.sol

3.1.2 Example

```
#!/usr/bin/env python
#
# Test for amplpy module
#

from nlpy.model import AmplModel
import numpy
```

```
import getopt, sys

PROGNAME = sys.argv[0]

def commandline_err( msg ):
    sys.stderr.write( "%s: %s\n" % ( PROGNAME, msg ) )
    sys.exit( 1 )

def parse_cmdline( arglist ):
    if len( arglist ) != 1:
        commandline_err( 'Specify file name (look in data directory)' )
        return None

    try: options, fname = getopt.getopt( arglist, '' )

    except getopt.error, e:
        commandline_err( "%s" % str( e ) )
        return None

    return fname[0]

ProblemName = parse_cmdline(sys.argv[1:])

# Create a model
print 'Problem', ProblemName
nlp = AmplModel( ProblemName ) #amplpy.AmplModel( ProblemName )

# Query the model
x0  = nlp.x0
pi0 = nlp.pi0
n = nlp.n
m = nlp.m
print 'There are %d variables and %d constraints' % ( n, m )

max_n = min( n, 5 )
max_m = min( m, 5 )

print
print ' Printing at most 5 first components of vectors'
print

print 'Initial point: ', x0[:max_n]
print 'Lower bounds on x: ', nlp.Lvar[:max_n]
print 'Upper bounds on x: ', nlp.Uvar[:max_n]
print 'f( x0 ) = ', nlp.obj( x0 )
g0 = nlp.grad( x0 )
print 'grad f( x0 ) = ', g0[:max_n]

if max_m > 0:
    print 'Initial multipliers: ', pi0[:max_m]
    print 'Lower constraint bounds: ', nlp.Lcon[:max_m]
    print 'Upper constraint bounds: ', nlp.Ucon[:max_m]
    c0 = nlp.cons( x0 )
    print 'c( x0 ) = ', c0[:max_m]

J = nlp.jac( x0 )
H = nlp.hess( x0, pi0 )
print
```

```
print ' nnzJ = ', J.nnz
print ' nnzH = ', H.nnz

print
print ' Printing at most first 5x5 principal submatrix'
print

print 'J( x0 ) = ', J[:max_m,:max_n]
print 'Hessian (lower triangle):', H[:max_n,:max_n]

print
print ' Evaluating constraints individually, sparse gradients'
print

for i in range(max_m):
    ci = nlp.icons( i, x0 )
    print 'c%d( x0 ) = %-g' % (i, ci)
    sgi =  nlp.sigrad( i, x0 )
    k = sgi.keys()
    ssgi = {}
    for j in range( min( 5, len( k ) ) ):
        ssgi[ k[j] ] = sgi[ k[j] ]
    print 'grad c%d( x0 ) = ' % i, ssgi

print
print ' Testing matrix-vector product:'
print

e = numpy.ones( n, 'd' )
e[0] = 2
e[1] = -1
He = nlp.hprod( pi0, e )
print 'He = ', He[:max_n]

# Output "solution"
nlp.writesol( x0, pi0, 'And the winner is' )

# Clean-up
nlp.close()
```

3.2 Modeling with *NLPModel* Objects

3.2.1 The `nlp` Module

class `nlp.`**`NLPModel`**(*n=0*, *m=0*, *name='Generic'*, ***kwargs*)

Instances of class NLPModel represent an abstract nonlinear optimization problem. It features methods to evaluate the objective and constraint functions, and their derivatives. Instances of the general class do not do anything interesting; they must be subclassed and specialized.

Parameters

n number of variables (default: 0)

m number of general (non bound) constraints (default: 0)

name model name (default: 'Generic')

Keywords

x0 initial point (default: all 0)

pi0 vector of initial multipliers (default: all 0)

Lvar vector of lower bounds on the variables (default: all -Infinity)

Uvar vector of upper bounds on the variables (default: all +Infinity)

Lcon vector of lower bounds on the constraints (default: all -Infinity)

Ucon vector of upper bounds on the constraints (default: all +Infinity)

Constraints are classified into 3 classes: linear, nonlinear and network.

Indices of linear constraints are found in member `lin` (default: empty).

Indices of nonlinear constraints are found in member `nln` (default: all).

Indices of network constraints are found in member `net` (default: empty).

If necessary, additional arguments may be passed in kwargs.

AtOptimality(*x*, *z*, ***kwargs*)

OptimalityResiduals(*x*, *z*, ***kwargs*)

ResetCounters()

cons(*x*, ***kwargs*)

grad(*x*, ***kwargs*)

hess(*x*, *z*, ***kwargs*)

hprod(*x*, *z*, *p*, ***kwargs*)

icons(*i*, *x*, ***kwargs*)

igrad(*i*, *x*, ***kwargs*)

jac(*x*, ***kwargs*)

obj(*x*, ***kwargs*)

sigrad(*i*, *x*, ***kwargs*)

3.2.2 Example

Todo

Insert example.

3.3 Using the Slack Formulation of a Model

3.3.1 The `slacks` Module

A framework for converting a general nonlinear program into a farm with (possibly nonlinear) equality constraints and bounds only, by adding slack variables.

SlackFramework is a general framework for converting a nonlinear optimization problem to a form using slack variables.

The initial problem consists in minimizing an objective $f(x)$ subject to the constraints

$$\begin{aligned} c_i(x) = a_i, &\qquad i = 1, \ldots, m, \\ g_j^L \leq g_j(x) \leq g_j^U, &\qquad j = 1, \ldots, p, \\ x_k^L \leq x_k \leq x_k^U, &\qquad k = 1, \ldots, n, \end{aligned}$$

where some or all lower bounds g_j^L and x_k^L may be equal to $-\infty$, and some or all upper bounds g_j^U and x_k^U may be equal to $+\infty$.

The transformed problem is in the variables *x*, *s* and *t* and its constraints have the form

$$\begin{aligned} c_i(x) - a_i = 0, &\qquad i = 1, \ldots, m, \\ g_j(x) - g_j^L - s_j^L = 0, &\qquad j = 1, \ldots, p, \text{ for which } g_j^L > -\infty, \\ s_j^L \geq 0, &\qquad j = 1, \ldots, p, \text{ for which } g_j^L > -\infty, \\ g_j^U - g_j(x) - s_j^U = 0, &\qquad j = 1, \ldots, p, \text{ for which } g_j^U < +\infty, \\ s_j^U >= 0, &\qquad j = 1, \ldots, p, \text{ for which } g_j^U < +\infty, \\ x_k - x_k^L - t_k^L = 0, &\qquad k = 1, \ldots, n, \text{ for which } x_k^L > -\infty, \\ t_k^L >= 0, &\qquad k = 1, \ldots, n, \text{ for which } x_k^L > -\infty, \\ x_k^U - x_k - t_k^U = 0, &\qquad k = 1, \ldots, n, \text{ for which } x_k^U < +\infty, \\ t_k^U >= 0, &\qquad k = 1, \ldots, n, \text{ for which } x_k^U < +\infty. \end{aligned}$$

In the latter problem, the only inequality constraints are bounds on the slack variables. The other constraints are (typically) nonlinear equalities.

The order of variables in the transformed problem is as follows:

[x | sL | sU | tL | tU]

where:

- sL = [sLL | sLR], sLL being the slack variables corresponding to general constraints with a lower bound only, and sLR being the slack variables corresponding to the 'lower' side of range constraints.
- sU = [sUU | sUR], sUU being the slack variables corresponding to general constraints with an upper bound only, and sUR being the slack variables corresponding to the 'upper' side of range constraints.
- tL = [tLL | tLR], tLL being the slack variables corresponding to variables with a lower bound only, and tLR being the slack variables corresponding to the 'lower' side of two-sided bounds.
- tU = [tUU | tUR], tUU being the slack variables corresponding to variables with an upper bound only, and tLR being the slack variables corresponding to the 'upper' side of two-sided bounds.

This framework initializes the slack variables sL, sU, tL, and tU to zero by default.

Note that the slack framework does not update all members of AmplModel, such as the index set of constraints with an upper bound, etc., but rather performs the evaluations of the constraints for the updated model implicitly.

class `slacks.`**`SlackFramework`**(*model*, ***kwargs*)

Bases: `nlpy.model.amplpy.AmplModel`

General framework for converting a nonlinear optimization problem to a form using slack variables.

In the latter problem, the only inequality constraints are bounds on the slack variables. The other constraints are (typically) nonlinear equalities.

The order of variables in the transformed problem is as follows:

1.x, the original problem variables.

2. sL = [sLL | sLR], sLL being the slack variables corresponding to general constraints with a lower bound only, and sLR being the slack variables corresponding to the 'lower' side of range constraints.
3. sU = [sUU | sUR], sUU being the slack variables corresponding to general constraints with an upper bound only, and sUR being the slack variables corresponding to the 'upper' side of range constraints.
4. tL = [tLL | tLR], tLL being the slack variables corresponding to variables with a lower bound only, and tLR being the slack variables corresponding to the 'lower' side of two-sided bounds.
5. tU = [tUU | tUR], tUU being the slack variables corresponding to variables with an upper bound only, and tLR being the slack variables corresponding to the 'upper' side of two-sided bounds.

This framework initializes the slack variables sL, sU, tL, and tU to zero by default.

Note that the slack framework does not update all members of AmplModel, such as the index set of constraints with an upper bound, etc., but rather performs the evaluations of the constraints for the updated model implicitly.

A()
Return the constraint matrix if the problem is a linear program. See the documentation of `jac()` for more information.

Bounds(*x*)
Evaluate the vector of equality constraints corresponding to bounds on the variables in the original problem.

InitializeSlacks(*val=0.0*, ***kwargs*)
Initialize all slack variables to given value. This method may need to be overridden.

cons(*x*)
Evaluate the vector of general constraints for the modified problem. Constraints are stored in the order in which they appear in the original problem. If constraint i is a range constraint, c[i] will be the constraint that has the slack on the lower bound on c[i]. The constraint with the slack on the upper bound on c[i] will be stored in position m + k, where k is the position of index i in rangeC, i.e., k=0 iff constraint i is the range constraint that appears first, k=1 iff it appears second, etc.

Constraints appear in the following order:

[c] general constraints in orignal order [cR] 'upper' side of range constraints [b] linear constraints corresponding to bounds on original problem [bR] linear constraints corresponding to 'upper' side of two-sided

bounds

jac(*x*)
Evaluate the Jacobian matrix of all equality constraints of the transformed problem. The gradients of the general constraints appear in 'natural' order, i.e., in the order in which they appear in the problem. The gradients of range constraints appear in two places: first in the 'natural' location and again after all other general constraints, with a flipped sign to account for the upper bound on those constraints.

The gradients of the linear equalities corresponding to bounds on the original variables appear in the following order:

1. variables with a lower bound only
2. lower bound on variables with two-sided bounds
3. variables with an upper bound only
4. upper bound on variables with two-sided bounds

The overall Jacobian of the new constraints thus has the form:

```
[ J     −I           ]
[−JR       −I        ]
[ I           −I     ]
[−I              −I  ]
```

where the columns correspond, in order, to the variables x, s, sU, t, and tU, the rows correspond, in order, to

1. general constraints (in natural order)
2. 'upper' side of range constraints
3. bounds, ordered as explained above
4. 'upper' side of two-sided bounds,

and where the signs corresponding to 'upper' constraints and upper bounds are flipped in the (1,1) and (3,1) blocks.

obj (x)

3.3.2 Example

Todo

Insert example.

3.3.3 Inheritance Diagram

CHAPTER

FOUR

DIRECT SOLUTION OF SYSTEMS OF LINEAR EQUATIONS

4.1 Scaling a Sparse Matrix

4.1.1 The `scaling` Module

Todo

Write Python wrapper.

4.2 Direct Solution of Symmetric Systems of Equations

4.2.1 The `sils` Module

SILS: An abstract framework for the factorization of symmetric indefinite matrices. The factorizations currently implemented are those of MA27 and MA57 from the Harwell Subroutine Library (http://hsl.rl.ac.uk).

class `sils.`**`Sils`**(*A*, ***kwargs*)

Abstract class for the factorization and solution of symmetric indefinite systems of linear equations. The methods of this class must be overridden.

`fetch_perm`()

Must be subclassed.

`refine`(*b*, *nitref=3*)

Must be subclassed.

`solve`(*b*, *get_resid=True*)

Must be subclassed.

4.2.2 The `pyma27` Module

Ma27: Direct multifrontal solution of symmetric systems

class `pyma27.`**`PyMa27Context`**(*A*, ***kwargs*)

Bases: `sils.Sils`

`fetch_lb`()

fetch_lb() returns the factors L and B of A such that

P^T A P = L B L^T

where P is as in fetch_perm(), L is unit upper triangular and B is block diagonal with 1x1 and 2x2 blocks. Access to the factors is available as soon as a PyMa27Context has been instantiated.

fetch_perm()
fetch_perm() returns the permutation vector p used to compute the factorization of A. Rows and columns were permuted so that

P^T A P = L B L^T

where i-th row of P is the p(i)-th row of the identity matrix, L is unit upper triangular and B is block diagonal with 1x1 and 2x2 blocks.

refine(*b*, *nitref=3*, *tol=1e-08*, ***kwargs*)
refine(b, tol, nitref) performs iterative refinement if necessary until the scaled residual norm ||b-Ax||/(1+||b||) falls below the threshold 'tol' or until nitref steps are taken. Make sure you have called solve() with the same right-hand side b before calling refine(). The residual vector self.residual will be updated to reflect the updated approximate solution.

By default, tol = 1.0e-8 and nitref = 3.

solve(*b*, *get_resid=True*)
solve(b) solves the linear system of equations Ax = b. The solution will be found in self.x and residual in self.residual.

4.2.3 The pyma57 Module

Ma57: Direct multifrontal solution of symmetric systems

class pyma57.**PyMa57Context**(*A*, *factorize=True*, ***kwargs*)
Bases: sils.Sils

factorize(*A*)
Perform numerical factorization. Before this can be done, symbolic factorization (the "analyze" phase) must have been performed.

The values of the elements of the matrix may have been altered since the analyze phase but the sparsity pattern must not have changed. Use the optional argument newA to specify the updated matrix if applicable.

fetch_perm()
fetch_perm() returns the permutation vector p used to compute the factorization of A. Rows and columns were permuted so that

P^T A P = L B L^T

where i-th row of P is the p(i)-th row of the identity matrix, L is unit upper triangular and B is block diagonal with 1x1 and 2x2 blocks.

refine(*b*, *nitref=3*, ***kwargs*)
refine(b, nitref) performs iterative refinement if necessary until the scaled residual norm ||b-Ax||/(1+||b||) falls below the threshold 'tol' or until nitref steps are taken. Make sure you have called solve() with the same right-hand side b before calling refine(). The residual vector self.residual will be updated to reflect the updated approximate solution.

By default, nitref = 3.

solve(*b*, *get_resid=True*)
solve(b) solves the linear system of equations Ax = b. The solution will be found in self.x and residual in self.residual.

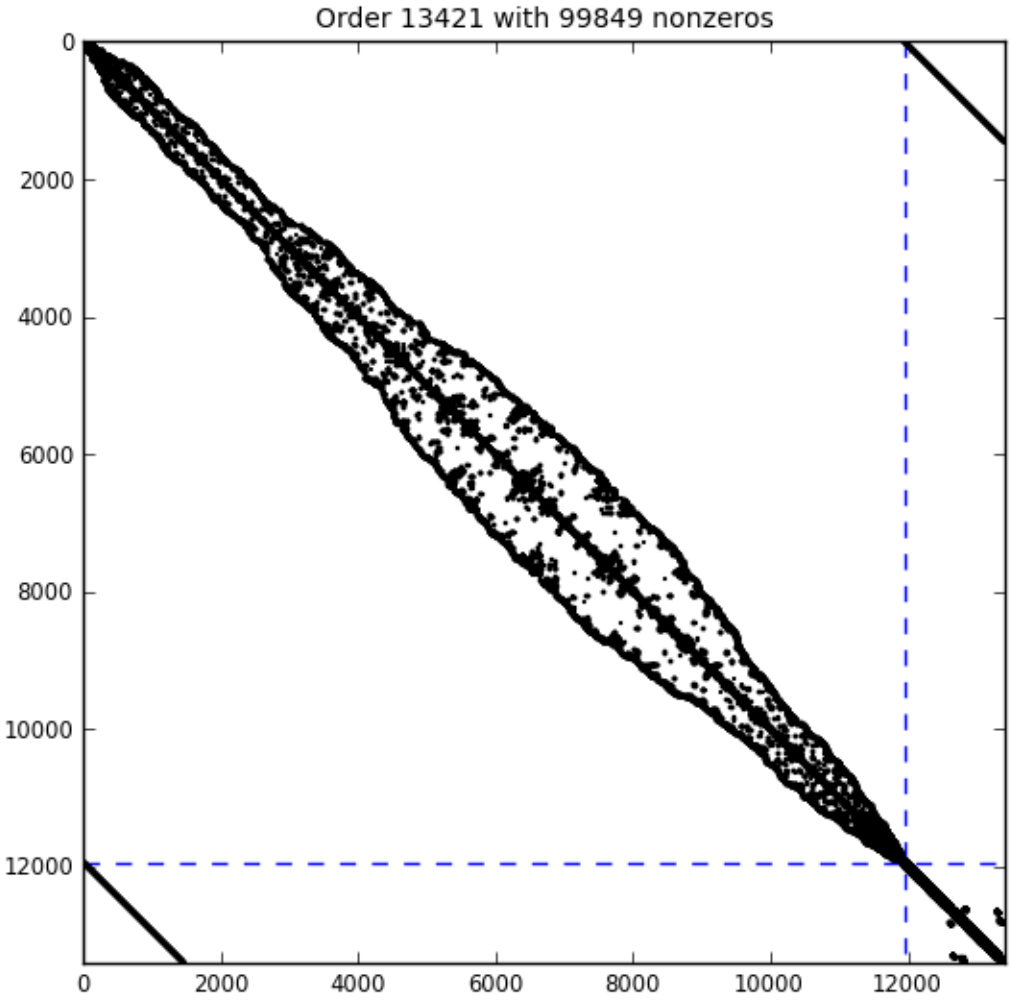
4.2.4 Example

```
# Demonstrate usage of PyMa27 abstract class for the solution of symmetric
# systems of linear equations.
# Example usage: python demo_ma27.py file1.mtx ... fileN.mtx
# where each fileK.mtx is in MatrixMarket format.

from nlpy.linalg.pyma27 import PyMa27Context as LBLContext
#from nlpy.linalg.pyma57 import PyMa57Context as LBLContext
from pysparse import spmatrix
from nlpy.tools import norms
from nlpy.tools.timing import cputime
import numpy

def Hilbert(n):
    """
    The cream of ill conditioning: the Hilbert matrix.  See Higham,
    "Accuracy and Stability of Numerical Algoriths", section 28.1.
    The matrix has elements H(i,j) = 1/(i+j-1) when indexed
    i,j=1..n.  However, here we index as i,j=0..n-1, so the elements
    are H(i,j) = 1/(i+j+1).
    """
    if n <= 0: return None
    if n == 1: return 1.0
    nnz = n * (n - 1)/2
    H = spmatrix.ll_mat_sym(n, nnz)
    for i in range(n):
        for j in range(i+1):
            H[i,j] = 1.0/(i+j+1)
    return H

def Ma27SpecSheet():
    # This is the example from the MA27 spec sheet
    # Solution should be [1,2,3,4,5]
    A = spmatrix.ll_mat_sym(5, 7)
    A[0,0] = 2
    A[1,0] = 3
    A[2,1] = 4
    A[2,2] = 1
    A[3,2] = 5
    A[4,1] = 6
    A[4,4] = 1

    rhs = numpy.ones(5, 'd')
    rhs[0] = 8
    rhs[1] = 45
    rhs[2] = 31
    rhs[3] = 15
    rhs[4] = 17

    return (A, rhs)

def SolveSystem(A, rhs, itref_threshold=1.0e-6, nitrefmax=5, **kwargs):

    # Obtain Sils context object
    t = cputime()
    LBL = LBLContext(A, **kwargs)
    t_analyze = cputime() - t
```

```
57
58      # Solve system and compute residual
59      t = cputime()
60      LBL.solve(rhs)
61      t_solve = cputime() - t_analyze
62
63      # Compute residual norm
64      nrhsp1 = norms.norm_infty(rhs) + 1
65      nr = norms.norm2(LBL.residual)/nrhsp1
66
67      # If residual not small, perform iterative refinement
68      LBL.refine(rhs, tol = itref_threshold, nitref=nitrefmax)
69      nr1 = norms.norm_infty(LBL.residual)/nrhsp1
70
71      return (LBL.x, LBL.residual, nr, nr1, t_analyze, t_solve, LBL.neig)
72
73  if __name__ == '__main__':
74      import sys
75      import os
76      matrices = sys.argv[1:]
77
78      hdr_fmt = '%-13s  %-11s  %-11s  %-11s  %-7s  %-7s  %-5s\n'
79      res_fmt = '%-13s  %-11.5e  %-11.5e  %-11.5e  %-7.3f  %-7.3f  %-5d\n'
80      hdrs = ('Name', 'Rel. resid.', 'Residual', 'Resid itref', 'Analyze',
81              'Solve', 'neig')
82      header = hdr_fmt % hdrs
83      lhead = len(header)
84      sys.stderr.write('-' * lhead + '\n')
85      sys.stderr.write(header)
86      sys.stderr.write('-' * lhead + '\n')
87
88      # Solve example from the spec sheet
89      (A, rhs) = Ma27SpecSheet()
90      (x, r, nr, nr1, t_an, t_sl, neig) = SolveSystem(A, rhs)
91      exact = numpy.arange(5, dtype = 'd') + 1
92      relres = norms.norm2(x - exact) / norms.norm2(exact)
93      sys.stdout.write(res_fmt % ('Spec sheet',relres,nr,nr1,t_an,t_sl,neig))
94
95      # Solve example with Hilbert matrix
96      n = 10
97      H = Hilbert(n)
98      e = numpy.ones(n, 'd')
99      rhs = numpy.empty(n, 'd')
100     H.matvec(e, rhs)
101     (x, r, nr, nr1, t_an, t_sl, neig) = SolveSystem(H, rhs)
102     relres = norms.norm2(x - e) / norms.norm2(e)
103     sys.stdout.write(res_fmt % ('Hilbert', relres, nr, nr1, t_an, t_sl, neig))
104
105     # Process matrices given on the command line
106     for matrix in matrices:
107         M = spmatrix.ll_mat_from_mtx(matrix)
108         (m,n) = M.shape
109         if m != n: break
110         e = numpy.ones(n, 'd')
111         rhs = numpy.empty(n, 'd')
112         M.matvec(e, rhs)
113         (x, r, nr, nr1, t_an, t_sl, neig) = SolveSystem(M, rhs)
114         relres = norms.norm2(x - e) / norms.norm2(e)
```

```
115         probname = os.path.basename(matrix)
116         if probname[-4:] == '.mtx': probname = probname[:-4]
117         sys.stdout.write(res_fmt % (probname,relres,nr,nr1,t_an,t_sl,neig))
118     sys.stderr.write('-' * lhead + '\n')
```

4.3 Direct Solution of Symmetric Quasi-Definite Systems of Equations



CHAPTER

FIVE

ITERATIVE SOLUTION OF SYSTEMS OF LINEAR EQUATIONS

5.1 Symmetric Systems of Equations

5.1.1 The `minres` Module

Solve the linear system

A x = b

or the least-squares problem

minimize ||Ax-b||

using Minres. This is a line-by-line translation from Matlab code available at http://www.stanford.edu/group/SOL/software/minres.htm.

class `minres.`**`Minres`**(*A*, ***kwargs*)

K = Minres(A) ; K.solve(b)

This class implements the Minres iterative solver of Paige and Saunders. Minres solves the system of linear equations Ax = b or the least squares problem min ||Ax - b||_2^2, where A is a symmetric matrix (possibly indefinite or singular) and b is a given vector.

A may be given explicitly as a matrix or be a function such that

A(x, y)

stores in y the product Ax for any given vector x. If A is an instance of some matrix class, it should have a 'matvec' method such that A.matvec(x, y) has the behaviour described above.

Optional keyword arguments are:

precon optional preconditioner, given as an operator (None) shift optional shift value (0.0) show display information along the iterations (True) check perform some argument checks (True) itnlim maximum number of iterations (5n) rtol relative stopping tolerance (1.0e-12)

If precon is given, it must define a positive-definite preconditioner M = C C'. The precon operator must be such that

x = precon(y)

returns the solution x to the linear system M x = y, for any given y.

If shift != 0, minres really solves (A - shift*I)x = b or the corresponding least squares problem if shift is an eigenvalue of A.

The return values (as returned by the Matlab version of Minres) are stored in the members

x, *istop*, *itn*, *rnorm*, *Arnorm*, *Anorm*, *Acond*, *ynorm*

of the class, after completion of `solve()`.

Python version: Dominique Orban, Ecole Polytechnique de Montreal, 2008, translated and adapted from the Matlab version of Minres, written by

Michael Saunders, SOL, Stanford University Sou Cheng Choi, SCCM, Stanford University

See also http://www.stanford.edu/group/SOL/software/minres.html

applyA(*x*, *y*)
Given x, compute the matrix-vector product y = Ax

normof2(*x*, *y*)

solve(*b*, ***kwargs*)

5.1.2 The `pcg` Module

A pure Python/numpy implementation of the Steihaug-Toint truncated preconditioned conjugate gradient algorithm as described in

T. Steihaug, *The conjugate gradient method and trust regions in large scale optimization*, SIAM Journal on Numerical Analysis **20** (3), pp. 626-637, 1983.

class `pcg.`**TruncatedCG**(*g*, ***kwargs*)

Solve(***kwargs*)
Solve the trust-region subproblem.

to_boundary(*s*, *p*, *radius*, *ss=None*)
Given vectors *s* and *p* and a trust-region radius *radius* > 0, return the positive scalar *sigma* such that

|| *s* + *sigma* * *p* || = *radius*

in Euclidian norm. If known, supply optional argument *ss* whose value should be the squared Euclidian norm of *s*.

5.1.3 The `pygltr` Module

Solution of a trust-region subproblem using the preconditioned Lanczos method.

class `pygltr.`**PyGltrContext**(*g*, ***kwargs*)
Create a new instance of a PyGltrContext object, representing a context to solve the quadratic problem

min <g, d> + 1/2 <d, Hd> s.t ||d|| <= radius

where either the Hessian matrix H or a means to compute matrix-vector products with H, are to be specified later.

Parameters

g gradient vector (of length *n*)

Keywords

radius trust-region radius (default: 1.0)

reltol relative stopping tolerance (default: sqrt(eps))

abstol absolute stopping tolerance (default: 0.0)

prec function solving preconditioning systems. If M is a preconditioner, *prec(v)* returns a solution to the linear system of equations Mx = v (default: *None*)

itmax maximum number of iterations (default: *n*)

litmax maximum number of Lanczos iterations on the boundary (default: *n*)

ST Use Steihaug-Toint strategy (default: *False*)

boundary Indicates whether the solution is thought to lie on the boundary of the trust region (default: *False*)

equality Require that the solution lie on the boundary (default: *False*)

fraction Fraction of optimality that is acceptable. A value smaller than *1.0* results in a correspondingly sub-optimal solution. (default: *1.0*)

See the GLTR spec sheet for more information on these parameters.

Convergence of the iteration takes place as soon as

Norm(Hd + l Md + g) <= max(Norm(g) * reltol, abstol)

where M is a preconditioner l is an estimate of the Lagrange multipliers Norm() is the M^{-1}-norm

explicit_solve(*H*)
Solves the quadratic trust-region subproblem whose data was specified upon initialization. During the reverse communication phase, matrix vector products with the Hessian H will be computed explicitly using the matvec method of the object H. For instance, if H is an ll_mat, or csr_mat, products will be evaluated using H.matvec(x,y).

implicit_solve(*hessprod*)
Solves the quadratic trust-region subproblem whose data was specified upon initialization. During the reverse communication phase, matrix vector products with the Hessian H will be computed implicitly using the supplied hessprod method. Given an array v, hessprod must return an array of the same size containing the result of the multiplication H*v. For instance, if the problem is from an Ampl model called nlp, the hessprod method could be

lambda v: nlp.hprod(z, v)

for some multiplier estimates z.

5.1.4 The `projKrylov` Module

A general framework for implementing projected Krylov methods. Such methods are variations on all the well-known Krylov methods to solve block augmented linear systems, i.e., linear systems of the form

[H A^T] [x] = [c] [A 0] [y] [b],

where H and A are matrices and A^T is the transpose of A. Here, H may or may not be symmetric and may or may not have stronger properties, such as positive definiteness. It may be available as an operator only or explicitly. However, all projected Krylov methods currently require that B be available explicitly.

Such matrices arise, for example, in the solution of partial differential equations (e.g., Maxwell or Navier-Stokes) by the finite-element method. For more information on general Krylov methods and projected Krylov methods, see the references below.

This module defines the *ProjectedKrylov* generic class. Other modules subclass *ProjectedKrylov* to implement specific algorithms. Currently, the following methods are implemented

Method	Module	Class
Projected Conjugate Gradient	ppcg	Ppcg
Projected Bi-CGSTAB	pbcgstab	Pbcgstab

Other projected iterative methods may be found as part of the PyKrylov package. See http://github.com/dpo/pykrylov.

References

class projKrylov.**ProjectedKrylov**(*c*, ***kwargs*)

CheckAccurate()
Make sure constraints are consistent and residual is satisfactory

Factorize()
Assemble projection matrix and factorize it

P = [G A^T] [A 0],

where G is the preconditioner, or the identity matrix if no preconditioner was given.

FindFeasible()
If rhs was specified, obtain x_feasible satisfying the constraints

Solve()
This is the Solve method of the abstract projectedKrylov class. The class must be specialized and this method overridden.

5.1.5 The ppcg Module

An implementation of the projected conjugate gradient algorithm as described in

N.I.M. Gould, M.E. Hribar and J. Nocedal, *On the Solution of Equality Constrained Quadratic Programming Problems Arising in Optimization*, SIAM Journal on Scientific Computing **23** (4), pp. 1376-1395, 2001.

with the addition of an optional trust-region constraint.

class ppcg.**ProjectedCG**(*c*, ***kwargs*)
Bases: nlpy.krylov.projKrylov.ProjectedKrylov

CheckAccurate()
Make sure constraints are consistent and residual is satisfactory

Factorize()
Assemble projection matrix and factorize it

P = [G A^T] [A 0],

where G is the preconditioner, or the identity matrix if no preconditioner was given.

FindFeasible()
If rhs was specified, obtain x_feasible satisfying the constraints

Solve()

ftb(*s*, *p*)
If fraction-to-the-boundary rule is to be enforced, compute step length to satisfy s + t*p >= btol * cur_iter.

to_boundary(*s*, *p*, *Delta*, *ss=None*)
Given vectors *s* and *p* and a trust-region radius *Delta* > 0, return the positive scalar sigma such that

|| s + sigma * p || = Delta

in Euclidian norm. If known, supply optional argument *ss* whose value should be the squared Euclidian norm of argument *s*.

5.2 Non-Symmetric Systems of Equations

5.2.1 The `pbcgstab` Module

An implementation of the projected Bi-CGSTAB algorithm as described in

D. Orban *Projected Krylov Methods for Unsymmetric Augmented Systems*, Cahiers du GERAD G-2008-46, GERAD, Montreal, Canada, 2008.

class pbcgstab.**ProjectedBCGSTAB**(*c*, ***kwargs*)
Bases: projKrylov.ProjectedKrylov

CheckAccurate()
Make sure constraints are consistent and residual is satisfactory

Factorize()
Assemble projection matrix and factorize it

P = [G A^T] [A 0],

where G is the preconditioner, or the identity matrix if no preconditioner was given.

FindFeasible()
If rhs was specified, obtain x_feasible satisfying the constraints

Solve()

CHAPTER

SIX

OPTIMIZATION TOOLS

6.1 Linesearch Methods

6.1.1 The `linesearch` Module

class `linesearch.`**`LineSearch`**(***kwargs*)

A generic linesearch class. Most methods of this class should be overridden by subclassing.

`search`(*func*, *x*, *d*, *slope*, *f=None*, ***kwargs*)

Given a descent direction d for function func at the current iterate x, compute a steplength t such that func(x + t * d) satisfies a linesearch condition when compared to func(x). The value of the argument slope should be the directional derivative of func in the direction d: slope = f'(x;d) < 0. If given, f should be the value of func(x). If not given, it will be evaluated.

func can point to a defined function or be a lambda function. For example, in the univariate case:

test(lambda x: x**2, 2.0, -1, 4.0)

class `linesearch.`**`ArmijoLineSearch`**(***kwargs*)

Bases: `linesearch.LineSearch`

An Armijo linesearch with backtracking. This class implements the simple Armijo test

f(x + t * d) <= f(x) + t * beta * f'(x;d)

where 0 < beta < 1/2 and f'(x;d) is the directional derivative of f in the direction d. Note that f'(x;d) < 0 must be true.

Keywords

beta Value of beta. Default: 0.001

tfactor Amount by which to reduce the steplength during the backtracking. Default: 0.5.

`search`(*func*, *x*, *d*, *slope*, *f=None*, ***kwargs*)

Given a descent direction d for function func at the current iterate x, compute a steplength t such that func(x + t * d) satisfies the Armijo linesearch condition when compared to func(x). The value of the argument slope should be the directional derivative of func in the direction d: slope = f'(x;d) < 0. If given, f should be the value of func(x). If not given, it will be evaluated.

func can point to a defined function or be a lambda function. For example, in the univariate case:

*test(lambda x: x**2, 2.0, -1, 4.0)*

6.1.2 The pyswolfe Module

Linesearch methods guaranteeing satisfaction of the strong Wolfe conditions.

class pyswolfe.**StrongWolfeLineSearch**(*f*, *g*, *d*, *obj*, *grad*, ***kwargs*)

A general-purpose linesearch procedure enforcing the strong Wolfe conditions

f(x+td) <= f(x) + ftol * t * <g(x),d> (Armijo condition) <g(x+td),d> | <= gtol * | <g(x),d> | (curvature condition)

This is a Python interface to the More and Thuente linesearch.

Instantiate as follows

SWLS = StrongWolfeLineSearch(f, g, d, obj, grad)

where

- f is the objective value at the current iterate x
- g is the objective gradient at the current iterate x
- d is the current search direction
- **obj is a scalar function used to evaluate the value of** the objective at x + t d, given t.
- **grad is a scalar function used to evaluate the gradient** of the objective at x + t d, given t.

Keywords

ftol the constant used in the Armijo condition (1e-3)

gtol the constant used in the curvature condition (0.9)

xtol a minimal relative step bracket length (1e-10)

stp an initial step value (1.0)

stpmin the initial lower bound of the bracket

stpmax the initial upper bound of the bracket

To ensure existence of a step satisfying the strong Wolfe conditions, d should be a descent direction for f at x and ftol <= gtol.

The final value of the step will be held in SWLS.stp

In case an error happens, the return value SWLS.stp will be set to None and SWLS.message will describe what happened.

After the search, SWLS.armijo will be set to True if the step computed satisfies the Armijo condition and SWLS.curvature will be set to True if the step satisfies the curvature condition.

search()

6.1.3 The pymswolfe Module

PyMSWolfe: Jorge Nocedal's modified More and Thuente linesearch guaranteeing satisfaction of the strong Wolfe conditions.

class pymswolfe.**StrongWolfeLineSearch**(*f*, *x*, *g*, *d*, *obj*, *grad*, ***kwargs*)

A general-purpose linesearch procedure enforcing the strong Wolfe conditions

f(x+td) <= f(x) + ftol * t * <g(x),d> (Armijo condition) <g(x+td),d> | <= gtol * | <g(x),d> | (curvature condition)

This is a Python interface to Jorge Nocedal's modification of the More and Thuente linesearch. Usage of this class is slightly different from the original More and Thuente linesearch.

Instantiate as follows

SWLS = StrongWolfeLineSearch(f, x, g, d, obj, grad)

where

- f is the objective value at the current iterate x
- x is the current iterate
- g is the objective gradient at the current iterate x
- d is the current search direction
- **obj is a scalar function used to evaluate the value of** the objective at a given point
- **grad is a scalar function used to evaluate the gradient** of the objective at a given point.

Keywords

ftol the constant used in the Armijo condition (1e-4)

gtol the constant used in the curvature condition (0.9)

xtol a minimal relative step bracket length (1e-16)

stp an initial step value (1.0)

stpmin the initial lower bound of the bracket (1e-20)

stpmax the initial upper bound of the bracket (1e+20)

maxfev the maximum number of function evaluations permitted (20)

To ensure existence of a step satisfying the strong Wolfe conditions, d should be a descent direction for f at x and ftol <= gtol.

The final value of the step will be held in SWLS.stp

After the search, SWLS.armijo will be set to True if the step computed satisfies the Armijo condition and SWLS.curvature will be set to True if the step satisfies the curvature condition.

search()

6.2 Trust-Region Methods

6.2.1 The `trustregion` Module

Class definition for Trust-Region Algorithm

class trustregion.**TrustRegionFramework**(***kwargs*)

Initializes an object allowing management of a trust region.

Keywords

Delta Initial trust-region radius (default: 1.0)

eta1 Step acceptance threshold (default: 0.01)

eta2 Radius increase threshold (default: 0.99)

gamma1 Radius decrease factor (default: 1/3)

gamma2 Radius increase factor (default: 2.5)

Subclass and override `UpdateRadius()` to implement custom trust-region management rules.

See, e.g.,

A. R. Conn, N. I. M. Gould and Ph. L. Toint, Trust-Region Methods, MP01 MPS-SIAM Series on Optimization, 2000.

ResetRadius()
Reset radius to original value

Rho(*f*, *f_trial*, *m*)
Compute the ratio of actual versus predicted reduction rho = (f - f_trial)/(-m)

UpdateRadius(*rho*, *stepNorm*)
Update the trust-region radius. The rule implemented by this method is:

Delta = gamma1 * stepNorm if ared/pred < eta1 Delta = gamma2 * Delta if ared/pred >= eta2 Delta unchanged otherwise,

where ared/pred is the quotient computed by self.Rho().

class trustregion.**TrustRegionSolver**(*g*, ***kwargs*)
A generic template class for implementing solvers for the trust-region subproblem

minimize q(d) subject to ||d|| <= radius,

where q(d) is a quadratic function of the n-vector d, i.e., q has the general form q(d) = g' d + 1/2 d' H d,

where *g* is a n-vector typically interpreted as the gradient of some merit function and *H* is a real symmetric n-by-n matrix. Note that *H* need not be positive semi-definite.

The trust-region constraint ||*d*|| <= *radius* can be defined in any norm although most derived classes currently implement the Euclidian norm only. Note however that any elliptical norm may be used via a preconditioner.

For more information on trust-region methods, see

A. R. Conn, N. I. M. Gould and Ph. L. Toint, Trust-Region Methods, MP01 MPS-SIAM Series on Optimization, 2000.

Solve()
Solve the trust-region subproblem. This method must be overridden.

class trustregion.**TrustRegionCG**(*g*, ***kwargs*)
Bases: `trustregion.TrustRegionSolver`

Instantiate a trust-region subproblem solver based on the truncated conjugate gradient method of Steihaug and Toint. See the `pcg` module for more information.

The main difference between this class and the `TrustRegionPCG` class is that `TrustRegionPCG` only accepts explicit preconditioners (i.e. in matrix form). This class accepts an implicit preconditioner, i.e., any callable object.

Solve(***kwargs*)
Solve trust-region subproblem using the truncated conjugate-gradient algorithm.

class trustregion.**TrustRegionPCG**(*g*, *A*, ***kwargs*)
Bases: `trustregion.TrustRegionSolver`

Instantiate a trust-region subproblem solver based on the projected truncated conjugate gradient of Gould, Hribar and Nocedal. See the `ppcg` module for more information.

The trust-region subproblem has the form

minimize q(d) subject to Ad = 0 and ||d|| <= radius,

where q(d) is a quadratic function of the n-vector d, i.e., q has the general form q(d) = g' d + 1/2 d' H d,

where *g* is a n-vector typically interpreted as the gradient of some merit function and *H* is a real symmetric n-by-n matrix. Note that *H* need not be positive semi-definite.

The trust-region constraint ||*d*|| <= *radius* can be defined in any norm although most derived classes currently implement the Euclidian norm only. Note however that any elliptical norm may be used via a preconditioner.

For more information on trust-region methods, see

A. R. Conn, N. I. M. Gould and Ph. L. Toint, Trust-Region Methods, MP01 MPS-SIAM Series on Optimization, 2000.

Solve()
Solve trust-region subproblem using the projected truncated conjugate gradient algorithm.

class trustregion.**TrustRegionGLTR**(*g*, ***kwargs*)
Bases: trustregion.TrustRegionSolver

Instantiate a trust-region subproblem solver based on the Generalized Lanczos iterative method of Gould, Lucidi, Roma and Toint. See pygltr for more information.

Solve()
Solve the trust-region subproblem using the generalized Lanczos method.

6.3 Complete Solvers

6.3.1 Linear Least-Squares Problems

Solve the least-squares problem

minimize ||Ax-b||

using LSQR. This is a line-by-line translation from Matlab code available at http://www.stanford.edu/~saunders/lsqr.

Michael P. Friedlander, University of British Columbia Dominique Orban, Ecole Polytechnique de Montreal

class lsqr.**LSQRFramework**(*m*, *n*, *aprod*)
LSQR solves *Ax = b* or *minimize |b - Ax|* in Euclidian norm if *damp = 0*, or *minimize |b - Ax| + damp * |x|* in Euclidian norm if *damp > 0*.

A is an (m x n) matrix defined by *y = aprod(mode, m, n, x)*, where *aprod* refers to a function that performs matrix-vector products. If *mode* = 1, *aprod* must return *y = Ax* without altering *x*. If *mode* = 2, *aprod* must return *y = A'x* without altering *x*.

LSQR uses an iterative (conjugate-gradient-like) method.

For further information, see

1. C. C. Paige and M. A. Saunders (1982a). LSQR: An algorithm for sparse linear equations and sparse least squares, ACM TOMS 8(1), 43-71.
2. C. C. Paige and M. A. Saunders (1982b). Algorithm 583. LSQR: Sparse linear equations and least squares problems, ACM TOMS 8(2), 195-209.
3. M. A. Saunders (1995). Solution of sparse rectangular systems using LSQR and CRAIG, BIT 35, 588-604.

solve(*rhs*, *itnlim=0*, *damp=0.0*, *atol=9.9999999999999995e-07*, *btol=9.9999999999999995e-08*, *conlim=100000000.0*, *show=False*, *wantvar=False*)

Solve the linear system, linear least-squares problem or regularized linear least-squares problem with specified parameters. All return values below are stored in members of the same name.

Parameters

rhs right-hand side vector.

itnlim is an explicit limit on iterations (for safety).

damp damping/regularization parameter.

Keywords

atol

btol are stopping tolerances. If both are 1.0e-9 (say), the final residual norm should be accurate to about 9 digits. (The final x will usually have fewer correct digits, depending on *cond(A)* and the size of *damp*.)

conlim is also a stopping tolerance. lsqr terminates if an estimate of *cond(A)* exceeds *conlim*. For compatible systems *Ax* = *b*, *conlim* could be as large as 1.0e+12 (say). For least-squares problems, *conlim* should be less than 1.0e+8. Maximum precision can be obtained by setting *atol* = *btol* = *conlim* = zero, but the number of iterations may then be excessive.

show if set to *True*, gives an iteration log. If set to *False*, suppresses output.

Returns

x is the final solution.

istop gives the reason for termination.

istop = 1 means x is an approximate solution to Ax = b. = 2 means x approximately solves the least-squares problem.

r1norm = norm(r), where r = b - Ax.

r2norm = sqrt(norm(r)^2 + damp^2 * norm(x)^2) = r1norm if damp = 0.

anorm = estimate of Frobenius norm of (regularized) A.

acond = estimate of cond(Abar).

arnorm = estimate of norm(A'r - damp^2 x).

xnorm = norm(x).

var (if present) estimates all diagonals of (A'A)^{-1} (if damp=0) or more generally (A'A + damp^2*I)^{-1}. This is well defined if A has full column rank or damp > 0. (Not sure what var means if rank(A) < n and damp = 0.)

6.3.2 Linear Programming

class lp.**RegLPInteriorPointSolver**(*lp*, ***kwargs*)

display_stats()

Display vital statistics about the input problem.

maxStepLength(*x*, *d*)

Returns the max step length from x to the boundary of the nonnegative orthant in the direction d.

scale(***kwargs*)

Equilibrate the constraint matrix of the linear program. Equilibration is done by first dividing every row by its largest element in absolute value and then by dividing every column by its largest element in absolute value. In effect the original problem

minimize c'x subject to A1 x + A2 s = b, x >= 0

is converted to

minimize (Cc)'x subject to R A1 C x + R A2 C s = Rb, x >= 0,

where the diagonal matrices R and C operate row and column scaling respectively.

Upon return, the matrix A and the right-hand side b are scaled and the members *row_scale* and *col_scale* are set to the row and column scaling factors.

The scaling may be undone by subsequently calling `unscale()`. It is necessary to unscale the problem in order to unscale the final dual variables. Normally, the `solve()` method takes care of unscaling the problem upon termination.

set_initial_guess(*lp*, ***kwargs*)

Compute initial guess according the Mehrotra's heuristic. Initial values of x are computed as the solution to the least-squares problem

minimize ||s|| subject to A1 x + A2 s = b

which is also the solution to the augmented system

[0 0 A1'] [x] [0] [0 I A2'] [s] = [0] [A1 A2 0] [w] [b].

Initial values for (y,z) are chosen as the solution to the least-squares problem

minimize ||z|| subject to A1' y = c, A2' y + z = 0

which can be computed as the solution to the augmented system

[0 0 A1'] [w] [c] [0 I A2'] [z] = [0] [A1 A2 0] [y] [0].

To ensure stability and nonsingularity when A does not have full row rank, the (1,1) block is perturbed to 1.0e-4 * I and the (3,3) block is perturbed to -1.0e-4 * I.

The values of s and z are subsequently adjusted to ensure they are positive. See [Methrotra, 1992] for details.

solve(***kwargs*)

Solve the input problem with the primal-dual-regularized interior-point method. Accepted input keyword arguments are

Keywords

itermax The maximum allowed number of iterations (default: 10n)

tolerance Stopping tolerance (default: 1.0e-6)

PredictorCorrector Use the predictor-corrector method (default: *True*). If set to *False*, a variant of the long-step method is used. The long-step method is generally slower and less robust.

Upon exit, the following members of the class instance are set:

x.....final iterate y.....final value of the Lagrange multipliers associated

to A1 x + A2 s = b

z.....final value of the Lagrange multipliers associated to s>=0

obj_value.....final cost iter.....total number of iterations kktResid.....final relative residual solve_time.....time to solve the LP status.....string describing the exit status short_status...short version of status, used for printing.

solveSystem(*rhs*, *itref_threshold=1.0000000000000001e-05*, *nitrefmax=3*)

unscale(***kwargs*)
Restore the constraint matrix A, the right-hand side b and the cost vector c to their original value by undoing the row and column equilibration scaling.

class lp.**RegLPInteriorPointSolver29**(*lp*, ***kwargs*)
Bases: lp.RegLPInteriorPointSolver

display_stats()
Display vital statistics about the input problem.

maxStepLength(*x*, *d*)
Returns the max step length from x to the boundary of the nonnegative orthant in the direction d.

scale(***kwargs*)
Scale the constraint matrix of the linear program. The scaling is done so that the scaled matrix has all its entries near 1.0 in the sense that the square of the sum of the logarithms of the entries is minimized.

In effect the original problem

minimize c'x subject to A1 x + A2 s = b, x >= 0

is converted to

minimize (Cc)'x subject to R A1 C x + R A2 C s = Rb, x >= 0,

where the diagonal matrices R and C operate row and column scaling respectively.

Upon return, the matrix A and the right-hand side b are scaled and the members *row_scale* and *col_scale* are set to the row and column scaling factors.

The scaling may be undone by subsequently calling `unscale()`. It is necessary to unscale the problem in order to unscale the final dual variables. Normally, the `solve()` method takes care of unscaling the problem upon termination.

set_initial_guess(*lp*, ***kwargs*)
Compute initial guess according the Mehrotra's heuristic. Initial values of x are computed as the solution to the least-squares problem

minimize ||s|| subject to A1 x + A2 s = b

which is also the solution to the augmented system

[0 0 A1'] [x] [0] [0 I A2'] [s] = [0] [A1 A2 0] [w] [b].

Initial values for (y,z) are chosen as the solution to the least-squares problem

minimize ||z|| subject to A1' y = c, A2' y + z = 0

which can be computed as the solution to the augmented system

[0 0 A1'] [w] [c] [0 I A2'] [z] = [0] [A1 A2 0] [y] [0].

To ensure stability and nonsingularity when A does not have full row rank, the (1,1) block is perturbed to 1.0e-4 * I and the (3,3) block is perturbed to -1.0e-4 * I.

The values of s and z are subsequently adjusted to ensure they are positive. See [Methrotra, 1992] for details.

solve(***kwargs*)
Solve the input problem with the primal-dual-regularized interior-point method. Accepted input keyword arguments are

Keywords

itermax The maximum allowed number of iterations (default: 10n)

tolerance Stopping tolerance (default: 1.0e-6)

PredictorCorrector Use the predictor-corrector method (default: *True*). If set to *False*, a variant of the long-step method is used. The long-step method is generally slower and less robust.

Upon exit, the following members of the class instance are set:

x.....final iterate y.....final value of the Lagrange multipliers associated

to A1 x + A2 s = b

z.....final value of the Lagrange multipliers associated to s>=0

obj_value.....final cost iter.....total number of iterations kktResid.....final relative residual solve_time.....time to solve the LP status.....string describing the exit status short_status...short version of status, used for printing.

solveSystem(*rhs*, *itref_threshold=1.0000000000000001e-05*, *nitrefmax=3*)

unscale(***kwargs*)
Restore the constraint matrix A, the right-hand side b and the cost vector c to their original value by undoing the row and column equilibration scaling.

6.3.3 Convex Quadratic Programming

class cqp.**RegQPInteriorPointSolver**(*qp*, ***kwargs*)

display_stats()
Display vital statistics about the input problem.

maxStepLength(*x*, *d*)
Returns the max step length from x to the boundary of the nonnegative orthant in the direction d.

scale(***kwargs*)
Equilibrate the constraint matrix of the linear program. Equilibration is done by first dividing every row by its largest element in absolute value and then by dividing every column by its largest element in absolute value. In effect the original problem

minimize c' x + 1/2 x' Q x subject to A1 x + A2 s = b, x >= 0

is converted to

minimize (Cc)' x + 1/2 x' (CQC') x subject to R A1 C x + R A2 C s = Rb, x >= 0,

where the diagonal matrices R and C operate row and column scaling respectively.

Upon return, the matrix A and the right-hand side b are scaled and the members *row_scale* and *col_scale* are set to the row and column scaling factors.

The scaling may be undone by subsequently calling `unscale()`. It is necessary to unscale the problem in order to unscale the final dual variables. Normally, the `solve()` method takes care of unscaling the problem upon termination.

set_initial_guess(*qp*, ***kwargs*)

Compute initial guess according the Mehrotra's heuristic. Initial values of x are computed as the solution to the least-squares problem

minimize ||s|| subject to A1 x + A2 s = b

which is also the solution to the augmented system

[0 0 A1'] [x] [0] [0 I A2'] [s] = [0] [A1 A2 0] [w] [b].

Initial values for (y,z) are chosen as the solution to the least-squares problem

minimize ||z|| subject to A1' y = c, A2' y + z = 0

which can be computed as the solution to the augmented system

[0 0 A1'] [w] [c] [0 I A2'] [z] = [0] [A1 A2 0] [y] [0].

To ensure stability and nonsingularity when A does not have full row rank, the (1,1) block is perturbed to 1.0e-4 * I and the (3,3) block is perturbed to -1.0e-4 * I.

The values of s and z are subsequently adjusted to ensure they are positive. See [Methrotra, 1992] for details.

solve(***kwargs*)

Solve the input problem with the primal-dual-regularized interior-point method. Accepted input keyword arguments are

Keywords

itermax The maximum allowed number of iterations (default: 10n)

tolerance Stopping tolerance (default: 1.0e-6)

PredictorCorrector Use the predictor-corrector method (default: *True*). If set to *False*, a variant of the long-step method is used. The long-step method is generally slower and less robust.

Upon exit, the following members of the class instance are set:

- x.....final iterate
- **y.....final value of the Lagrange multipliers associated** to A1 x + A2 s = b
- **z.....final value of the Lagrange multipliers associated** to s>=0
- obj_value.....final cost
- iter.....total number of iterations
- kktResid.....final relative residual
- solve_time.....time to solve the QP
- status.....string describing the exit status.
- short_status...short version of status, used for printing.

solveSystem(*rhs*, *itref_threshold=1.0000000000000001e-05*, *nitrefmax=5*)

unscale(***kwargs*)

Restore the constraint matrix A, the right-hand side b and the cost vector c to their original value by undoing the row and column equilibration scaling.

class cqp.**RegQPInteriorPointSolver29**(*qp*, ***kwargs*)

Bases: cqp.RegQPInteriorPointSolver

display_stats()
Display vital statistics about the input problem.

maxStepLength(*x*, *d*)
Returns the max step length from x to the boundary of the nonnegative orthant in the direction d.

scale(***kwargs*)
Scale the constraint matrix of the linear program. The scaling is done so that the scaled matrix has all its entries near 1.0 in the sense that the square of the sum of the logarithms of the entries is minimized.

In effect the original problem

minimize c'x + 1/2 x'Qx subject to A1 x + A2 s = b, x >= 0

is converted to

minimize (Cc)'x + 1/2 x' (CQC') x subject to R A1 C x + R A2 C s = Rb, x >= 0,

where the diagonal matrices R and C operate row and column scaling respectively.

Upon return, the matrix A and the right-hand side b are scaled and the members *row_scale* and *col_scale* are set to the row and column scaling factors.

The scaling may be undone by subsequently calling `unscale()`. It is necessary to unscale the problem in order to unscale the final dual variables. Normally, the `solve()` method takes care of unscaling the problem upon termination.

set_initial_guess(*qp*, ***kwargs*)
Compute initial guess according the Mehrotra's heuristic. Initial values of x are computed as the solution to the least-squares problem

minimize ||s|| subject to A1 x + A2 s = b

which is also the solution to the augmented system

[0 0 A1'] [x] [0] [0 I A2'] [s] = [0] [A1 A2 0] [w] [b].

Initial values for (y,z) are chosen as the solution to the least-squares problem

minimize ||z|| subject to A1' y = c, A2' y + z = 0

which can be computed as the solution to the augmented system

[0 0 A1'] [w] [c] [0 I A2'] [z] = [0] [A1 A2 0] [y] [0].

To ensure stability and nonsingularity when A does not have full row rank, the (1,1) block is perturbed to 1.0e-4 * I and the (3,3) block is perturbed to -1.0e-4 * I.

The values of s and z are subsequently adjusted to ensure they are positive. See [Methrotra, 1992] for details.

solve(***kwargs*)
Solve the input problem with the primal-dual-regularized interior-point method. Accepted input keyword arguments are

Keywords

itermax The maximum allowed number of iterations (default: 10n)

tolerance Stopping tolerance (default: 1.0e-6)

PredictorCorrector Use the predictor-corrector method (default: *True*). If set to *False*, a variant of the long-step method is used. The long-step method is generally slower and less robust.

Upon exit, the following members of the class instance are set:

- x.....final iterate
- **y.....final value of the Lagrange multipliers associated** to A1 x + A2 s = b
- **z.....final value of the Lagrange multipliers associated** to s>=0
- obj_value.....final cost
- iter.....total number of iterations
- kktResid.....final relative residual
- solve_time.....time to solve the QP
- status.....string describing the exit status.
- short_status...short version of status, used for printing.

solveSystem(*rhs*, *itref_threshold=1.0000000000000001e-05*, *nitrefmax=5*)

unscale(***kwargs*)

Restore the constraint matrix A, the right-hand side b and the cost vector c to their original value by undoing the row and column equilibration scaling.

6.3.4 Unconstrained Programming

TRUNK Trust-Region Method for Unconstrained Programming.

A first unconstrained optimization solver in Python The Python version of the celebrated F90/95 solver D. Orban Montreal Sept. 2003

class `trunk.`**TrunkFramework**(*nlp*, *TR*, *TrSolver*, ***kwargs*)

An abstract framework for a trust-region-based algorithm for nonlinear unconstrained programming. Instantiate using

TRNK = TrunkFramework(nlp, TR, TrSolver)

Parameters

nlp a `NLPModel` object representing the problem. For instance, nlp may arise from an AMPL model

TR a `TrustRegionFramework` object

TrSolver a `TrustRegionSolver` object.

Keywords

x0 starting point (default nlp.x0)

reltol relative stopping tolerance (default *nlp.stop_d*)

abstol absolute stopping tolerance (default 1.0e-6)

maxiter maximum number of iterations (default max(1000,10n))

inexact use inexact Newton stopping tol (default False)

ny apply Nocedal/Yuan linesearch (default False)

nbk max number of backtracking steps in Nocedal/Yuan linesearch (default 5)

monotone use monotone descent strategy (default False)

nIterNonMono number of iterations for which non-strict descent can be tolerated if monotone=False (default 25)

silent verbosity level (default False)

Once a *TrunkFramework* object has been instantiated and the problem is set up, solve problem by issuing a call to *TRNK.solve()*. The algorithm stops as soon as the Euclidian norm of the gradient falls below

```
max(abstol, reltol * g0)
```

where `g0` is the Euclidian norm of the gradient at the initial point.

PostIteration(***kwargs*)
Override this method to perform work at the end of an iteration. For example, use this method for preconditioners that need updating, e.g., a limited-memory BFGS preconditioner.

Precon(*v*, ***kwargs*)
Generic preconditioning method—must be overridden

Solve(***kwargs*)

class `trunk.`**TrunkLbfgsFramework**(*nlp*, *TR*, *TrSolver*, ***kwargs*)
Bases: `trunk.TrunkFramework`

Class TrunkLbfgsFramework is a subclass of TrunkFramework. The method is based on the same trust-region algorithm with Nocedal-Yuan backtracking. The only difference is that a limited-memory BFGS preconditioner is used and maintained along the iterations. See class TrunkFramework for more information.

PostIteration(***kwargs*)
This method updates the limited-memory BFGS preconditioner by appending the most rencet (s,y) pair to it and possibly discarding the oldest one if all the memory has been used.

Precon(*v*, ***kwargs*)
This method implements limited-memory BFGS preconditioning. It overrides the default Precon() of class TrunkFramework.

Solve(***kwargs*)

class `lbfgs.`**InverseLBFGS**(*n*, *npairs=5*, ***kwargs*)
Class InverseLBFGS is a container used to store and manipulate limited-memory BFGS matrices. It may be used, e.g., in a LBFGS solver for unconstrained minimization or as a preconditioner. The limited-memory matrix that is implicitly stored is a positive definite approximation to the inverse Hessian. Therefore, search directions may be obtained by computing matrix-vector products only. Such products are efficiently computed by means of a two-loop recursion.

Instantiation is as follows

lbfgsupdate = InverseLBFGS(n)

where n is the number of variables of the problem.

Keywords

npairs the number of (s,y) pairs stored (default: 5)

scaling enable scaling of the 'initial matrix'. Scaling is done as 'method M3' in the LBFGS paper by Zhou and Nocedal; the scaling factor is <sk,yk>/<yk,yk> (default: False).

Member functions are

- **store to store a new (s,y) pair and discard the oldest one** in case the maximum storage has been reached,
- **matvec to compute a matrix-vector product between the current** positive-definite approximation to the inverse Hessian and a given vector.

matvec(*iter*, *v*)
Compute a matrix-vector product between the current limited-memory positive-definite approximation to the inverse Hessian matrix and the vector v using the LBFGS two-loop recursion formula. The 'iter' argument is the current iteration number.

When the inner product <y,s> of one of the pairs is nearly zero, the function returns the input vector v, i.e., no preconditioning occurs. In this case, a safeguarding step should probably be taken.

solve(*iter*, *v*)
This is an alias for matvec used for preconditioning.

store(*iter*, *new_s*, *new_y*)
Store the new pair (new_s,new_y) computed at iteration iter.

class lbfgs.**LBFGSFramework**(*nlp*, ***kwargs*)
Class LBFGSFramework provides a framework for solving unconstrained optimization problems by means of the limited-memory BFGS method.

Instantiation is done by

lbfgs = LBFGSFramework(nlp)

where nlp is an instance of a nonlinear problem. A solution of the problem is obtained by called the solve member function, as in

lbfgs.solve().

Keywords

npairs the number of (s,y) pairs to store (default: 5)

x0 the starting point (default: nlp.x0)

maxiter the maximum number of iterations (default: max(10n,1000))

abstol absolute stopping tolerance (default: 1.0e-6)

reltol relative stopping tolerance (default: *nlp.stop_d*)

Other keyword arguments will be passed to InverseLBFGS.

The linesearch used in this version is Jorge Nocedal's modified More and Thuente linesearch, attempting to ensure satisfaction of the strong Wolfe conditions. The modifications attempt to limit the effects of rounding error inherent to the More and Thuente linesearch.

solve()

6.3.5 Bound-Constrained Programming

This module implements a purely primal-dual interior-point methods for bound-constrained optimization. The method uses the primal-dual merit function of Forsgren and Gill and solves subproblems by means of a truncated conjugate gradient method with trust region.

References:

[1] A. Forsgren and Ph. E. Gill, Primal-Dual Interior Methods for Nonconvex Nonlinear Programming, SIAM Journal on Optimization, 8(4):1132-1152, 1998

[2] Ph. E. Gill and E. M. Gertz, A primal-dual trust region algorithm for nonlinear optimization, Mathematical Programming, 100(1):49-94, 2004

[3] P. Armand and D. Orban, A Trust-Region Interior-Point Method for Bound-Constrained Programming Based on a Primal-Dual Merit Function, Cahier du GERAD G-xxxx, GERAD, Montreal, Quebec, Canada, 2008.

4. Orban, Montreal

class pdmerit.**PrimalDualInteriorPointFramework**(*nlp*, *TR*, *TrSolver*, ***kwargs*)

AtOptimality(*x*, *z*, ***kwargs*)
Shortcut.

GradPDMerit(*x*, *z*, ***kwargs*)
Evaluate the gradient of the primal-dual merit function at (x,z). See PDMerit() for a description of *z*.

HessProd(*x*, *z*, *p*, ***kwargs*)
Compute the matrix-vector product between the Hessian matrix of the primal-dual merit function at (x,z) and the vector p. See help(PDMerit) for a description of z. If there are b bounded variables and q two-sided bounds, the vector p should have length n+b+2q. The Hessian matrix has the general form

[H + 2 mu X^{-2} I] [I mu Z^{-2}].

PDHess(*x*, *z*, ***kwargs*)
Assemble the modified Hessian matrix of the primal-dual merit function at (x,z). See PDMerit() for a description of *z*. The Hessian matrix has the general form

[H + 2 X^{-1} Z I] [I Z^{-1} X].

PDHessProd(*x*, *z*, *p*, ***kwargs*)
Compute the matrix-vector product between the modified Hessian matrix of the primal-dual merit function at (x,z) and the vector p. See help(PDMerit) for a description of z. If there are b bounded variables and q two-sided bounds, the vector p should have length n+b+2q. The Hessian matrix has the general form

[H + 2 X^{-1} Z I] [I Z^{-1} X].

PDHessTemplate(***kwargs*)
Assemble the part of the modified Hessian matrix of the primal-dual merit function that is iteration independent. The function PDHess() fills in the blanks by updating the rest of the matrix.

PDMerit(*x*, *z*, ***kwargs*)
Evaluate the primal-dual merit function at (x,z). If there are b >= 0 one-sided bound constraints and q >= 0 two-sided bound constraints, the vector z should have length b+2q. The components z[i] (0 <= i < b+q) correspond, in increasing order of variable indices to

- variables with a lower bound only,
- variables with an upper bound only,
- the lower bound on variables that are bounded below and above.

The components z[i] (b+q <= i < b+2q) correspond to the upper bounds on variables that are bounded below and above.

This function returns the value of the primal-dual merit function. The current value of the objective function can be supplied via the keyword argument *f*.

Precon(*v*, ***kwargs*)
Generic preconditioning method—must be overridden.

PrimalMultipliers(*x*, ***kwargs*)
Return the vector of primal multipliers at *x*. The value of the barrier parameter used can either be supplied using the keyword argument *mu* or the current value of the instance is used.

SetupPrecon(***kwargs*)
Construct or set up the preconditioner—must be overridden.

SolveInner (***kwargs*)
Perform a series of inner iterations so as to minimize the primal-dual merit function with the current value of the barrier parameter to within some given tolerance. The only optional argument recognized is

stopTol stopping tolerance (default: muerrfact * mu).

SolveOuter (***kwargs*)

StartingPoint (***kwargs*)
Compute a strictly feasible initial primal-dual estimate (x0, z0).

UpdateMu (***kwargs*)
Update the barrier parameter before the next round of inner iterations.

UpdatePrecon (***kwargs*)
Override this method for preconditioners that need updating, e.g., a limited-memory BFGS preconditioner.

ftb (*x*, *z*, *step*, ***kwargs*)

Compute the largest alpha in]0,1] such that (x,z) + alpha * step >= (1 - tau) * (x,z)

where 0 < tau < 1. By default, tau = 0.9.

6.3.6 General Nonlinear Programming

Todo

Insert this module.

Some pages of this documentation display equations via the jsMath package. They should look reasonably good with most setups but the best rendering is obtained by installing the TeX fonts. Please refer to http://www.math.union.edu/~dpvc/jsMath/users/welcome.html.

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[Kel95] C. T. Kelley, *Iterative Methods for Linear and Nonlinear Equations*, SIAM, Philadelphia, PA, 1995.

[Orb08] D. Orban, *Projected Krylov Methods for Unsymmetric Augmented Systems*, Cahiers du GERAD G-2008-46, GERAD, Montreal, Canada, 2008.

PYTHON MODULE INDEX

INDEX

J

L

M

N

O

P

R

S

T

U

W