User guide

Python optimization algorithms

The nlpalg library

Support:

This work was supported by ENACOM.

Updates

02/07/2018 Adriano Lisboa

i. initial version

05/07/2018 Pedro Ribeiro

i. fixed compilation command and report title
ii. importing instructions subsection

06/07/2018 Adriano Lisboa

i. science optimization library example

19/07/2018 Adriano Lisboa

i. shallow cut and log options for ellipsoid method

23/07/2018 Adriano Lisboa

i. decomposition and memory options for ellipsoid method
ii. ellipsoid matrix input and output arguments
iii. output arguments for ellipsoid method

24/07/2018 Adriano Lisboa

i. complete path of ellipsoid method

25/07/2018 Adriano Lisboa

i. convergence return of ellipsoid method

27/07/2018 Adriano Lisboa

i. stop criterion flag for ellipsoid method
ii. nlpalg performance

Gaia

Contents

1 Library

The optimization algorithm library is called “nlpalg” which stands for “nonlinear programming algorithms”. It is written in C++ and binded to Python using the library “pybind11”, which requires C++ version 11 and is tested in Python 3.6. It is basically composed by several nonlinear optimization algorithms.

1.1 PyBind11

To compile and use the "nlpalg", the library "pybind11" must be installed. To install it, the github page have the instructions to compile and install the package. Another simpler way is to use the command bellow:

```
pip install pybind11
```

1.2 Compilation

The output of the compilation must be a dynamic library file named “nlpalg.pyd” on Windows and “nlpalg.so” on Linux. The compilation must include “Python” and “pybind11” libraries and the compiler must support C++ version 11. The Visual Studio 2017 project files are provided for compilation on Windows. For compilation on Linux, the command line

```
g++ -O3 -Wall -shared -std=c++11 -fPIC `python3 -m pybind11
--includes` -I./ -I/usr/include/pybind11/ nlpalg.cpp -o nlpalg`python3-config --extension-suffix`
```

can be used.

2 Ellipsoid method

The implemented C++ ellipsoid method returns a solution $x^{\star}$ that equals or dominates the starting point x_0 in case x_0 is feasible for the optimization problem in the form

$$\text{minimize} \quad f(x) \tag{1}$$
$$\text{subject to} \quad g(x) \leq 0 \tag{2}$$
$$Ax \leq b \tag{3}$$
$$A_{eq}x = b_{eq} \tag{4}$$
$$x_{\min} \leq x \leq x_{\max} \tag{5}$$

where $x \in \mathbb{R}^n$ are the design variables for $n \in \{2, 3, ...\}$, $f : \mathbb{R}^n \mapsto \mathbb{R}^o$ are the objective functions for $o \in \{1, 2, ...\}$, $g : \mathbb{R}^n \mapsto \mathbb{R}^{m'}$ are the inequality constraint functions for

$m' \in \{0, 1, ...\}$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ form the linear inequality constraints for $m \in \{0, 1, ...\}$, $A_{eq} \in \mathbb{R}^{p \times n}$ and $b_{eq} \in \mathbb{R}^p$ form the linear equality constraints for $p \in \{0, 1, ...\}$, $x_{\min} \in \mathbb{R}^n$ and $x_{\max} \in \mathbb{R}^n$ are the respective lower and upper bounds. The solution $x^\star$ is empty if the problem is infeasible. It uses multiple cuts and guarantees that the oracles f and g will only be queried where the linear inequality constraints (3) and (5) are satisfied.

2.1 Prototype

The ellipsoid method in Python has the prototype

```
[xb, fxb, x, fx, Qi, stop] = nlpalg.ellipsoidmethod(f, df, g, dg, A, b, Aeq, beq, xmin, xmax, x0, Qi0,
                                                    epsilon, kmax, kimax, shallowcut, decomposition, memory, log)
```

where "`f`" is the objective function which should be minimized, "`df`" is the objective gradient function, "`g`" is the inequality constraint function $g(x) \leq 0$, "`dg`" is the inequality constraint gradient function, "`A`" and "`b`" form the linear inequality constraints $Ax \leq b$, "`Aeq`" and "`beq`" form the linear equality constraints $A_{eq}x = b_{eq}$, "`xmin`" and "`xmax`" form the bounds $x_{\min} \leq x \leq x_{\max}$, "`x0`" is the starting point $x_0 \in \mathbb{R}^n$ (centered start $x_0 = (x_{\min} + x_{\max})/2$ if empty), "`Qi0`" is the starting inverse ellipsoid matrix (tight start if empty), "`epsilon`" is the maximum uncertainty for stop criterion (0 for no stopping on this criterion), "`kmax`" is the maximum number of iterations for stop criterion, "`kimax`" is the maximum number of cuts per iteration, "`shallowcut`" is the index of shallow cuts usage $\in [0, 1]$, "`decomposition`" is matrix decomposition indicator, "`memory`" is cut memory indicator, "`log`" is the feasible path log logical indicator. It returns the optimal point "`xb`" (or path if "`log = True`") and respective function values "`fxb`", and the final point "`x`" (or path if "`log = True`") and respective function values "`fx`" and inverse ellipsoid matrix "`Qi`". The "`stop`" flag indicates the stop reason: 0 for stop by maximum number of iterations, 1 for stop by ellipsoid volume reduction, 2 for stop by empty localizing set, 3 for stop by degenerate ellipsoid.

2.2 Examples

2.2.1 Linear constraints

Consider the optimization problem with linear constraints

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}(x-c)^T Q(x-c) \\ \text{subject to} \quad & x_1 + x_2 \leq 10 \\ & x_1 = x_2 \\ & -10 \leq x \leq 10 \end{aligned}$$

where

$$Q = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}, \quad c = \begin{bmatrix} 3 \\ 5 \\ 40 \end{bmatrix}$$

Its solution lies on the line $x_1 = x_2 = (c_1 + c_2)/2 = 4$ at $x_3 = 10$, which can be verified using the following Python code.

```
# libraries
import nlpalg # nonlinear programming algorithm library
import numpy as np

# quadratic objective function
xf = np.array([3, 5, 40]).reshape(-1, 1)
Af = 2*np.identity(3)
bf = -np.matmul(Af, xf)
cf = .5*np.matmul(np.transpose(xf), np.matmul(Af, xf))
f = lambda x: [.5*np.matmul(np.transpose(x), np.matmul(Af, x)) + np.matmul(np.transpose(x), bf) + cf]
df = lambda x: np.matmul(Af, x) + bf

# empty nonlinear inequality constraint
g = lambda x: np.zeros((0, 1))
dg = lambda x: np.zeros((3, 0))

# linear inequality constraint
A = np.array([[1, 1, 0]])
b = np.array([10])

# linear inequality constraint
Aeq = np.array([[1, -1, 0]])
beq = np.array([0])

# bounds
xmin = np.array([-10, -10, -10]).reshape(-1, 1) # lower
xmax = np.array([10, 10, 10]).reshape(-1, 1) # upper

# solution with ellipsoid method
x0 = np.array([[]]) # starting point
Qi0 = np.array([[]]) # starting inverse ellipsoid matrix
epsilon = 0 # uncertainty on each variable
kmax = 300 # maximum number of iterations
kimax = 32 # maximum number of cuts per iteration
shallowcut = 0 # use of shallow cuts [0, 1]
decomposition = True # square root decomposition of ellipsoid matrix indicator
memory = True # cut memorization through iterations indicator
log = True # path log indicator
[xb, fxb, x, fx, Qi, stop] = nlpalg.ellipsoidmethod(f, df, g, dg, A, b, Aeq, beq, xmin, xmax, x0, Qi0,
                                              epsilon, kmax, kimax, shallowcut, decomposition, memory, log)

# print solution
print("Stop criterion", stop)
print(xb)
print(fxb)
print(Qi[...,-1])
```

2.2.2 Multiobjective nonlinear programming

Consider the biobjective nonlinear quadratic optimization problem

$$\begin{aligned}
\text{minimize} \quad & f(x) = \begin{bmatrix} \frac{1}{2}(x-c_1)^T Q_1 (x-c_1) \\ \frac{1}{2}(x-c_2)^T Q_2 (x-c_2) \end{bmatrix} \\
\text{subject to} \quad & \frac{1}{2}(x-c_3)^T Q_3 (x-c_3) \leq 1 \\
& \frac{1}{2}(x-c_4)^T Q_4 (x-c_4) \leq 1 \\
& -10 \leq x \leq 10
\end{aligned}$$

where

$$Q_1 = Q_3 = Q_4 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}, \quad Q_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 4 \end{bmatrix}, \quad c_1 = -c_2 = c_4 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad c_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The optimality condition at the solution of this problem is

$$\frac{\nabla f_1(x)}{\|\nabla f_1(x)\|} = -\frac{\nabla f_2(x)}{\|\nabla f_2(x)\|}$$

which can be verified with the following Python code.

```python
# libraries
import nlpalg # nonlinear programming algorithm library
import numpy as np

# quadratic objective functions
xf = np.array([1, 1, 1]).reshape(-1, 1)
Af = 2*np.identity(3)
bf = -np.matmul(Af, xf)
cf = .5*np.matmul(np.transpose(xf), np.matmul(Af, xf))
xf2 = np.array([-1, -1, -1]).reshape(-1, 1)
Af2 = np.diag([1, 2, 4])
bf2 = -np.matmul(Af2, xf2)
cf2 = .5*np.matmul(np.transpose(xf2), np.matmul(Af2, xf2))
f = lambda x: np.vstack((
    .5*np.matmul(np.transpose(x), np.matmul(Af, x)) + np.matmul(np.transpose(x), bf) + cf,
    .5*np.matmul(np.transpose(x), np.matmul(Af2, x)) + np.matmul(np.transpose(x), bf2) + cf2))
df = lambda x: np.hstack((np.matmul(Af, x) + bf, np.matmul(Af2, x) + bf2))

# quadratic inequality constraint functions
Ag = 2*np.identity(3)
bg = np.zeros((3, 1))
cg = -1
xg2 = np.array([1, 1, 1]).reshape(-1, 1)
Ag2 = 2*np.identity(3)
bg2 = -np.matmul(Ag2, xg2)
cg2 = .5*np.matmul(np.transpose(xg2), np.matmul(Ag2, xg2)) - 1
g = lambda x: np.vstack((
    .5*np.matmul(np.transpose(x), np.matmul(Ag, x)) + np.matmul(np.transpose(x), bg) + cg,
    .5*np.matmul(np.transpose(x), np.matmul(Ag2, x)) + np.matmul(np.transpose(x), bg2) + cg2))
dg = lambda x: np.hstack((np.matmul(Ag, x) + bg, np.matmul(Ag2, x) + bg2))
```

```
# empty linear inequality constraints
A = np.zeros((0, 3))
b = np.zeros((0, 1))

# empty linear equality constraints
Aeq = np.zeros((0, 3))
beq = np.zeros((0, 1))

# bounds
xmin = np.array([[-10], [-10], [-10]]) # lower
xmax = np.array([[10], [10], [10]]) # uppder

# solution
x0 = np.array([[20], [20], [20]]) # starting point
Qi0 = np.array([[]]) # starting inverse ellipsoid matrix
epsilon = 0 # uncertainty on each variable
kmax = 300 # maximum number of iterations
kimax = 32 # maximum number of cuts per iteration
shallowcut = 0 # use of shallow cuts [0, 1]
decomposition = True # square root decomposition of ellipsoid matrix indicator
memory = True # cut memorization through iterations indicator
log = False # path log indicator
[xb, fxb, x0, fx, Qi0, stop] = nlpalg.ellipsoidmethod(f, df, g, dg, A, b, Aeq, beq, xmin, xmax, x0, Qi0,
                                                epsilon, 10, kimax, shallowcut, decomposition, memory, log)
[xb, fxb, x, fx, Qi, stop] = nlpalg.ellipsoidmethod(f, df, g, dg, A, b, Aeq, beq, xmin, xmax, x0, Qi0,
                                              epsilon, kmax, kimax, shallowcut, decomposition, memory, log)

# optimality condition: opposite objective gradient directions
print(xb)
if (xb.size):
    dfx = df(xb)
    dfx[...,0] = dfx[...,0]/np.sqrt(np.sum(dfx[...,0]*dfx[...,0]))
    dfx[...,1] = dfx[...,1]/np.sqrt(np.sum(dfx[...,1]*dfx[...,1]))
    print(dfx)
```

2.2.3 Science optimization library

Consider the optimization problem

$$\begin{aligned} \text{minimize} \quad & (x_1 - 1)^2 + 4x_2^2 \\ \text{subject to} \quad & -5 \le x \le 5 \end{aligned}$$

whose solution is $(1, 0)$, as can be verified with the following Python code using the science optimization library.

```
# libraries
import nlpalg # nonlinear programming algorithm library
import numpy as np
from science_optimization.builder import OptimizationProblem
from science_optimization.function import QuadraticFunction
from science_optimization.problems import GenericProblem

# Problem: (x[0]-1)^2 + 4.0*x[1]^2
Q = np.array([[1, 0], [0, 4]])
c = np.array([-2, 0]).reshape(-1, 1)
d = 1
f = [QuadraticFunction(Q=Q, c=c, d=d)]
x_lim = np.hstack((np.array([[-5 ], [-5]]), np.array([[5 ], [5]]))) # bounds
```

```python
op = OptimizationProblem(builder=GenericProblem(f=f, eq_cons=[], ineq_cons=[], x_lim=x_lim))

# solution
f = lambda x: op.objective.objective_functions.eval(x) # objective function
df = lambda x: op.objective.objective_functions.gradient(x) # objective gradient function
g = lambda x: np.zeros((0, 1)) # empty constraints
dg = lambda x: np.zeros((2, 0))
A = np.zeros((0, 2)) # empty linear inequality constraints
b = np.zeros((0, 1))
Aeq = np.zeros((0, 2)) # empty linear equality constraints
beq = np.zeros((0, 1))
xmin = op.variables.x_min() # lower bound
xmax = op.variables.x_max() # upper bound
x0 = (xmin + xmax)/2 # starting point
Qi0 = np.array([[]]) # starting inverse ellipsoid matrix
epsilon = 0 # uncertainty on each variable
kmax = 300 # maximum number of iterations
kimax = 32 # maximum number of cuts per iteration
shallowcut = 0 # use of shallow cuts [0, 1]
decomposition = True # square root decomposition of ellipsoid matrix indicator
memory = True # cut memorization through iterations indicator
log = False # path log indicator
[xb, fxb, x, fx, Qi, stop] = nlpalg.ellipsoidmethod(f, df, g, dg, A, b, Aeq, beq, xmin, xmax, x0, Qi0,
                                                  epsilon, kmax, kimax, shallowcut, decomposition, memory, log)
print(xb)
```

2.3 Performance

In order to have a performance reference of the ellipsoid method in nlpalg library, a MATLAB comparison has been made using the following python code. The nlpalg takes about 0.035 second to find each solution, while the MATLAB code takes about 0.6 second: nlpalg is about 17 times faster than MATLAB.

```python
# libraries
import nlpalg # nonlinear programming algorithm library
import numpy as np
import time

# quadratic objective function
xf = np.array([3, 5, 40]).reshape(-1, 1)
Af = 2*np.identity(3)
bf = -np.matmul(Af, xf)
cf = .5*np.matmul(np.transpose(xf), np.matmul(Af, xf))
f = lambda x: [.5*np.matmul(np.transpose(x), np.matmul(Af, x)) + np.matmul(np.transpose(x), bf) + cf]
df = lambda x: np.matmul(Af, x) + bf

# empty nonlinear inequality constraint
g = lambda x: np.zeros((0, 1))
dg = lambda x: np.zeros((3, 0))

# linear inequality constraint
A = np.array([[1, 1, 0], [2, 2, 0]])
b = np.array([10, 30])

# linear inequality constraint
Aeq = np.array([[1, -1, 0]])
beq = np.array([0])
```

```
# bounds
xmin = np.array([-10, -10, -10]).reshape(-1, 1) # lower
xmax = np.array([10, 10, 10]).reshape(-1, 1) # upper

# solution with ellipsoid method
x0 = np.array([0, 0, 0]).reshape(-1, 1) # starting point
Qi0 = np.array([[]]) # starting inverse ellipsoid matrix
epsilon = 0 # uncertainty on each variable
kmax = 300 # maximum number of iterations
kimax = 32 # maximum number of cuts per iteration
shallowcut = 0 # use of shallow cuts [0, 1]
decomposition = True # square root decomposition of ellipsoid matrix indicator
memory = True # cut memorization through iterations indicator
log = False # path log indicator
start = time.time()
for i in range(10):
    [xb, fxb, x, fx, Qi, stop] =
              nlpalg.ellipsoidmethod(f, df, g, dg, A, b, Aeq, beq, xmin, xmax, x0, Qi0,
              epsilon, kmax, kimax, shallowcut, decomposition, memory, log)
end = time.time()

# print solution
print((end - start)/10, " to find solution")
print(xb)
```