

## Laborationsinformation

### 1 Program för olinjär optimering

#### 1.1 OpenOpt

OpenOpt är ett paket för lösning av olinjära optimeringsproblem med eller utan bivillkor. Det är skrivet i programspråket Python. Man kör det genom att öppna ett terminalfönster, ge `TAOP86setup` och sedan skriva:

```
python mittproblem.py
```

(om man har skrivit in sitt problem i filen "mittproblem.py"). Formatet för problemfilen ges i följande avsnitt. På hemsidan <http://www.openopt.org/Welcome> återfinnes mer information om OpenOpt. (Därifrån kan man även ladda hem paketet till sin egen dator.) Glöm inte att logga ut i terminalfönstret med `logout` när du är färdig.

#### 1.2 Format för inmatning av problem i OpenOpt

Python (som OpenOpt är skrivet i) är ett interpreterande språk, så man kan i princip lösa sitt problem genom att skriva `python` och sedan skriva in en rad i taget. Vi rekommenderar dock att man istället skriver det hela på en fil, med något smart namn, såsom "kajsproblem1-lab1.py" (ändelsen ska alltid vara "py"). Sedan löser man problemet genom att skriva

```
python kajsproblem1-lab1.py
```

Filen "kajsproblem1-lab1.py" konstrueras enligt nedanstående. Man kan även titta på (kopiera) filen "ilplab0.py" från kurshemsidan. Den beskriver följande problem.

$$\begin{aligned} \min f(x) &= (x_1 - 2)^2 + 2(x_2 - 1)^2 + x_1x_2 \\ \text{då } x_1 + x_2 &\leq 1, x_1^2 + x_2^2 \leq 1, (x_1 - 1)^2 + x_2^2 \leq 0.8, 0 \leq x_1 \leq 7 \end{aligned}$$

Filen inleds alltid med följande två rader:

```
from openopt import NLP
from numpy import *
```

Sedan ges en startpunkt:

```
x0 = [0,0]
```

Därefter ges målfunktionen. Variablernas index börjar med 0, så  $x_1$  blir `x[0]` osv. Upphöjt till skrivs med `**`. (Ordet `lambda` har en speciell mening och ska alltid finnas med.)

```
f = lambda x: (x[0]-2)**2 + 2*(x[1]-1)**2 + x[0]*x[1]
```

Man kan ge övre och undre gränser för variablerna.

```
lb = [0, -inf] # lower bound
ub = [7, inf] # upper bound
```

Linjära bivillkor i formen  $Ax \leq b$  ges på följande sätt.

```
A = [1, 1]
```

```
b = 1
```

Olinjära bivillkor i formen  $c(x) \leq 0$  ges som följer. Bivillkor separeras med kommatecken.

```
c = [lambda x: x[0]**2 + x[1]**2 - 1, lambda x: (x[0]-1)**2 + x[1]**2 - 0.8]
```

Problemet definieras genom följande rad. NLP är problemtypen ("Non Linear Programming"). Endast funktionen (**f**) och startpunkten (**x0**) är nödvändiga. Allt annat kan tas bort. Om man t.ex. inte har några linjära bivillkor tas **A=A** och **b=b** bort, om man inte har några övre gränser tas **ub=ub** bort, etc.

```
p = NLP(f, x0, lb=lb, ub=ub, A=A, b=b, c=c)
```

Man kan plotta funktionsvärden och total bivillkorsavvikelse för varje iteration.

```
p.plot = True
```

Man kan välja lösningsalgoritm, beroende på vilka algoritmer som finns installerade. Följande möjligheter finns med som standard: **ralg**, **scipy\_coby** och **scipy\_slsqp**.

```
solver = 'ralg'
```

Problemet löses nu med:

```
r = p.solve(solver)
```

Lösningen skrives enklast ut med

```
print r.xf
```

Mer detaljer återfinns på hemsidan, <http://www.openopt.org/Welcome>, samt genom att i python ge `help(NLP)`.

Den kompletta filen för detta exempel ges nedan. (Allt efter `#` på en rad är kommentar.)

```
from openopt import NLP
```

```
from numpy import *
```

```
x0 = [0,0]    # starting point
```

```
f = lambda x: (x[0]-2)**2 + 2*(x[1]-1)**2 + x[0]*x[1]    # objective function
```

```
lb = [0, -inf]    # lower bound
```

```
ub = [7, inf]    # upper bound
```

```
# linear constraints Ax <= b
```

```
A = [1, 1]
```

```
b = 1
```

```
# nonlinear constraints c(x) <= 0
```

```
c = [lambda x: x[0]**2 + x[1]**2 - 1, lambda x: (x[0]-1)**2 + x[1]**2 - 0.8]
```

```
p = NLP(f, x0, lb=lb, ub=ub, A=A, b=b, c=c)    # define the problem
```

```
p.plot = True    # enable plot
```

```
solver = 'ralg'    # choose solver
```

```
r = p.solve(solver)    # solve the problem
```

```
print r.xf    # print solution
```

När man kör den med  
python ilplab0.py  
fås följande utskrifter på skärmen:

```
openopt:~> python ilplab0.py
-----
solver: ralg   problem: unnamed   type: NLP   goal: minimum
  iter   objFunVal   log10(maxResidual)
    0   6.000e+00         -0.70
   10   2.986e+00        -100.00
   20   2.876e+00        -100.00
   30   2.875e+00        -100.00
   40   2.875e+00         -6.21
istop: 3 (|| X[k] - X[k-1] || < xtol)
Solver:   Time Elapsed = 0.18   CPU Time Elapsed = 0.18
Plotting: Time Elapsed = 0.77   CPU Time Elapsed = 0.73
objFunValue: 2.8750945 (feasible, MaxResidual = 6.13365e-07)
[ 0.74307577  0.25692484]
```

Man får en utskrift var tionde iteration, dels målfunktionsvärdet och dels hur otillåten lösningen är.

Det slutgiltiga målfunktionsvärdet för den erhållna punkten är 2.8750945 (näst sista raden), och lösningen är  $x_1 = 0.74307577$ ,  $x_2 = 0.25692484$  (sista raden). Man ska aldrig förvänta sig att punkten är exakt. Det finns olika toleranser man kan sätta, men det går vi inte in på här, utan nöjer oss med standardinställningarna.

### 1.3 ILP-matlab

ILP-matlab är ett demonstrationsprogram skrivet i MATLAB för studium av lösningsmetoder för olinjära problem utan bivillkor.

För att starta programmet görs följande: Börja med att öppna ett terminalfönster och ge kommandot `TAOP86setup`. Skriv sedan `matlab` i terminalfönstret, och tryck enter. När MATLAB-fönstret öppnats, skriv `ilpmeny`. Funktionerna som skall studeras finns redan inskrivna och det enda som behövs för att lösa ett problem är att välja funktion, lösningsmetod samt startpunkt. Med "BL" avses brantaste lutningsmetoden och med "Newtons modifierade metod" avses Newtons metod med linjesökning.