Continuous Optimization with AMPL

Todd Munson

Mathematics and Computer Science Division
Argonne National Laboratory

Outline

- Continuous Optimization
- Numerical Methods
- Modeling Languages
- Beginning AMPL
- Conclusion

Continuous Optimization

- Minimize or maximize function subject to constraints

$$\begin{array}{ll} \min\limits_{x \geq 0} & f(x) \\ \text{subject to} & c(x) \leq 0 \\ & h(x) = 0 \end{array}$$

- Functions are nonlinear but sufficiently smooth
- Includes both inequality and equality constraints
- Potentially large number of variables and constraints
- Restrict ourselves to finding local solutions

Economic Applications

- Portfolio Optimization
- Principal-Agent Problems
- Maximum Likelihood Estimation
- Optimal Growth and Life-Cycle Problems
- Social Planning Problems

Numerical Methods

- Several "standard" techniques exist
  - Augmented Lagrangian methods
  - Interior-point methods
  - Sequential quadratic programming
- Many variations on these themes
  - Choice of merit function or filter
  - Line search or trust region
  - Dealing with infeasibility
  - Linear algebra employed
  - Derivative requirements
- Best algorithm and settings are problem dependent
  - Number of variables and constraints
  - Amount of nonlinearity in problem

Local Sequential Quadratic Programming Method

1. Compute direction by solving

$$\begin{aligned} \min_{d} \quad & \tfrac{1}{2} d^T H_k d + \nabla f(x^k)^T d \\ \text{subject to} \quad & \nabla c(x^k) d + c(x^k) \leq 0 \\ & \nabla h(x^k) d + h(x^k) = 0 \\ & d + x^k \geq 0 \end{aligned}$$

2. Update iterate

$$x^{k+1} = x^k + d$$

3. Repeat until convergence

Issues to Address

- Efficiently solving the subproblems
  - Warm starts
  - Rank deficiency and degeneracy
- Dealing with infeasible subproblems
  - Elastic mode
  - Feasibility restoration
- Handling unbounded subproblems
- Computing the Hessian of Lagrangian H_k
  - Possibly an indefinite matrix
  - Positive definite approximation
- Obtaining global convergence
  - Merit function or filter
  - Line search or trust region

General-Purpose Algorithms on NEOS

- Augmented Lagrangian Methods
  - LANCELOT – trust region
  - MINOS – linearly constrained Lagrangian
  - PENNON – generalized Lagrangian
- Interior-Point Methods
  - IPOPT – filter line search
  - KNITRO – trust region
  - LOQO – line search
- Sequential Quadratic Programming Methods
  - FILTER – filter
  - SNOPT – line search
- Other Techniques
  - CONOPT – generalized reduced gradient

Traditional Method with Libraries

- Download and compile code or obtain libraries
- Read documentation to learn function calls
- Write application code
  - Construct sparsity pattern
  - Compute derivative information
  - Call optimization routines
  - Verify correctness
- Compile, link, and execute code
- Fix bugs – could take days to months
- Find out algorithm not appropriate for application
  - Start over with different library
  - Function calls not standardized
  - Some code can be reused

The KNITRO Interface

```
int KTR_solve(KTR_context *kc,
    double *f,      /* (scalar) Objective function value          */
    int ftype,      /* (scalar) Type of objective                 */
    int n,          /* (scalar) The number of unknowns            */
    double *x,      /* (length n, vector) Unknown values          */
    double *bl,     /* (length n, vector) Lower bounds            */
    double *bu,     /* (length n, vector) Upper bounds            */
    double *fgrad,  /* (length n, vector) Obj. gradient           */
    int m,          /* (scalar) The number of constraints         */
    double *c,      /* (length m, vector) Constraint values       */
    double *cl,     /* (length m, vector) Lower bounds            */
    double *cu,     /* (length m, vector) Upper bounds            */
    int *ctype,     /* (length m, vector) Constraint type         */
    int nnzj,       /* (scalar) Number of nonzeros in Jacobian    */
    double *cjac,   /* (length nnzj, vector) Jacobian data        */
    int *indvar,    /* (length nnzj) column start index           */
    int *indfun,    /* (length nnzj) row index                    */
    double *lambda, /* (length m+n, vector) Multiplier estimates */
    int nnzh,       /* (scalar) Number of nonzeros in Hessian     */
    double *hess,   /* (length nnzh, vector) Hessian data         */
    int *hrow,      /* (length nnzh) row index                    */
    int *hcol,      /* (length nnzh) column index                 */
    double *vector, /* (length n) vector for Hessian products     */
    void *user      /* pointer for callback functions             */
);
```

Modeling Languages: A Better Way

- Portable language for specifying problems
  - Based on algebraic description
  - Large models can be processed
  - Easy to modify code as needed
  - Include programming language features
- Able to switch algorithms at will
  - No need to compile codes
  - Interfaces written by developers
  - Derivatives automatically calculated
  - Algorithm specific options can be set
- Support for other facilities
  - Relational databases and spreadsheets
  - MATLAB interface to obtain function evaluations

Notes on Modeling Languages

- Pros
  - Excellent documentation
  - Easy to use with convenient syntax
  - Many solvers are available
  - Automatic differentiation
  - Separation of model and data
  - Large user communities
- Cons
  - Interpreted languages
    - May be slow for some applications
    - Can consume large amount of memory
  - Not easily extensible to new paradigms (SDP)

AMPL Overview: Model Declaration

- **Set declarations**
  - **Unordered, ordered, and circular sets**
  - **Cross products and point to set mappings**
  - **Set manipulation operations**
- **Parameters and variables**
  - **Attributes**
  - **Check statements**
  - **Defined variables**
- **Objectives and constraints**
  - **Equality, inequality, and range constraints**
  - **Complementarity constraints**
  - **Multiple objectives**
- **Problem statement**

AMPL Overview: Data and Commands

- Data Declaration
  - Set definitions
    - Explicit list of elements
    - Implicit list in parameter statements
  - Parameter definitions
    - Tables and transposed tables
    - Higher dimensional parameters
- Execution Commands
  - Load model and data
  - Select problem, algorithm, and options
  - Solve the instance
  - Output results

AMPL Overview: Other Operations

- Let and fix statements
- Conditionals and loop constructs
- Execution of external programs
- Access to relational databases

Social Planning for Endowment Economy

- Economy with n agents and m commodities
  - $e \in \Re^{n \times m}$ are the endowments
  - $\alpha, \beta \in \Re^{n \times m}$ are the utility parameters
  - $\lambda \in \Re^{n}$ are the social weights
- Social planning problem

$$\begin{array}{lll}
\max\limits_{x \geq 0, u} & \sum\limits_{i=1}^{n} \lambda_i u_i & \\
\text{subject to} & \sum\limits_{i=1}^{n} x_{i,k} \leq \sum\limits_{i=1}^{n} e_{i,k} & \forall k = 1, \ldots, m \\
 & u_i = \sum\limits_{k=1}^{n} \dfrac{\alpha_{i,k}(1 + x_{i,k})^{1-\beta_{i,k}}}{1 - \beta_{i,k}} & \forall i = 1, \ldots, n
\end{array}$$

Model Definition: `social1.mod`

```
param n > 0, integer;                     # Agents
param m > 0, integer;                     # Commodities

param e {1..n, 1..m} >= 0, default 1; # Endowment

param lambda {1..n} > 0;                  # Social weights
param alpha {1..n, 1..m} > 0;             # Utility parameters
param beta {1..n, 1..m} > 0;

var x {1..n, 1..m} >= 0;                  # Consumption
var u {i in 1..n} =                       # Utility
  sum {k in 1..m} alpha[i,k] * (1 + x[i,k])^(1 - beta[i,k]) / (1 - beta[i,k]);

maximize welfare:
    sum {i in 1..n} lambda[i] * u[i];

subject to
  consumption {k in 1..m}:
    sum {i in 1..n} x[i,k] <= sum {i in 1..n} e[i,k];
```

Data Definition: `social1.dat`

```
param n := 3;           # Agents
param m := 4;           # Commodities

param alpha : 1    2    3    4 :=
       1      1    1    1    1
       2      1    2    3    4
       3      2    1    1    5;

param beta (tr) : 1    2    3 :=
       1         1.5  2   0.6
       2         1.6  3   0.7
       3         1.7  2   2.0
       4         1.8  2   2.5;

param : lambda :=
    1     1
    2     1
    3     1;
```

Execution Commands: `social1.cmd`

```
# Load model and data
model social1.mod;
data social1.dat;

# Specify solver and options
option solver "minos";
option minos_options "outlev=1";

# Solve the instance
solve;

# Output results
display x;
printf {i in 1..n} "%2d: % 5.4e\n", i, u[i];
```

Output from AMPL

```
ampl: include social1.cmd;
MINOS 5.5: outlev=1
MINOS 5.5: optimal solution found.
25 iterations, objective 2.252422003
Nonlin evals: obj = 44, grad = 43.
x :=
1 1   0.0811471
1 2   0.574164
1 3   0.703454
1 4   0.267241
2 1   0.060263
2 2   0.604858
2 3   1.7239
2 4   1.47516
3 1   2.85859
3 2   1.82098
3 3   0.572645
3 4   1.2576
;

 1: -5.2111e+00
 2: -4.0488e+00
 3:  1.1512e+01
ampl: quit;
```

Model Definition: `social2.mod`

```
set AGENTS;                                           # Agents
set COMMODITIES;                                      # Commodities

param e {AGENTS, COMMODITIES} >= 0, default 1; # Endowment

param lambda {AGENTS} > 0;                            # Social weights
param alpha {AGENTS, COMMODITIES} > 0;                # Utility parameters
param beta {AGENTS, COMMODITIES} > 0;

param gamma {i in AGENTS, k in COMMODITIES} := 1 - beta[i,k];

var x {AGENTS, COMMODITIES} >= 0;                     # Consumption
var u {i in AGENTS} =                                 # Utility
  sum {k in COMMODITIES} alpha[i,k] * (1 + x[i,k])^gamma[i,k] / gamma[i,k];

maximize welfare:
    sum {i in AGENTS} lambda[i] * u[i];

subject to
  consumption {k in COMMODITIES}:
    sum {i in AGENTS} x[i,k] <= sum {i in AGENTS} e[i,k];
```

Data Definition: `social2.dat`

```
set COMMODITIES := Books, Cars, Food, Pens;

param: AGENTS : lambda :=
       Jorge      1
       Sven       1
       Todd       1;

param alpha : Books  Cars  Food  Pens :=
     Jorge      1     1     1     1
     Sven       1     2     3     4
     Todd       2     1     1     5;

param beta (tr): Jorge  Sven  Todd :=
     Books        1.5    2     0.6
     Cars         1.6    3     0.7
     Food         1.7    2     2.0
     Pens         1.8    2     2.5;
```

Execution Commands: `social.cmd`

```
# Load model and data
model social2.mod;
data social2.dat;

# Specify solver and options
option solver "minos";
option minos_options "outlev=1";

# Solve the instance
solve;

# Output results
display x;
printf {i in AGENTS} "%5s: % 5.4e\n", i, u[i];
```

Output from AMPL

```
ampl: include social.cmd;
MINOS 5.5: outlev=1
MINOS 5.5: optimal solution found.
25 iterations, objective 2.252422003
Nonlin evals: obj = 44, grad = 43.
x :=
Jorge Books   0.0811471
Jorge Cars    0.574164
Jorge Food    0.703454
Jorge Pens    0.267241
Sven  Books   0.060263
Sven  Cars    0.604858
Sven  Food    1.7239
Sven  Pens    1.47516
Todd  Books   2.85859
Todd  Cars    1.82098
Todd  Food    0.572645
Todd  Pens    1.2576
;

Jorge: -5.2111e+00
 Sven: -4.0488e+00
 Todd:  1.1512e+01
ampl: quit;
```

Traffic Routing with Congestion

- Route commodities through network
  - $\mathcal{N}$ denotes the nodes and $\mathcal{A} \subseteq \mathcal{N} \times \mathcal{N}$ the arcs
  - $\mathcal{K}$ denotes the commodities
  - α, β are the congestion parameters
  - b denotes the supply and demand
- Multicommodity network flow problem

$$\begin{array}{lll}
\displaystyle\min_{x \geq 0, f \geq 0} & \displaystyle\sum_{(i,j)\in\mathcal{A}} \alpha_{i,j} f_{i,j} + \beta_{i,j} f_{i,j}^4 & \\
\text{subject to} & \displaystyle\sum_{(i,j)\in\mathcal{A}} x_{i,j,k} \leq \sum_{(j,i)\in\mathcal{A}} x_{j,i,k} + b_{i,k} & \forall i \in \mathcal{N}, k \in \mathcal{K} \\
& \displaystyle f_{i,j} = \sum_{k\in\mathcal{K}} x_{i,j,k} & \forall (i,j) \in \mathcal{A}
\end{array}$$

Model Definition: network.mod

```
set NODES;                                          # Nodes in network
set ARCS within NODES cross NODES;                  # Arcs in network
set COMMODITIES := 1..3;                            # Commodities

param b {NODES, COMMODITIES} default 0;             # Supply/demand
check {k in COMMODITIES}:                           # Supply exceeds demand
  sum{i in NODES} b[i,k] >= 0;

param alpha {ARCS} >= 0;                            # Linear part
param beta {ARCS} >= 0;                             # Nonlinear part

var x {ARCS, COMMODITIES} >= 0;                     # Flow on arcs
var f {(i,j) in ARCS} =                             # Total flow
  sum {k in COMMODITIES} x[i,j,k];

minimize time:
  sum {(i,j) in ARCS} (alpha[i,j]*f[i,j] + beta[i,j]*f[i,j]^4);

subject to
  conserve {i in NODES, k in COMMODITIES}:
    sum {(i,j) in ARCS} x[i,j,k] <= sum{(j,i) in ARCS} x[j,i,k] + b[i,k];
```

Data Definition: `network.dat`

```
set NODES := 1 2 3 4 5;

param: ARCS : alpha beta =
        1 2      1    0.5
        1 3      1    0.4
        2 3      2    0.7
        2 4      3    0.1
        3 2      1    0.0
        3 4      4    0.5
        4 1      5    0.0
        4 5      2    0.1
        5 2      0    1.0;

let b[1,1] :=  7;     # Node 1, Commodity 1 supply
let b[4,1] := -7;     # Node 4, Commodity 1 demand
let b[2,2] :=  3;     # Node 2, Commodity 2 supply
let b[5,2] := -3;     # Node 5, Commodity 2 demand
let b[3,3] :=  5;     # Node 1, Commodity 3 supply
let b[1,3] := -5;     # Node 4, Commodity 3 demand

fix {i in NODES, k in COMMODITIES: (i,i) in ARCS} x[i,i,k] := 0;
```

Execution Commands: `network.cmd`

```
# Load model and data
model network.mod;
data network.dat;

# Specify solver and options
option solver "minos";
option minos_options "outlev=1";

# Solve the instance
solve;

# Output results
for {k in COMMODITIES} {
  printf "Commodity: %d\n", k > network.out;
  printf {(i,j) in ARCS: x[i,j,k] > 0} "%d.%d = % 5.4e\n", i, j, x[i,j,k] > network.out;
  printf "\n" > network.out;
}
```

Output File: `network.out`

```
Commodity: 1
1.2 =  3.3775e+00
1.3 =  3.6225e+00
2.4 =  6.4649e+00
3.2 =  3.0874e+00
3.4 =  5.3510e-01

Commodity: 2
2.4 =  3.0000e+00
4.5 =  3.0000e+00

Commodity: 3
3.4 =  5.0000e+00
4.1 =  5.0000e+00
```

Accessing Other Solvers: Kestrel Client

```
# Load model and data
model network.mod;
data network.dat;

# Specify solver and options
option solver "kestrel";
option kestrel_options "solver=knitro";
option knitro_options "outlev=1";

# Solve the instance
solve;

# Output results
for {k in COMMODITIES} {
  printf "Commodity: %d\n", k > network.out;
  printf {(i,j) in ARCS: x[i,j,k] > 0} "%d.%d = % 5.4e\n", i, j, x[i,j,k] > network.out;
  printf "\n" > network.out;
}
```

Edited Output from AMPL

```
Job has been submitted to Kestrel
Kestrel/NEOS Job number    : 579001
Kestrel/NEOS Job password  : XDdwVLQU
Check the following URL for progress report :
     http://www-neos.mcs.anl.gov/neos/neos-cgi/check-status.cgi?job=579001&pass=XDdwVLQU
In case of problems, e-mail :
     neos-comments@mcs.anl.gov

Intermediate Solver Output:
KNITRO 4.0: 12/15/04
======================================
             KNITRO 4.0.4
       Ziena Optimization, Inc.
       website:  www.ziena.com
       email:   info@ziena.com
======================================

  Iter      Objective      Feas err    Opt err     ||Step||    CG its
--------  -------------  ----------  ----------  ----------  --------
       0   3.131973e+02   7.000e+00
      10   1.467524e+03   4.710e-02   6.322e+00   4.704e-01         0
      15   1.505526e+03   2.019e-07   2.844e-05   8.428e-04         0

EXIT: LOCALLY OPTIMAL SOLUTION FOUND.
=========================================================================
```

Conclusion

- Optimization problems are prevalent
- Algorithms for solving them are mature
  - Commercially available
  - Usable through NEOS
- Modeling languages good for expressing problems
  - Based on algebraic description
  - Large models can be processed
  - Many features