7.Nonlinear Programming and MIP Technology

Roberto Amadini

Department of Computer Science and Engineering, University of Bologna, Italy

Combinatorial Decision Making and Optimization

2nd cycle degree programme in Artificial Intelligence

University of Bologna, Academic Year 2024/25

From Linear to Nonlinear

- Some problems can be easily encoded in standard LP form
- For some others is not trivial to get a linear formulation, they may involve constraints like:
  - $y = x^2 + \frac{1}{z}$
  - $c > 10 \implies a \geq b \vee d \neq e$
  - $allDifferent(x_1, \ldots, x_n)$
- Mathematical methods for handling these problems either:
  - Use a specific Nonlinear Programming approach, or
  - Encode the problem into an equisatisfiable linear problem

Nonlinear programming

- Nonlinear Programming (NLP) problems have generic form:

$$\begin{array}{ll} \min & f(x) \\ s.t. & g_i(x) \leq 0 \quad i = 1, \ldots, m \end{array}$$

  where one function among $f, g_1, \ldots, g_m$ is non-linear

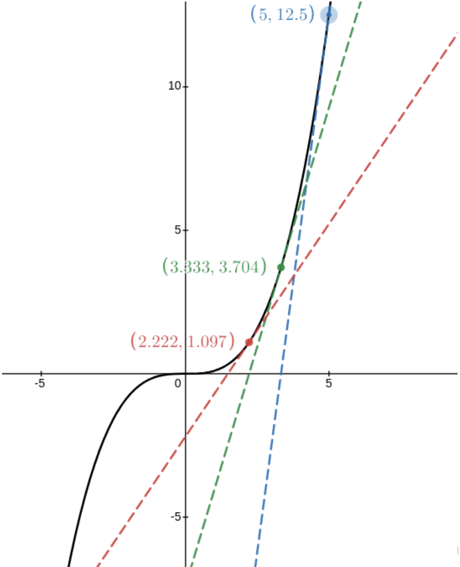
- Different NLP methods exist based on the problem type
  - E.g., Quadratic programming (QP): min $\frac{1}{2}x^t\, Q\, x + c^t x$ s.t $Ax \leq b$
  - Newton's method
  - Steepest descent
  - Lagrange multipliers
  - ...

Newton-Raphson's method (root finding)

- Let $f : X \to \mathbb{R}$ be a function, f' its derivative and $x_0 \in X$
- The tangent of f at $(x_0, f(x_0))$ is $t_0(x) = f'(x_0)(x - x_0) + f(x_0)$
  - We can use t_0 as linear approximation of f
- The intersection of t_0 with x-axis is a point x_1 such that $t_0(x_1) = 0$, hence $f'(x_0)(x_1 - x_0) + f(x_0) = 0$ so $x_1 = x_0 - \dfrac{f(x_0)}{f'(x_0)}$
- We can iterate to get $x_2 = x_1 - \dfrac{f(x_1)}{f'(x_1)}, \ldots, x_{k+1} = x_k - \dfrac{f(x_k)}{f'(x_k)}$
- Under some conditions (...) we converge to a x_k such that $f(x_k) = 0$

Example

E.g. if $f(x) = \frac{x^3}{10}$ and $x_0 = 5$:

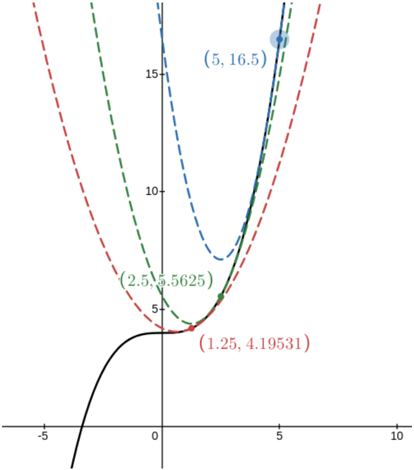


Newton-Raphson's method (optimization)

- If f is twice-differentiable, this method is applicable to find the roots of f' too: $f'(x) = 0 \implies$ stationary points
  - Either a minimum, maximum or inflection (saddle) point
- At k-th iteration, instead of tangent t_k we consider the parabola $p_k(x) = f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2}f''(x_k)(x - x_k)^2$ with same slope and curvature of $f(x_k)$ and then proceed with $x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$
- If $f(x) = ax^2 + bx + c$ then only one step needed to converge: $x_1 = x_0 - \frac{2ax_0 + b}{2a} = -\frac{b}{2a}$ and $f'(x_1) = 2a \cdot \frac{b^2}{4a^2} - b \cdot \frac{b}{2a} = 0$

Example

E.g. if $f(x) = \dfrac{x^3}{10} + 4$ and $x_0 = 5$:

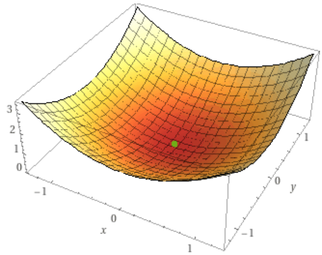


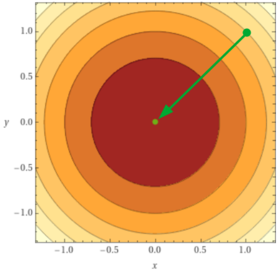
Newton-Raphson's method (multivariable minimization)

- If $f : \mathbb{R}^n \rightarrow \mathbb{R}$ with $n > 1$, we use the gradient instead of f':
  $\nabla f(x) = \left\{ \left(\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_n}(x) \right) \mid x \in \mathbb{R}^n \right\}$
  - ∇f is a vector in $\mathbb{R}^n$ denoting the direction of steepest ascent
- Instead of f'' we use Hessian $\nabla^2 f(x) = H_f = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ & \cdots & \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$
  - $\nabla^2 f$ is a matrix in $\mathbb{R}^{n \times n}$ denoting the curvature of f
- We start with $\mathbf{x_0} \in \mathbb{R}^n$ and $\mathbf{x_{k+1}} = x_k - (\nabla^2 f(\mathbf{x_k}))^{-1} \cdot \nabla f(\mathbf{x_k})$
  - $\nabla f(x_k)$ points "uphill" $\rightarrow -\nabla f(x_k)$ points "downhill"
  - H_f "adjusts" the direction according to "how sharp" the surface is

Example

- E.g., if $f(x, y) = x^2 + y^2$ then $\nabla f = (2x, 2y)$ and $\nabla^2 f = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$
- If $x_0 = (1, 1)$ we have $\nabla f(x_0) = (2, 2)$ and $\nabla^2 f(x_0) = \nabla^2 f$
- $x_1 = x_0 - (\nabla^2 f(\mathbf{x_0}))^{-1} \cdot \nabla f(\mathbf{x_0}) = (1, 1) - \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix} \begin{pmatrix} 2 \\ 2 \end{pmatrix} = (0, 0)$
- $\nabla f(x_1) = (0, 0)$: we found a local minimum
  - $\nabla f(x) = (0, \ldots, 0)$ is necessary but not sufficient condition for x be a global optimum





Newton-Raphson's method

- Interior point approaches can exploit Newton-Raphson's method to traverse the feasible region of constrained (N)LP problems
  - E.g., primal-dual interior-point method
- This is a second-order method: 2^{nd}-order derivatives used to compute the optimization trajectory (Hessian)
  - While first-order methods are based on 1^{st} order derivatives (gradient)
  - Derivatives-free methods also exist (e.g. dichotomic search)
- Computing and inverting Hessian matrix is expensive
  - Also no guarantee of convergence if not positive definite

Local Search

- Newton-Raphson's method is a form of Local Search (LS):
  - It starts from an initial state
  - It moves from one state to another one
  - Each move uses local information only (neighborhood states)
  - Each move should improve the current state
  - Global optimality not guaranteed in general
- Other well known LS methods are: simulated annealing, hill climbing, steepest descent, tabu search, LNS...
- Gradient descent: steepest descent method using the negative gradient to decide direction of next move

Gradient descent

- Gradient descent (GD) is a well-known 1st order approach that gained popularity with the rise of ML and (deep) neural networks
- GD often used for training ML models: a loss function $L(x, \theta)$ over input data $x = (x_1, \dots, x_n)$ and model parameters $\theta = (\theta_1, \dots, \theta_m)$ is iteratively minimized starting from an initial state $\theta^{(0)}$
- GD computes $\theta^{(k+1)} = \theta^{(k)} - \lambda \nabla L(\theta^{(k)})$ for $k = 0, 1, 2, \dots$
- Step $k + 1$ depends on direction $-\nabla L(\theta^k)$ and learning rate λ
  - $-\nabla L(\theta^k)$ specifies "where to go" to minimize the loss
  - λ is the "size of a step" towards minimum loss (typically small)

Gradient descent

- GD can be used in different modes:
  - Batch: model updated after evaluating all training set (training epoch)
  - Stochastic: subset of N samples randomly selected from training set
    - If $N > 1$ is small, sometimes referred as mini-batch
- Apart from neural networks, GD can be used to train linear classifiers
  - Less robust than SVMs
  - Nonlinear logistic classifiers better for binary classification
- Newton's can converge in fewer steps than GD but computing and inverting the Hessian might make it slower in practice

Lagrange multipliers

- Instead of solving $\min f(x)$ s.t. $g_i(x) \leq 0$, unconstrain the problem and only keep a new loss function: $\min \left(L(x, \Lambda) = f(x) + \Lambda_i p_i(x) \right)$
  - Weights $\Lambda_i \geq 0$ are called Lagrangian multipliers
  - Terms $p_i(x) \geq 0$ are penalty functions such that $g_i(x) \leq 0 \Leftrightarrow p(x) = 0$
- Penalties $p_i(x)$ should reflect "how far" x is from satisfying $g_i(x) \leq 0$
  - E.g., $p_i(x) = \max(0, g_i(x))$ better than $p_i(x) = \begin{cases} 0 & \text{if } g_i(x) \leq 0 \\ 1 & \text{if } g_i(x) > 0 \end{cases}$
- Lagrangian-based LS can be applied for any constraint $c_i(x)$ by using steepest descent to iteratively update weights Λ_i at each step
  - To use gradient descent, penalties $p_i(x)$ should be differentiable

Linearization

- An alternative approach, suitable for combinatorial problems, is to linearize nonlinear constraints to get an equisatisfiable MIP problem
- This approach works if the variables are bounded
- E.g., nonlinear constraint $5x \leq 18 \lor -y + 2z < 3$
  - i.e., $f(x, y, z) \geq 1$ with $f(x, y, z) = \begin{cases} 1 & \text{if } 5x \leq 18 \text{ or } -y + 2z < 3 \\ 0 & \text{otherwise} \end{cases}$

Reification

- Logical combinations of constraints can be handled via reification
- Given constraint $C(x_1, \ldots, x_k)$ a (full) reification of C is a Boolean variable b s.t. $b = true \iff C(x_1, \ldots, x_k)$
  - An integer reification is s.t. $b \in \{0, 1\}$ and $b = 1 \iff C(x_1, \ldots, x_k)$
  - An integer half-reification is s.t. $b \in \{0, 1\}$ and $b = 1 \Rightarrow C(x_1, \ldots, x_k)$
    - $b = 0 \ \vee \ C(x_1, \ldots, x_k)$
- Easy case: integer reification of Boolean operations. If C, C_1, C_2 are Boolean variables (i.e., propositions or 0-arity predicates):
  - $C = C_1 \vee C_2$ becomes $b_1 \leq b \wedge b_2 \leq b \wedge b \leq b_1 + b_2$
  - $C = C_1 \wedge C_2$ becomes $b \leq b_1 \wedge b \leq b_2 \wedge b_1 + b_2 \leq b + 1$
  - $C = \neg C_1$ becomes $b = 1 - b_1$

  with $b, b_1, b_2 \in \{0, 1\}$

Reification

- Logical combinations of constraints can be handled via reification
- Given constraint $C(x_1, \ldots, x_k)$ a (full) reification of C is a Boolean variable b s.t. $b = true \iff C(x_1, \ldots, x_k)$
  - An integer reification is s.t. $b \in \{0, 1\}$ and $b = 1 \iff C(x_1, \ldots, x_k)$
  - An integer half-reification is s.t. $b \in \{0, 1\}$ and $b = 1 \Rightarrow C(x_1, \ldots, x_k)$
    - $b = 0 \ \vee \ C(x_1, \ldots, x_k)$
- Common case: $C_1 \vee \cdots \vee C_m$ where C_i are linear inequalities: $C_i \equiv \sum_j \alpha_{i,j} x_j \leq \beta_i$ for $i = 1, \ldots, m$ and $j = 1, \ldots, n$
  - At least one inequality must hold
  - How to linearize it via integer reification?

Big-M trick

- We introduce integer reifications $b_1, \ldots, b_m \in \{0,1\}$, impose $\sum_{i=1}^{m} b_i \geq 1$ and for each inequality $\sum_j \alpha_{i,j} x_j \leq \beta_i$ we enforce $\sum_j \alpha_{i,j} x_j - \beta_i \leq M \cdot (1 - b_i)$ where M is a *"big enough"* constant:
  - $b_i = 0 \implies \sum_j \alpha_{i,j} x_j - \beta_i \leq M$: always satisfied
  - $b_i = 1 \implies \sum_j \alpha_{i,j} x_j \leq \beta_i$: original inequality C_i must be satisfied
- M is called a Big-M number. If $x_j \in [l_j..u_j]$ we can define a specialized Big-M for each equation: $M_i = -\beta_i + \sum_j \max\{\alpha_{i,j} l_j, \alpha_{i,j} u_j\}$
- E.g., if $5x \leq 18 \vee -y + 2z \leq 3$ with $x \in [0..30]$, $y \in [-5..-2]$, $z \in [-6..7]$ we introduce $b_1, b_2 \in \{0,1\}$ and we impose:
  - $b_1 + b_2 \geq 1$
  - $5x \leq 18 + (\max\{5 \cdot 0, 5 \cdot 30\} - 18) \cdot (1 - b_1) = 18 + 132 \cdot (1 - b_1)$
  - $-y + 2z \leq ((-1) \cdot (-5) + 2 \cdot 7 - 3) \cdot (1 - b_2) = 3 + 16 \cdot (1 - b_2)$

Min/max constraints

- To linearize $y = \min\{x_1, x_2\}$ with $x_1 \in [l_1..u_1], x_2 \in [l_2..u_2]$ we could linearize $(x_1 \leq x_2 \Rightarrow y = x_1) \wedge (x_2 < x_1 \Rightarrow y = x_2)$ as above
  - 4 binary variables introduced
- ...Or simply add 1 new variable $b \in \{0, 1\}$ and impose:
  - $y \leq x_1$
  - $y \leq x_2$
  - $y \geq x_1 - (u_1 - l_2) \cdot (1 - b)$
  - $y \geq x_2 - (u_2 - l_1) \cdot b$
- In this way:
  - $b = 0 \implies y \geq \overbrace{x_1 - u_1}^{\leq 0} + l_2 \ \wedge \ y \geq x_2 \implies y = x_2$
  - $b = 1 \implies y \geq \underbrace{x_2 - u_2}_{\leq 0} + l_1 \ \wedge \ y \geq x_1 \implies y = x_1$

Min/max constraints

- If $y = \min\{x_1, \dots, x_k\}$, we introduce $b_1, \dots, b_k \in \{0, 1\}$ and impose $\sum_{i=1}^{k} b_i = 1 \quad \wedge \quad y \leq x_i \quad \wedge \quad y \geq x_i - (u_i - l_{min})(1 - b_i)$
  - $l_{min} = \min\{l_1, \dots, l_k\}$, so $b_i = 1 \implies x_i = \min\{x_1, \dots, x_k\}$
- If $y = \max\{x_1, \dots, x_k\}$, $y \geq x_i \;\wedge\; y \leq x_i + (u_{max} - l_i)(1 - b_i)$
  - $u_{max} = \max\{u_1, \dots, u_k\}$ and $b_i = 1 \implies x_i = \max\{x_1, \dots, x_k\}$
- This approach can be used for other nonlinear constraints
  - E.g. $y = |x|$, $x \neq y$ or $y = kx$ with $k \in \{0, 1\}$. Try as exercise!

Unary encoding

- Alternative approach: unary encoding. If $D(x)$ is the domain of x we introduce $|D(x)|$ binary variables $b_k^x \in \{0,1\}$ and impose:

$$\sum_{k \in D(x)} b_k^x = 1 \;\wedge\; \sum_{k \in D(x)} k \cdot b_k^x = x \qquad (1)$$

- In this way $b_k^x = 1 \iff x = k$
  - E.g., $x \in [4..6]$ encoded as $b_4^x + b_5^x + b_6^x = 1 \;\wedge\; 4b_4^x + 5b_5^x + 6b_6^x = x$
- Pros: tighter linear relaxation, better encoding of global constraints
- Cons: lots of binary variables if the domain is big

Unary encoding

- E.g., for *allDifferent*$(x_1, \ldots, x_n)$ we impose (1) and $\sum_{i=1}^{n} \alpha_{i,j} b_j^{x_i} \leq 1$ for $j \in \bigcup_{1 \leq h < k \leq n} (D(x_h) \cap D(x_k))$ and $\alpha_{i,j} = \begin{cases} 1 & \text{if } j \in D(x_i) \\ 0 & \text{otherwise} \end{cases}$

- E.g., if $x \in [2..11]$, $y \in [-5..4]$, $z \in [3..5]$ we impose (1) for x, y, z and we constrain variables b_j for $j \in [2..4] \cup [3..5] \cup [3..5] = [2..5]$:

$$\begin{array}{ccccccc} b_2^x & + & b_2^y & + & 0\cancel{b_2^z} & \leq & 1 \\ b_3^x & + & b_3^y & + & b_3^z & \leq & 1 \\ b_4^x & + & b_4^y & + & b_4^z & \leq & 1 \\ b_5^x & + & 0\cancel{b_5^y} & + & b_5^z & \leq & 1 \end{array}$$

Unary encoding

- Easy encoding of element: $z = [x_1, \ldots, x_n][y]$ as $z = \sum_{i=1}^{n} b_i^y x_i$
  - Not linear, but easy to linearize
- Multiplication harder! E.g., $z = xy$ with $y \in [l_y..u_y]$ can be defined as $z = [xl_y, \ldots, xu_y][y - l_y + 1]$
  - E.g., $z = xy$ with $y \in [2..3]$ becomes $z = [2x, 3x][y - 1]$
- Particular case: $y = x^n$ with n constant
- E.g., $y = x^3$ with $x \in [-2..1]$ becomes $y = [-8, -1, 0, 1][x + 3]$

MiniZinc encoding

- MiniZinc allows one to compile a CP model with finite domains into an equisatisfiable FlatZinc instance with linear constraints only
  - So any MIP solver supporting FlatZinc can solve it!
  - Useful, not necessarily efficient...
- See `share/minizinc/linear/domain_encodings.mzn` where function `eq_encode(var int:x)` enforces (1) for integer variable x
  - MiniZinc common subexpression elimination ensures that binary variables b_i^x are reused if x is encoded again
- More details in: Belov, Gleb, et al. *"Improved linearization of constraint programming models."* CP 2016.

MiniZinc solvers

MiniZinc bundle contains different MIP solvers. Some of them are directly usable, without further installations:

- COIN-BC (a.k.a. CBC)
  - Based on Branch-and-Cut `https://github.com/coin-or/Cbc`
  - Part of open-source COIN-OR initiative
  - Default solver of PuLP Python module for LP
- HiGHS
  - Based on high performance dual revised simplex implementation (HSOL) and its parallel variant (PAMI) by Q. Huangfu*
  - Freely available `https://highs.dev`

*Parallelizing the dual revised simplex method, Q. Huangfu and J. A. J. Hall, Mathematical Programming Computation, 10 (1), 119-142, 2018.

MiniZinc solvers

Other MIP solvers need external plugins/licenses:

- IBM ILOG CPLEX (commercial, free licenses available)
  - https://www.ibm.com/products/ilog-cplex-optimization-studio
- Gurobi (commercial, free licenses available)
  - https://www.gurobi.com/
- SCIP (non-commercial)
  - https://www.scipopt.org/)
- FICO Xpress (commercial, free licenses available)
  - https://www.fico.com/en/products/fico-xpress-solver

MIP Technology

- MiniZinc is a "CP-oriented" modeling language. More "mathematical programming"-oriented languages exist
  - E.g., AMPL
- Alternatively, one can use LP libraries or add-in
  - E.g., PuLP or Excel solver
- Or simply the APIs provided by a particular solver
  - E.g., Gurobipy

AMPL

- AMPL (*A Mathematical Programming Language*) is an algebraic modeling language developed by Fourer, Gay, Kernighan in 1985
- It supports separation between model and data (*high-level*)
- Problems are passed to solvers as .nl files (*low-level*)
- Supported by many open-source and commercial solvers
- `https://ampl.com/`

Brewery example with AMPL

beer.dat

```
set PROD := beer ale;
set INGR := corn hops malt;

param: profit :=
ale  13
beer 23;

param: supply :=
corn  480
hops  160
malt 1190;

param amt: ale beer :=
corn          5  15
hops          4   4
malt         35  20;
```

beer.mod

```
set INGR;
set PROD;
param profit {PROD};
param supply {INGR};
param amt {INGR, PROD};
var x {PROD} >= 0;

maximize total_profit:
   sum {j in PROD} x[j] * profit[j];

subject to constraints {i in INGR}:
   sum {j in PROD} amt[i,j] * x[j] <= supply[i];
```

```
[cos226:tucson] ~> ampl
AMPL Version 20010215 (SunOS 5.7)
ampl: model beer.mod;
ampl: data beer.dat;
ampl: solve;
CPLEX 7.1.0: optimal solution; objective 800
ampl: display x;
x [*] :=  ale 12  beer 28;
```

From https://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/22LinearProgramming.pdf

Brewery example with GurobiPy

```python
import gurobipy as gp

# Create a new model
m = gp.Model()

# Create variables
A = m.addVar(vtype='I', name="Ale")
B = m.addVar(vtype='I', name="Beer")
# Set objective function
m.setObjective(13 * A + 23 * B, gp.GRB.MAXIMIZE)

# Add constraints
m.addConstr(5*A + 15*B <= 480)
m.addConstr(4*A +  4*B <= 160)
m.addConstr(35*A + 20*B <= 1190)

# Solve it!
m.optimize()

print(f"Optimal profit: {m.objVal}: {A.X} ale(s) and {B.X} beer(s)")
```

`brewery.py`

Brewery example with PULP

```
from pulp import *

prob = LpProblem("Brewery Problem", LpMaximize)

A = LpVariable("Ale", 0, None, LpInteger)
B = LpVariable("Beer", 0, None, LpInteger)

prob += 13*A + 23*B, "Profit"
prob += 5*A + 15*B <= 480, "Corn"
prob += 4*A + 4*B <= 160, "Hop"
prob += 35*A + 20*B <= 1190, "Malt"

# We can specify the solver to use as a parameter of solve
prob.solve()
```

brewery2.py

Take-home messages

- Nonlinear programming (NLP) involves nonlinear constraints and/or objective function
- One can tackle general NLP problems with ad hoc techniques...
  - Quadratic programming, Newton's method, Gradient descent, Lagrange multipliers, item ...
- ...or linearize nonlinear constraints to get an equisatisfiable linear problem
  - Big-M
  - Unary encoding
  - ...
- Plenty of MIP technology available (e.g., AMPL language)

Resources

- CDMO course a.y. 2020/21
- Belov, Gleb, et al. ”*Improved linearization of constraint programming models.*” CP 2016