



# Mathematical Programming: Modelling and Software

Leo Liberti

LIX, École Polytechnique, France



# Introduction



# Example: Set covering

There are 12 possible geographical positions  $A_1, \dots, A_{12}$  where some discharge water filtering plants can be built. These plants are supposed to service 5 cities  $C_1, \dots, C_5$ ; building a plant at site  $j$  ( $j \in \{1, \dots, 12\}$ ) has cost  $c_j$  and filtering capacity (in kg/year)  $f_j$ ; the total amount of discharge water produced by all cities is  $1.2 \times 10^{11}$  kg/year. A plant built on site  $j$  can serve city  $i$  if the corresponding  $(i, j)$ -th entry is marked by a '\*' in the table below.

	$A_1$	$A_2$	$A_3$	$A_4$	$A_5$	$A_6$	$A_7$	$A_8$	$A_9$	$A_{10}$	$A_{11}$	$A_{12}$
$C_1$	*		*		*		*	*				*
$C_2$		*	*			*			*		*	*
$C_3$	*	*				*	*			*		
$C_4$		*		*			*	*		*		*
$C_5$				*	*	*			*	*	*	*
$c_j$	7	9	12	3	4	4	5	11	8	6	7	16
$f_j$	15	39	26	31	34	24	51	19	18	36	41	34

What is the best placement for the plants?



# Example: Sudoku

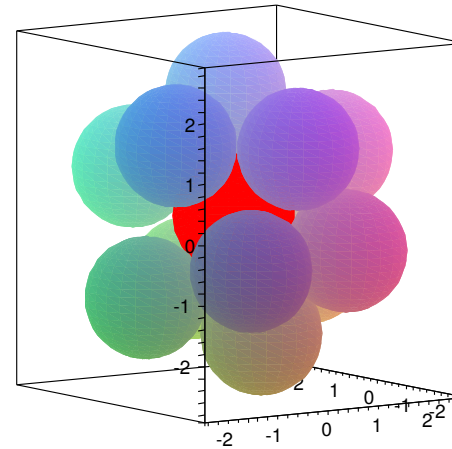
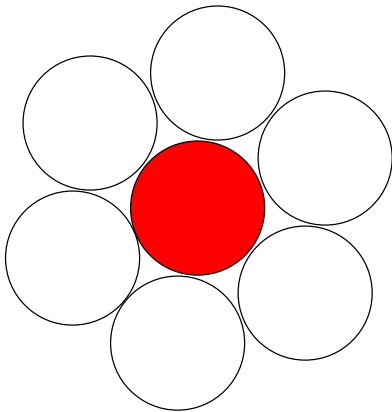
Given the Sudoku grid below, find a solution or prove that no solution exists

2								1
	4	1	9		2	8	6	
5	8						2	7
			5	1	3			
				9				
			7	8	6			
3	2	6					4	9
	1	9	4		5	2	8	
8								6

# Example: Kissing Number

How many unit balls with disjoint interior can be placed adjacent to a central unit ball in  $\mathbb{R}^d$ ?

In  $\mathbb{R}^2$



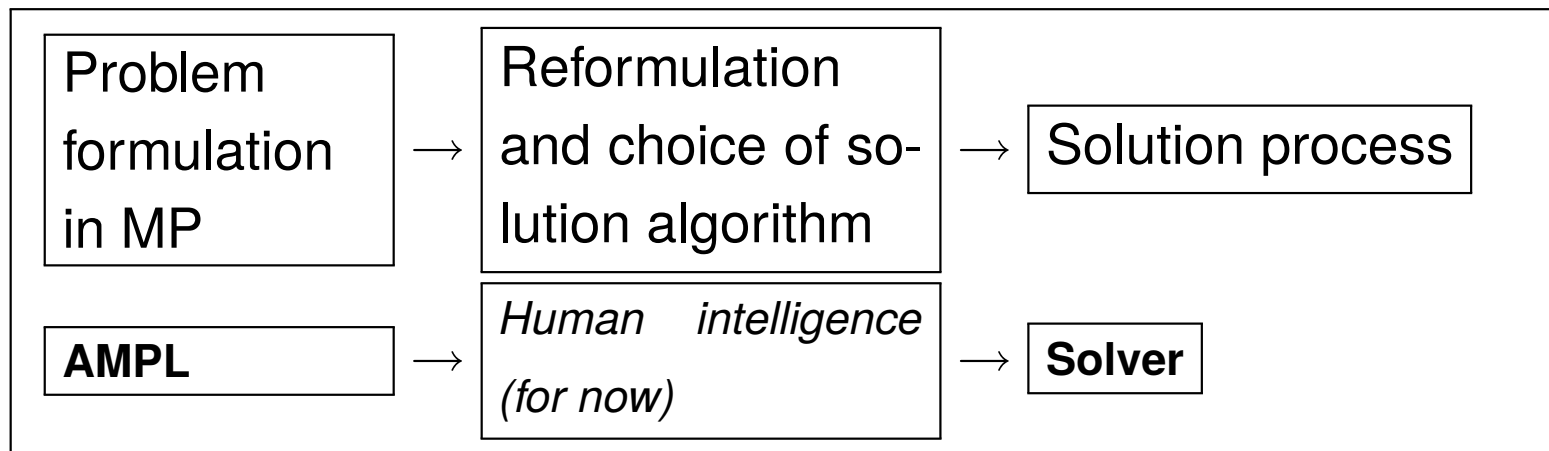
In  $\mathbb{R}^3$

( $D = 3$ : problem proposed by Newton in 1694, settled by [Schütte and van der Waerden 1953] and [Leech 1956])



# Mathematical programming

- The above three problems seemingly have *nothing* in common!
- Yet, there is a *formal language* that can be used to describe all three: **mathematical programming (MP)**
- Moreover, the MP language comes with a rich supply of solution algorithms so that problems can be solved right away





# Modelling questions

*Asking yourself the following questions should help you get started with your MP model*

- The given problem is usually a particular *instance* of a *problem class*; you should model the whole class, not just the instance (replace given numbers by parameter symbols)
- What are the decisions to be taken? Are they logical, integer or continuous?
- What is the objective function? Is it to be minimized or maximized?
- What constraints are there in the problem? Beware — some constraints may be “hidden” in the problem text

**If expressing objective and constraints is overly difficult, go back and change your variable definitions**



# Analysis

- What category does this mathematical program belong to?
  - Linear Programming (LP)
  - Mixed-Integer Linear Programming (MILP)
  - Nonlinear Programming (NLP)
  - Mixed-Integer Nonlinear Programming (MINLP)
- Does it have any notable mathematical property?
  - If an NLP, are the functions/constraints convex?
  - If a MILP, is the constraint matrix Totally Unimodular (TUM)?
  - Does it have any apparent symmetry?
- **Can it be reformulated to a form for which a fast solver is available?**





# Solvers



# Solvers

In order of solver reliability / effectiveness:

1. **LPs**: use an LP solver ( $O(10^6)$  vars/constrs, fast, e.g. CPLEX, CLP, GLPK)
2. **MILPs**: use a MILP solver ( $O(10^4)$  vars/constrs, can be slow, e.g. CPLEX, Symphony, GLPK)
3. **NLPs**: use a local NLP solver to get a local optimum ( $O(10^4)$  vars/constrs, quite fast, e.g. SNOPT, MINOS, IPOPT)
4. **NLPs/MINLPs**: use a heuristic solver to get a good local optimum ( $O(10^3)$ , quite fast, e.g. BONMIN, MINLP\_BB)
5. **NLPs**: use a global NLP solver to get an (approximated) global optimum ( $O(10^3)$  vars/constrs, can be slow, e.g. COUENNE, BARON)
6. **MINLPs**: use a global MINLP solver to get an (approximated) global optimum ( $O(10^3)$  vars/constrs, can be slow, e.g. COUENNE, BARON)

*Not all these solvers are available via AMPL*

# Solution algorithms (linear)

## ● **LPs:** (*convex*)

1. simplex algorithm (non-polynomial complexity but very fast in practice, reliable)
2. interior point algorithms (polynomial complexity, quite fast, fairly reliable)

## ● **MILPs:** (*nonconvex because of integrality*)

1. *Local* (heuristics): Local Branching, Feasibility Pump [Fischetti&Lodi 05], VNS [Hansen et al. 06] (quite fast, reliable)
2. *Global*: Branch-and-Bound (exact algorithm, non-polynomial complexity but often quite fast, heuristic if early termination, reliable)



# Solution algorithms (nonlinear)

## ● **NLPs:** (*may be convex or nonconvex*)

1. *Local:* Sequential Linear Programming (SLP), Sequential Quadratic Programming (SQP), interior point methods (linear/polynomial convergence, often quite fast, unreliable)
2. *Global:* spatial Branch-and-Bound [Smith&Pantelides 99] ( $\varepsilon$ -approximate, nonpolynomial complexity, often quite slow, heuristic if early termination, unreliable)

## ● **MINLPs:** (*nonconvex because of integrality and terms*)

1. *Local* (heuristics): Branching explorations [Fletcher&Leyffer 99], Outer approximation [Grossmann 86], Feasibility pump [Bonami et al. 06] (nonpolynomial complexity, often quite fast, unreliable)
2. *Global:* spatial Branch-and-Bound [Sahinidis&Tawarmalani 05] ( $\varepsilon$ -approximate, nonpolynomial complexity, often quite slow, heuristic if early termination, unreliable)



# MP language implementations

Software packages implementing (sub/supersets of the) MP language:

- **AMPL (our software of choice, mixture of MP and near-C language)**
  - commercial, but student version limited to 300 vars/constrs is available from [www.aml.com](http://www.aml.com)
  - quite a lot of solvers are hooked to AMPL
- **GNU MathProg (subset of AMPL)**
  - free, but only the GLPK solver (for LPs and MILPs) can be used
  - it is a significant subset of AMPL but not complete
- **GAMS (can do everything AMPL can, but looks like COBOL — ugh!)**
  - commercial, limited demo available from [www.gams.com](http://www.gams.com)
  - quite a lot of solvers are hooked to GAMS
- **Zimpl (free, C++ interface, linear modelling only)**
- **LINDO, MPL, ... (other commercial modelling/solution packages)**



# AMPL Basics



# AMPL

- AMPL means “A Mathematical Programming Language”
- AMPL is an implementation of the Mathematical Programming language
- Many solvers can work with AMPL
- AMPL works as follows:
  1. translates a user-defined model to a low-level formulation (called *flat form*) that can be understood by a solver
  2. passes the flat form to the solver
  3. reads a solution back from the solver and interprets it within the higher-level model (called *structured form*)



# Model, data, run

- AMPL usually requires three files:
  - the *model* file (extension `.mod`) holding the MP formulation
  - the *data* file (extension `.dat`), which lists the values to be assigned to each parameter symbol
  - the *run* file (extension `.run`), which contains the (imperative) commands necessary to solve the problem
- The model file is written in the MP language
- The data file simply contains numerical data together with the corresponding parameter symbols
- The run file is written in an imperative C-like language (many notable differences from C, however)
- Sometimes, MP language and imperative language commands can be mixed in the same file (usually the run file)

To run AMPL, type `ampl < problem.run` from the command line





# AMPL Grammar



# AMPL MP Language

- There are 5 main entities: sets, parameters, variables, objectives and constraints
- In AMPL, each entity has a name and can be quantified
  - `set name [{quantifier}] attributes ;`
  - `param name [{quantifier}] attributes ;`
  - `var name [{quantifier}] attributes ;`
  - `minimize | maximize name [{quantifier}]: iexpr ;`
  - `subject to name [{quantifier}]: iexpr <= | = | >= iexpr ;`
- Attributes on sets and parameters is used to validate values read from data files
- Attributes on vars specify integrality (`binary`, `integer`) and limit constraints (`>= lower`, `<= upper`)
- Entities indices: square brackets (e.g. `y[1]`, `x[i,k]`)
- The above is the basic syntax — there are some advanced options



# AMPL data specification

In general, syntax is in map-like form; a

```
param p{i in S} integer;
```

is a map  $S \rightarrow \mathbb{Z}$ , and each pair (domain, codomain) must be specified:

```
param p :=  
  1  4  
  2 -3  
  3  0;
```

The grammar is simple but tedious, best way is learning by example or trial and error



# LP example: .mod

```
# lp.mod  
param n integer, default 3;  
param m integer, default 4;  
set N := 1..n;  
set M := 1..m;  
param a{M,N};  
param b{M};  
param c{N};  
  
var x{N} >= 0;  
minimize objective: sum{j in N} c[j]*x[j];  
subject to constraints{i in M} :  
    sum{j in N} a[i,j]*x[j] <= b[i];
```



# LP example: .dat

```
# lp.dat
param n := 3; param m := 4;
param c          :=
    1  1
    2 -3
    3 -2.2 ;
param b          :=
    1 -1
    2  1.1
    3  2.4
    4  0.8 ;
param a : 1    2    3    :=
    1  0.1  0   -3.1
    2  2.7 -5.2  1.3
    3  1    0   -1
    4  1    1    0 ;
```



# LP example: `.run`

```
# lp.run  
  
model lp.mod;  
data lp.dat;  
option solver cplex;  
solve;  
display x;
```



# LP example: output

```
CPLEX 11.0.1: optimal solution; objective -11.30151  
0 dual simplex iterations (0 in phase I)  
x [*] :=  
1  0  
2  0.8  
3  4.04615  
;
```



# AMPL imperative language

- `model model_filename.mod ;`
- `data data_filename.dat ;`
- `option option_name literal_string, ... ;`
- `solve ;`
- `display [{quantifier}] : iexpr ; / printf (syntax similar to C)`
- `let [{quantifier}] ivar :=number ;`
- `if (clist) then { commands } [else { commands }]`
- `for {quantifier} {commands} / break ; / continue ;`
- `shell 'command_line' ; / exit number ; / quit ;`
- `cd dir_name ; / remove file_name ;`
- In all output commands, screen output can be redirected to a file by appending `> output_filename.txt` before the semicolon
- These are basic commands, there are some advanced ones