Introduction to Optimization

Author: Guanzhou (Jose) Hu 胡冠洲 @ UW-Madison CS524

Teacher: Prof. Michael Ferris

Figures included in this note are from Prof. Ferris's slides unless otherwise stated.

Definition of Optimization

Optimization Models

An **optimization model** is an *abstract* mathematical model that has the following three components:

1. **Decision variables**: variables representing the unknown quantities
   - Can define *definitive variables* as expressions of decision variables
2. **Constraints**: requirements that any *feasible* solution must satisfy

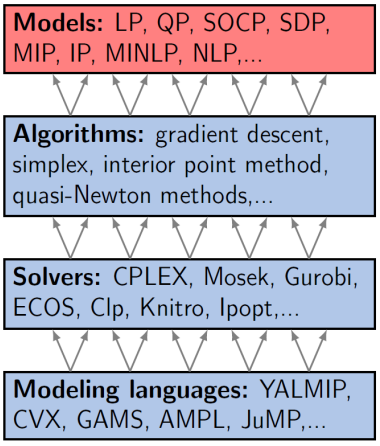
3. **Objective**: a quantity to be *maximized*/*minimized*, expressed as a function of decision variables
    - If to be minimized, can also call it *cost function*

Any concrete *decision problem* (*story*) can be abstracted into an optimization model, which we then be able solve with the help of software. The main purpose of this course is to learn to build correct, concise, & efficient optimization models given optimization problems.

Be aware that not all decision problems are best answered by optimization models.

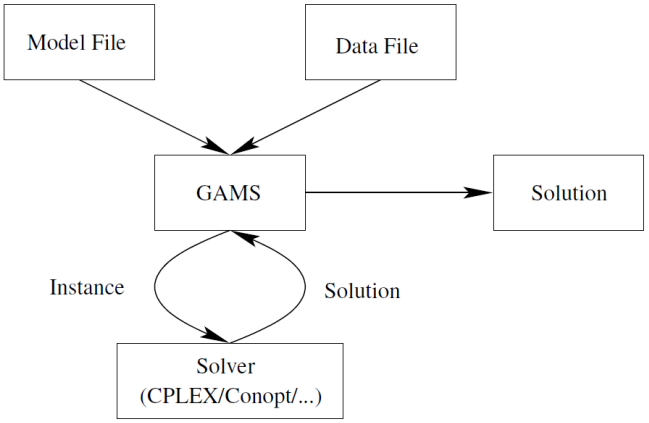
The Optimization Hierarchy

The software stack of mathematical modeling forms the following hierarchy:



- *Models* can be categorized on the types of variables, types of constraints, or type of objective
    - *Linear programming* (LP) vs. Non-linear
    - *Mixed-integer programming* (MIP, discrete) vs. Continuous
    - Convex vs. Non-convex
    - Deterministic vs. Stochastic
- *Algorithms* are numerical (usually iterative) procedures that can solve certain instances of optimization models
    - More specialized algorithms are usually faster
- *Solvers* are collections of implementations of algorithms
- *Modeling languages* provide an interface to many different solvers using common language

Take the **GAMS** software stack as an example, the procedure looks like:



Solution Analysis

Optimization is *extreme*, so if there is a mistake in your model, applying the software will usually find it.

- *Verification*: is the model itself correct?
- *Validation*: does the model give an accurate picture of reality?
- *Sensitivity analysis*: how much are extra resources worth to me? (*marginal* information)
- *What-if analysis*: change the instance and re-run

Linear Programming (LP)

Definition of LP

A *linear function* $f : \mathbb{R}^n \to \mathbb{R}$ is a weight sum:

$$f(x) = \sum_i c_i x_i = c^T x$$

Linear inequalities are inequalities in the form of $a^T x \leq b$ or $a^T x \geq b$. They define *half-spaces* of $\mathbb{R}^n$.

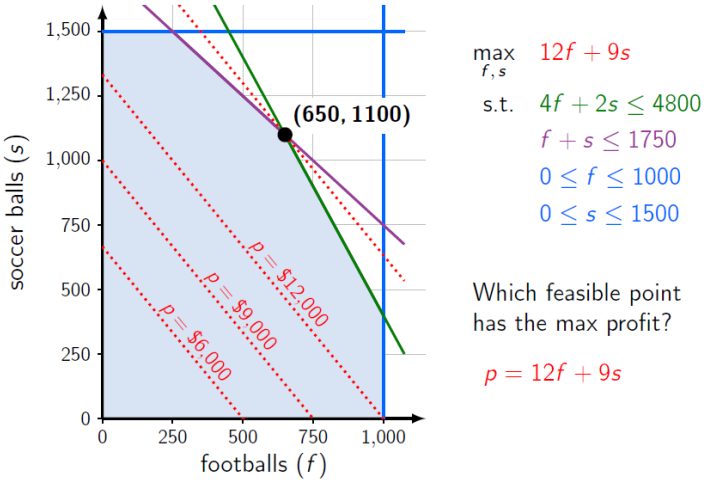
A **linear program** is an optimization problem where the objective function is a linear function and all of the constraints are linear inequalities.

- *Proportionality*: given different *levels* (values of decision variables), inputs/outputs/costs must keep the same proportions as levels change
  - Fixed-charges (make 0 = \$0, make ≥ 1 = \$C) cannot be modeled by pure linear programs
- *Additivity*: outputs are additive (linear) function of input variables
- *Divisibility*: decision variables can take any real value between upper and lower bounds
  - For discrete problems, one possible way could be *rounding-off* result to nearest integer, but it may produce a solution that is far from optimal

See the `topbrass` model example.

Solving LP Visually

For 2-dimensional LP with only 2 decision variables, high school knowledge teaches us how to visualize an LP problem and solve it intuitively:



However, for higher-dimensional problems, this method is neither easy nor efficient.

The Simplex Method

One famous method for solving LP is the *simplex method*. The observation is that for an LP instance that has an optimal solution, there exists an optimal solution at an *extreme point* of the feasible region.

On an n-dimensional LP:

1. Start from an extreme point
   - An inequality constraint $a^T x \leq b$ is binding at x if $a^T x = b$
   - An extreme point is the intersection of at least n inequalities
   - The indices of the n inequalities that are binding at an extreme point is called a *basis*
2. Find a direction along an edge, where the objective function is improving
   - If none exists, STOP: this point is an optimal solution
3. Move along that direction until hitting a new extreme point, repeat

The downsides of the simplex method are:

- The number of extreme points can grow wildly. For an n-d problem with m constraints, $E(m, n) \leq \binom{m}{n}$

- An LP problem could be *unbounded*: moving along an edge does not let you hit a new extreme point
  - Usually this means you forgot some constraints
  - But, an unbounded feasible region doesn't mean the LP is unbounded: the optimal extreme point may still exist but other directions are unbounded
- (subtle) There could be multiple optimal solutions along an edge

What Could Go Wrong

When solving real LP problem, the model does not always end in good status. Cases when things could go wrong:

- **Unbounded** model: cannot find a finite optimal solution
- **Infeasibility**: sometimes the constraints of a model could be too strict that there are no feasible solutions. In this case, one thing we could do is to allow one of the constraints to *slack* a bit
  - We introduce a *slack/surplus* variable and try to minimize it
  - After getting the minimal amount of "slack", use that to relax the chosen constraint and re-solve the previous model. See the `mcgreasy` model example
- **Degenerate LP**: when more than n inequalities intersect at a point. Solvers typically perturb the inequalities when finding intersections

Ordering & Conditions

Sometimes we want to model a *time-series* problem: e.g., decision variable at the end of a month depends on what was left at the end of last month. This is called **multi-period planning**.

GAMS provides three features around sets that are useful for such modelling:

- *Dynamic set* (doc): set declared over some *static set* (universe known at compile time), whose membership could change during execution time
  - Useful for controlling indices and conditions
  - Equations must be declared over static sets
  - But, can define equations over dynamic sets, so that the model focuses on a sub-scope
- *Set ordering* (doc): a 1-dimensional set is ordered if the initialization of elements corresponds to the order of their initial appearance in the GAMS file; an ordered set can be thought of as an array
  - `ord(I)` gives the order of current element in set `I`; `card(I)` gives the size (cardinality) of `I`
  - On a set in equation definition, can use *lag/lead* operators `+` / `-` to refer to the predecessor/successor of the current element. If non-exist, 0 will be used
- `$` *conditioning* (doc): write a `$` followed by a logical condition after an expression to control when it appears; these are expanded at compilation time. Examples:
  - `A(I) = 1$(ord(I) <= 2) + X(I-2)$(ord(I) > 2);`
  - `inventory_eq.. I0$(ord(T) = 1) + I(T-1) + x(T) =e= d(T) + I(T)`
  - `A(D)$weekday(D) = B(D)`

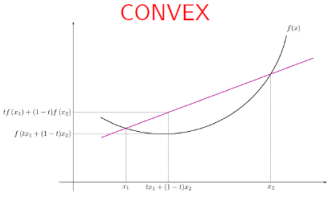
See the `shoeco` model example.

Convexity & Backlogging

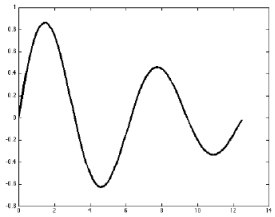
The **convexity** property of objective function is essential. Convex (or concave) function means that a local optimum is guaranteed to be the global optimum as well, which would greatly simplify the solving. A function f is called

- *Convex* if for any two points x, y, the graph of f lies below or on the straight line connect $(x, f(x))$ and $(y, f(y))$
- *Concave* if ... above or on ...
- *Nonconvex* if neither

CONVEX



NONCONVEX



Mathematical sets could also have convexity. A *polyhedral set* is the intersection of a finite number of halfspaces (not anything curvy or discrete). Such set S is convex if the straight line segment connecting any two points in the S lies entirely inside or on the boundary of S.

- Function f is convex if the *epigraph* (overpart) of f is a convex set
- Any discrete set of multiple data points is nonconvex
- LP could be easily solved if the *feasible region* is a convex set; we would like to preserve convexity as much as possible

One example of preserving convexity is when we allow **backlogging** (negative inventory/capitol) in the model. See the `shoeco2` model example.

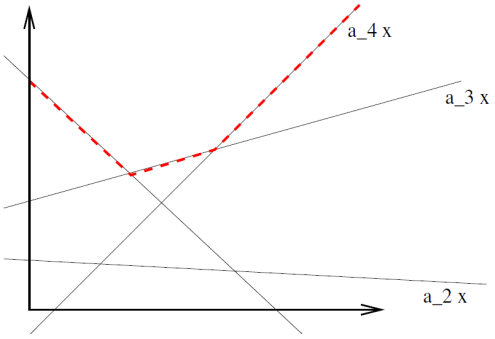
- Cost function becomes a *piece-wise linear function* with 2 pieces - still convex
- A trick is to model it with two positive variables and take the difference: $I_t = L_t - S_t$ (leftover $-$ shortage)
    - Objective gets terms $\sum_t (aL_t + bS_t)$, pay a storage cost per unit of leftover, pay b cost per unit of shortage to makeup in the future
    - Works only if we are minimizing the cost objective - at most one of L_t and S_t will be positive at any timepoint (the other one being zero)
    - Add constraint on last period $I_{|T|} \geq 0$ to say that all demand must be eventually met in our planning horizon

This trick applies to any definition variable v whose value is a convex piecewise-linear function. Define separate positive variables for each pieces, and add a constraint saying that the v is a sum/subtraction of them.

Minimax For Uncertainty

In reality, we don't always know the exact values of parameters. Such **uncertainty** means that we have to consider multiple scenarios where parameters could take different sets of values.

The *minimax* problem generates a model where we try to "minimize the maximum cost" we pay under any scenario.



The objective value z is now not an exact expression. Instead, it satisfies a constraint that $z \geq f(x_s), \forall s \in S$, where S is the collection of all possible scenarios, and we try to $\min z$. See the `shoeco3` model example.

Blending Constraints

Consider the case where a constraint says some variable must be of a specified ratio to some other variable. The *ratio* constraint is typically not a linear constraint.

A trick to convert such problem into a linear problem is called **blending**. Suppose that the denominator is strictly positive, we multiple both sides by the denominator. This transforms a ratio constraint to a linear constraint.

Min-Cost Network Flow (MCNF)

A **network** is a directed, possibly weighted graph. Vertices are called *nodes* and edges are called *arcs*.

- A network can be represented as a *node-arc incidence matrix* A, where $a_{i,j} = 1$ if arc j leaves node i, and $a_{i,j} = -1$ if arc i enters node i; If the network is *capacitated*, arcs have weights representing its capacity
- Matrix A is *totally unimodular* (TU) if every sub-determinant of A has value ± 1 or 0
    - Network incidence matrices are TU
    - Graph incidence matrices of *bipartite* graphs are TU
    - If we solve a linear program $Ax \leq b$ whose constraint matrix A is TU and b is integer valued, then the solution will be integer valued, which means we do not need to impose integrality on variables: LP methods work

Take the assignment problem (`assignpref`) as an example, we could model that with binary variables and solve with `mip`, but since the constraint is TU, doing `lp` is feasible as well. GAMS provides `rmip` (relaxed MIP) which automatically detects these situations and falls back to LP when doable.

A **min-cost** *network flow* (MCNF) problem on a network says that:

- Every node i has a supply/demand b_i: $b_i > 0$ means supply, $b_i < 0$ means demand
- Every arc a may have cost c_a and capacities u_a
- All demand must be met exactly: $\sum_i b_i = 0$; Often add a dummy node to account for this
- The goal is to find a minimum cost flow from supply to demand to fill the demands, without exceeding any arc capacity

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij}, \text{ s.t.}$$
$$\sum_{k:(i,k)\in A} x_{ik} - \sum_{j:(j,i)\in A} x_{ji} = b_i, \forall i \in N$$
$$0 \le x_{ij} \le u_{ij}, \forall (i,j) \in A$$

See the `mincost*` model example. The assignment problem is one example of MCNF, where we model every student and project as a node, and link arcs from every student node to every project node. Students supply 1 and project demand 1. Since there must be an integer solution, we need not restrict x_{ij} to be binary.

Many "cost minimization" problems could be transformed into an MCF problem if we draw out network graph. See the `sailco*` model example.

LP Duality

For an LP problem in the form:

$$z^* = \min c^T x, \text{ s.t. } Ax \ge b, x \ge 0$$

The **dual** form of this LP problem is:

$$w^* = \max b^T \pi, \text{ s.t. } A^T \pi \le c, \pi \ge 0$$

Intuitively, in the dual problem, we are trying to find a linear combination of the constraints of the **primal** problem that gives us the best guess of its lower bound.

- See slide #12 for an example
- Duality is converse: the dual problem of the dual problem is the primal problem
- Sometimes people use the perpendicular ($\perp$) symbol to denote duality

To construct the dual problem of any LP problem:

- One way is to convert the problem into *standard form* as described above
- A faster way is to first just transpose the data, then look up this table and write out the constraint types:

Min problem	Max problem
Nonnegative variable $\ge$	Inequality constraint $\le$
Nonpositive variable $\le$	Inequality constraint $\ge$
Free variable	Equality constraint $=$
Inequality constraint $\ge$	Nonnegative variable $\ge$
Inequality constraint $\le$	Nonpositive variable $\le$
Equality constraint $=$	Free Variable

Why we care about LP duality?

- **Marginal** values of the dual constraints are optimal values of the primal variables; vice versa, marginal values of the primal constraints are optimal values of the dual variables
- Marginal values of primal constraints (i.e., dual variables) are sometimes called *shadow prices*: they tell us how much would the objective value change if we change the right-hand side of some constraint by 1
- Marginal values of primal variables are sometimes called *reduced costs*: they tell us how much would the objective value change if we change the variable bounds by 1

Shortest Path Problem (SPP)

If optimization programming is the only tool we have, we can model a shortest path problem into a MCNF problem.

- Set a supply of 1 at *source* node and a demand of 1 at *sink* node (destination node)
- Set a supply of zero at all other nodes

If we look at the dual problem of an SPP MCNF problem, it has an interesting interpretation: each π_i is the minimum path length from node i to sink. We are actually solving more than just the shortest path from source to sink, but the shortest path from any node to sink, along the way.

Max-Flow & Min-Cut

In contrast to MCNF, we have **max-flow** problem, where every arc in the network has a capacity and we are trying to determine what is the maximum amount we can push from source node to sink node.

$$\max \sum_{j:(s,j)\in A} x_{sj} \text{ or } \max \sum_{i:(i,t)\in A} x_{it} \text{ (equivalent), s.t.}$$
$$\sum_{k:(i,k)\in A} x_{ik} - \sum_{j:(j,i)\in A} x_{ji} = 0, \forall i \in N - \{s,t\}$$
$$0 \le x_{ij} \le u_{ij}, \forall (i,j) \in A$$

See the `picnic` model for an example. Writing out the dual problem of a max flow problem is a good exercise (see slide #14 for the conversion formally).

An equivalent way of describing a max flow problem is in terms of **paths**. We enumerate all unique paths from source to sink and maximize the sum of flows through all paths, with the constraint that for any arc, the sum of flow of paths going through that arc is limited by its capacity.

$$\max \sum_{p\in P} f_p, \text{ s.t.}$$
$$\sum_{p\in P:(i,j)\in p} f_p \le u_{ij}, \forall (i,j) \in A$$

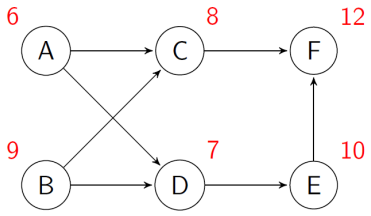
The dual problem of the path description turns out to be straight-forward.

$$\min \sum_{(i,j)\in A} u_{ij}\pi_{ij}, \text{ s.t.}$$
$$\sum_{(i,j)\in p} \pi_{ij} \ge 1, \forall p \in P$$

This dual form is called the **min-cut** problem, which identifies the minimum *critical* arcs one need to take out to cut out source from sink. In other words, max-flow problem $\equiv$ min-cut problem.

Critical Path Analysis

Imagine a project consisting of multiple activities with dependencies on each other (forming a DAG). Each activity takes certain amount of duration. The goal is to find a plan that completes the final stage as soon as possible. This is often called *project evaluation & review technique* (PERT) or **critical path** method (CPM).



To model this problem, we:

- Use a_i to denote the duration of activity i
- Create decision variables t_i that represent the time we start doing activity i
- The constraints are that any activity can only start all of its predecessors have finished: $t_j \ge t_i + a_i, \forall (i,j) \in A$
- Model it as a minimax problem: $\min z, \text{ s.t. } z \ge t_i + a_i, \forall i \in I$

The dual problem of PERT helps us identify the critical path of the project flow: if we delay an activity on the critical path, the overall completion time increases. See the `widgetco` model for an example. All the λ variables that have positive level values are on the critical path (critical paths may not be unique - this only gives us one critical path).

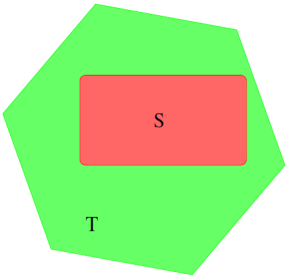
After solving PERT, if we fix the total duration and try to minimize/maximize the sum of t_i's, we can get the earliest/latest time when we can start activity i. For activities on the critical path, their earliest time $=$ latest time (this finds all activities on all critical paths).

Mixed Integer Programming (MIP)

Whenever a model involves *discrete* variables, it becomes an **integer program** (IP). Integer programs are much harder than linear programs: you solve them optimally by either 1) enumerate all possible values of discrete variables if their feasible sets are all finite, or 2) using techniques to relax them to LP and solve a bunch of LP problems.

Relaxation to LP

Relaxation is a technique in optimization where you relax the feasible region of some variables to be larger. For example, for the problem $z_S = \max f(x) : x \in S$, the problem $z_T = \max f(x) : x \in T, S \subset T$ is a relaxation.



It is trivial to see that the optimal solution of the relaxed problem $z_T \geq$ the optimal solution of the original problem z_S. (If it is a minimization problem, everything is just flipped.) This opens up an opportunity: if we get lucky and the optimal solution for the relaxed problem $x_T^* \in S$, then we successfully find the optimal solution to the original problem x_S^* as well.

For a MIP problem with constraints $x_j \in \mathbb{Z}, j \in Z \subseteq N$ (the set of variables), there is a relaxation to LP by removing all the integer constraints.

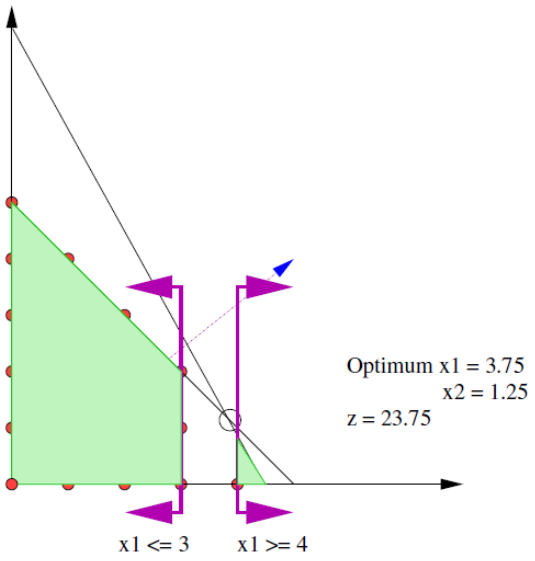
Branch & Bound

The **branch & bound** method is an approach to solving MIP problems based on relaxation:

1. Initialize the following things:
    - A list of open problems L_O, put in the original IP problem
    - A list of candidate solutions L_C, empty
    - A global lower bound $z_L = -\infty$; A global upper bound $z_U = +\infty$
2. Check if $z_L = z_U$, terminate - we found the solution
3. Pop a problem P from the list L_O
4. Solve the LP relaxation Q of it:
    - If Q is infeasible, then P must also be infeasible, go back to 2.
    - Otherwise, let x^* be the optimal solution of Q, and z^* be its objective value. Put the solution in L_C:
        - If x^* satisfies the integer restrictions of P, then z^* is the optimal value of P
        - Else, x^* does not satisfy the integer restrictions of P, we might need to *branch*:
            - If $z^* \leq z_L$ (*fathom*), all branches down this path won't be better than what we have seen
            - Otherwise, divide the problem. Choose some x_k^* that does not satisfy the integer restriction. Add two new open problems to L_O:
                - P with added constraint $x_k \leq \lfloor x_k^* \rfloor$
                - P with added constraint $x_k \geq \lceil x_k^* \rceil$
        - Either way, check if any problem in L_C was previously branched and both two branches have been checked - if so, remove it from L_C. Update $z_L = \max(z_L, z^*)$, update $z_U = \max(z)$ of all current z's in L_C. Go back to 2.

This method is built around two essential ideas:

- *Branching*: if we get unlucky on an LP relaxation, branch the problem into two by splitting the feasible region of some variable that does not satisfy the integer restriction; retry on the two branched problems



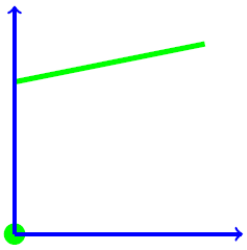
- *Bounding*: the algorithms maintains a lower bound z_L and an upper bound z_U, which gives us an idea of how close we are to the optimal solution; we can terminate the algorithm early when the best solution we found so far is within α% of optimality (i.e., $\frac{(z_U - z_L)}{\max(|z_L|, 1)} \geq \alpha\%$)

This is what happens behind the scenes of IP program solvers. Notice how much harder is an IP problem compared to LP - we solve relaxed LP problems many times in order to get the solution for an IP.

Fixed Costs & Indicators

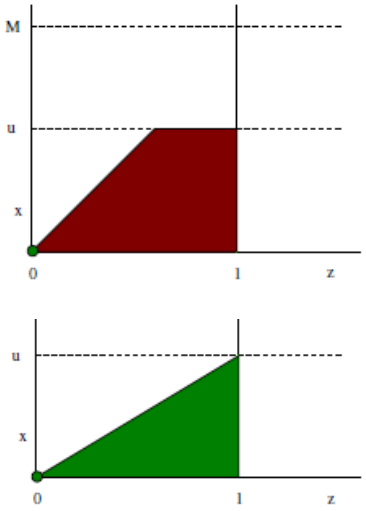
A typical situation where we must deal with IP models is the *fixed costs* scenario. For each type of product i, if we produce none of it, we do not need to rent the machine for it, so the cost $F(x_i)$ is 0. If we produce even a tiny amount of it, we must pay a fixed cost of f_i plus a linear cost of $c_i x_i$.

$$F(x_i) = \begin{cases} 0, & x = 0 \\ f_i + c_i x_i, & x > 0 \end{cases}$$



The "absolute value" trick / "piece-wise linear" trick we used in LP do not work here. To model fixed costs, the key idea is to introduce **indicator variables**. Use binary variables δ_i to indicate whether $x_i > 0$ or not.

- The constraint is now in the form $x_i \leq M_i \delta_i$ where M's are some positive scalars, and $\delta_i \in \{0, 1\}$
- This is mathematically equivalent to the logical implication of $x_i > 0 \Rightarrow \delta_i = 1$
- Small M's are good, while big M's are bad (bigger M's make the LP relaxation of the problem looser) - we should set M to be as close to the real upper bound u of x as possible (but cannot be smaller than u, otherwise it changes the problem)



Generalized Assignment Problem (GAP)

With fixed costs considered, we introduce the *generalized assignment problem* (GAP), where we have:

- A set of machines $W = \{1, \dots, m\}$, a set of jobs $N = \{1, \dots, n\}$
- Each machine i has a capacity of b_i units of work
- Each job j requires a_{ij} units of work to be completed if scheduled on machine i
- A job must be assigned to exactly one machine
- There's a fixed cost h_i if any job is assigned to machine i, otherwise the machine won't be started/bought so there's no cost

How fixed cost logic is encoded can greatly affect the efficiency of solving GAP with different solvers. There are several ways to encode this:

- $\sum_{j \in N} x_{ij} \le M z_i, \forall i \in W$: use $M = |N|$
  - The most general and loose form
- $\sum_{j \in N} x_{ij} \le M_i z_i$: use $M_i = |\{j : j \text{ is possible to be assigned to } i\}|$
  - Stricter than the above form
- $x_{ij} \le M z_i$: use $M = 1$
  - The most strict form, so theoretically the best encoding
  - But introduces too many constraint equations, probably slow on actual solver software

The above GAP problem is still modeled by $f(x_i) > 0 \Rightarrow \delta_i = 1$. What if we need $f(x_i) \ge 0 \Rightarrow \delta_i = 1$? This is not always doable, but in some cases, a trick is to find some ϵ s.t. $f(x_i) + \epsilon > 0 \equiv f(x_i) \ge 0$, then use $f(x) + \epsilon$ instead. See slide #18 and `gapext` model examples.

Logic Models

Indicator variable is a good tool for encoding **logic implications** into doable constraints.

- We have seen that to enforce $f(x) > 0 \Rightarrow \delta = 1$ we use constraint $f(x) \le M\delta$
- Conversely, to enforce $\delta = 1 \Rightarrow f(x) \le 0$ we use constraint $f(x) \le M(1 - \delta)$

Other logic constraints that could be modeled using indicator variables:

- *Variable lower bound*: if you produce product j, product at least l_j units of it
  - $x_j > 0 \Rightarrow z_j = 1 \Rightarrow x_j \ge l_j$
  - Use $x_j \le M_j z_j$ and $x_j \ge l_j z_j, \forall j$
- *Either-or constraints*: either $f(x) \le 0$, or $g(x) \le 0$
  - $f(x) > 0 \Rightarrow \delta = 1 \Rightarrow g(x) \le 0$
  - Use $f(x) \le M\delta$ and $g(x) \le M(1 - \delta)$

More generally, if we have a set N of candidate constraints and we want to impose general logic relations on them, we use a set of indicator variables δ_i, where $\delta_i = 1 \Rightarrow$ constraint i is turned on (must be satisfied). Note that when $\delta_i = 0$, the constraint may or may not be satisfied (we do not enforce that). Then, we can add MIP constraints on the indicator variable values to represent the logic relations:

General Statement	MIP Constraint
at least k out of n are true	$\sum_{i=1}^{n} \delta_i \geq k$
$P_1 \vee P_2 \vee \cdots P_n$	$\sum_{i=1}^{n} \delta_i \geq 1$
exactly k out of n are true	$\sum_{i=1}^{n} \delta_i = k$
at most k out of n are true	$\sum_{i=1}^{n} \delta_i \leq k$
if at least k out of n are true then $n+1$ is true	$\delta_{n+1} \geq \frac{\sum_{i=1}^{n} \delta_i - k + 1}{n-k+1}$
$(P_1 \wedge \cdots P_k) \rightarrow (P_{k+1} \vee P_n)$	$\sum_{i=1}^{k}(1-\delta_i) + \sum_{i=k+1}^{n} \delta_i \geq 1$

	Statement	MIP Constraint
1.	$\neg P_1$	$\delta_1 = 0$
2.	$P_1 \vee P_2$	$\delta_1 + \delta_2 \geq 1$
3.	$P_1 \veebar P_2$	$\delta_1 + \delta_2 = 1$
4.	$P_1 \wedge P_2$	$\delta_1 = 1,\ \delta_2 = 1$
5.	$\neg(P_1 \vee P_2)$	$\delta_1 = 0,\ \delta_2 = 0$
6.	$P_1 \rightarrow P_2$	$\delta_1 \leq \delta_2$
7.	$P_1 \rightarrow (\neg P_2)$	$\delta_1 + \delta_2 \leq 1$
8.	$P_1 \leftrightarrow P_2$	$\delta_1 = \delta_2$
9.	$P_1 \rightarrow (P_2 \wedge P_3)$	$\delta_1 \leq \delta_2,\ \delta_1 \leq \delta_3$
10.	$P_1 \rightarrow (P_2 \vee P_3)$	$\delta_1 \leq \delta_2 + \delta_3$
11.	$(P_1 \wedge P_2) \rightarrow P_3$	$\delta_1 + \delta_2 \leq 1 + \delta_3$
12.	$(P_1 \vee P_2) \rightarrow P_3$	$\delta_1 \leq \delta_3,\ \delta_2 \leq \delta_3$
13.	$P_1 \wedge (P_2 \vee P_3)$	$\delta_1 = 1,\ \delta_2 + \delta_3 \geq 1$
14.	$P_1 \vee (P_2 \wedge P_3)$	$\delta_1 + \delta_2 \geq 1,\ \delta_1 + \delta_3 \geq 1$

$P_n \leftrightarrow (P_1 \wedge \cdots \wedge P_k)$	$\delta_n + k \geq 1 + \sum_{i=1}^{k} \delta_i,\ \delta_j \geq \delta_n,\ j = 1, \ldots, k$
	(or equivalently) $\delta_n = \min(\delta_1, \ldots, \delta_k)$
	(or equivalently) $\delta_n = \delta_1 \times \delta_2 \times \cdots \times \delta_k$
$P_n \leftrightarrow (P_1 \vee \cdots \vee P_k)$	$\sum_{i=1}^{k} \delta_i \geq \delta_n,\ \delta_n \geq \delta_j,\ j = 1, \ldots, k$
	(or equivalently) $\delta_n = \max(\delta_1, \ldots, \delta_k)$

Also see slides #23 for formulations of $y = x\delta$, $y = \min x_i$, $y = \max x_i$, and $y = |x_1 - x_2|$, on continuous variables x_i 's.

Quadratic Assignment Problem (QAP)

The *quadratic assignment problem* (QAP) says that we want to assign facilities to locations, s.t. the sum of flow costs between pairs of facilities is minimized.

$$\min \sum_{k \in F, l \in F, i \in L, j \in L} d_{ij} f_{kl} x_{ki} x_{lj}, \text{ s.t.}$$

$$x_{ij} = \begin{cases} 1, & \text{assign facility } i \text{ to location } j \\ 0, & \text{otherwise} \end{cases}$$

$$\sum_{i \in F} x_{ij} = 1, \forall j \in L$$

$$\sum_{j \in L} x_{ij} = 1, \forall i \in F$$

A modeling trick we could use is to linearize the product of the two binaries to reduce the "discreteness" of this problem:

- $z_{klij} = 1 \Leftrightarrow x_{ki} = 1, x_{lj} = 1 \Leftrightarrow x_{ki} \wedge x_{lj}$
- Use $z_{klij} \leq x_{ki}$, $z_{klij} \leq x_{lj}$, and $z_{klij} \geq x_{ki} + x_{lj} - 1, \forall \cdots$

Set Covering

The *set covering* problem states that given a set of objects S and a collection of subsets $S_1, \ldots, S_n$ each associated with a cost c_j of being selected, choose a set of subsets that cover all the member elements of the base set with minimum cost.

To formularize this, let decision variable $x_j = 1$ if S_j is chosen in the result cover.

$$\min \sum_j c_j x_j, \text{ s.t.}$$
$$x_j \in \{0, 1\}, \forall j$$
$$\sum_{j \text{ that } i \in S_j} x_j \geq 1, \forall \text{ object } i$$

See the `simpsetcov` model for the basic model, and `kilroy` and `vr` models for real applications. A similar category of problems is the *cutting sheet* problem where x's are integers and each object need to be covered at least d_i times. See the `cutsh` and `cutstock` models.

Similarly, there are *set packing* problem (no overlapping) and *set partitioning* problem (exactly covered once).

Column Generation

The above formulation, though most complexity goes in data calculation, may get very inefficient/infeasible when our data is big. A practical technique is called *column generation*, where we start from a first few k columns and put in columns that are deemed likely to help reduce the cost:

1. Start with columns $A = [a_1, \dots, a_k]$
2. Solve relax MIP (`rmip`): minimize $\sum_j x_j$ subject to $Ax \geq d$
3. Obtain the dual variable λ to the $Ax \geq d$ constraint. If we add in a new column a, cost will drop by $a^T \lambda$
4. Suppose $a = [y_1, \dots, y_n]$, solve strict MIP (`mip`): maximize $a^T \lambda$ subject to $a^T d \leq u$ (e.g., the length of pipe, the area of sheet, etc.)
5. If the above MIP is feasible, add new column a^* to A. Repeat from step 2.

An interesting application of column generation is to the **traveling salesman** problem. See slide #22 for details.

Special Ordered Set (SOS)

We may want to model the constraint that a variable x can only take one (and must take one) value from a restricted set $\{a_1, \dots, a_m\}$. We introduce a *special ordered set* (SOS) of binary variables $\{\delta_1, \dots, \delta_m\}$, indicating which value does x take:

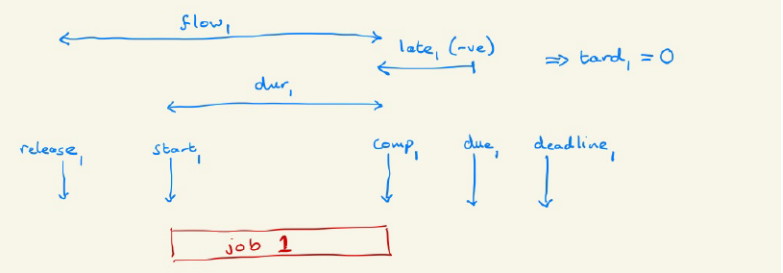
$$x = \sum_{j=1}^{m} a_j \delta_j$$
$$\sum_{j=1}^{m} \delta_j = 1$$

GAMS has the `sos1` variable type for doing this -- e.g., `sos1 variable s(A, B, SS)` declares an SOS set for each pair of elements in `(A, B)`, within each SOS the variables indexed `SS` are mutually exclusive. See the `warehouse` model example.

Merging MIP with **constraint logic programming** (CLP) is an active area of research. Techniques used in CLP are essentially clever ways to do enumerations efficiently.

Ordering & Scheduling

Ordering, in the context of logical models, means the arrangement of events. Though ordering problems are hard to solve with pure MIP, we consider a specific problem context of *job scheduling* here:



- Completion time $\text{comp}_j = \text{start}_j + \text{dur}_j$
- Flow time $\text{flow}_j = \text{start}_j + \text{dur}_j - \text{release}_j$
- Lateness $\text{late}_j = \text{start}_j + \text{dur}_j - \text{due}_j$

- Tardiness $\text{tard}_j = \max\{0, \text{late}_j\}$
- Makespan is the maximum completion time overall: $\text{makespan} = \max_j \text{comp}_j$

The objective could be to find out an ordering of jobs that minimizes makespan, minimizes lateness, etc.

Suppose job cannot be done in parallel (single-core CPU), to formulate this scheduling problem in MIP, there are several options:

1. Use binary variables r of ranking: $r_{ik} = 1$ iff job i has position k

$$\sum_i r_{ik} = 1, \forall k$$
$$\sum_k r_{ik} = 1, \forall i$$
$$\text{comp}_k \geq \text{comp}_{k-1} + \sum_j \text{dur}_j r_{ik}$$

   To let solvers run faster, declare r as SOS so that the solvers inherently restrict that only one position can be 1.

2. Use binary variables o of ordering: $o_{ij} = 1$ iff job i finishes before job j starts; this is an either-or constraint

$$\text{comp}_i \leq \text{start}_j + M(1 - o_{ij})$$
$$\text{comp}_j \leq \text{start}_i + Mo_{ij}$$

3. Introduce "violation" variables as SOS: violation_{ij} = `SOS(I, J, 's1')` and violation_{ji} = `SOS(I, J, 's2')`, so that either violation_{ij} is true or violation_{ji} is true

$$\text{comp}_i \leq \text{start}_j + \text{violation}_{ij}$$
$$\text{comp}_j \leq \text{start}_i + \text{violation}_{ji}$$

See the `jobshop` and `flowshop` model examples.

Non-Linear Optimization

The real world is not linear and not convex. It is important to look beyond linear optimization and integer enumeration into non-linear programming. We will still focus on techniques to reduce them into MIP or LP.

Piecewise Linear Function (PLF)

To model an arbitrary continuous function, a common structured used is a *piecewise linear function* (PLF) to **approximate** the original function.

$$f(x) = m_i x + c_i, x \in [B_{i-1}, B_i], \forall i \in [1, n]$$

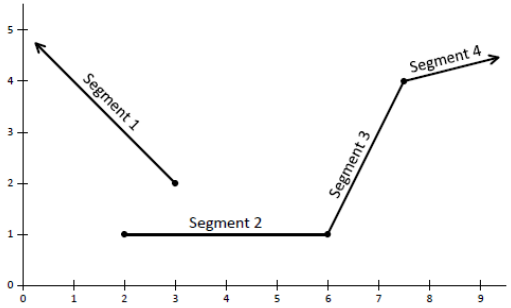
Assume now we know the PLF to use, we model the PLF by introducing new binary variables to indicate which is the current segment: $b_i = 1$ if $x \in [B_{i-1}, B_i]$ and $w_i = x$ if $x \in [B_{i-1}, B_i]$. At any point x:

$$x = \sum_i w_i \qquad f(x) = \sum_i (m_i w_i + c_i b_i)$$
$$B_{i-1} b_i \leq w_i \leq B_i b_i, \forall i \in [1, n]$$
$$\sum_i b_i = 1$$

To incorporate constraints like the fixed cost, we want the ability to "turn off" the function when some $\delta = 0$. Instead of adding a big-M constraint $x \leq B_n \delta$, we change the sum-of-b constraint into $\sum_i b_i = \delta$. This is a *locally ideal* formulation, where every extreme point of LP relaxation satisfies the discrete requirements.

This formulation cannot model two scenarios:

- Multi-valued PLF (which isn't a mathematical function)
- Infinite domain (x could go to infinity; l_i gives the length of segment, negative means described as growing negatively)



	x	f	l	g
Seg. 1	3	2	-∞	-1
Seg. 2	2	1	4	0
Seg. 3	6	1	$\sqrt{11.25}$	2
Seg. 4	7.5	4	∞	¼

We model such "graph" by introducing: binary variables z_i telling me which segment is chosen and variables λ_i telling how far I go from the starting point (x, f) of that segment. This gives one exact point on the graph.

$$x = \sum_i (x_i z_i + \text{sign}(l_i)\lambda_i) \qquad f = \sum_i (f_i z_i + \text{sign}(l_i) g_i \lambda_i)$$

$$\sum_i z_i = 1$$

$$\lambda_i \leq |l_i| z_i, \text{ for each segment that has finite length } l_i$$

To model segments of infinite length, add an SOS variable saying that for each segment, either λ_i or $1 - z_i$ is positive.

Quadratic Programming (QP)

A **quadratic function** is a sum of terms of the form $q_{ij} x_i x_j$, i.e., each term is at most of degree 2. Any quadratic function can be written in the form $x^T Q x$, where Q is a *symmetirc matrix* ($Q^T = Q$).

A recap in linear algebra:

- Symmetric Q can be written in the form $Q = V \Lambda V^T$, where each column of V is an *eigenvector* of Q and Λ is a diagonal matrix where each value on the diagnosis is the *eigenvalue* λ_k for the corresponding eigenvector
- *Positive semidefinite matrix*: if Q is a symmetric and all eigenvalues of Q are non-negative, then Q is positive semidefinite, and the quadratic function defined by Q is always non-negative (proof given by writing Q in new coordinates along eigenvectors)
- Formally:

Name	Definition	Notation
Positive semidefinite	all $\lambda_i \geq 0$	$Q \succeq 0$
Positive definite	all $\lambda_i > 0$	$Q \succ 0$
Negative semidefinite	all $\lambda_i \leq 0$	$Q \preceq 0$
Negative definite	all $\lambda_i < 0$	$Q \prec 0$
Indefinite	everything else	(none)

- The second derivative of a function can be written as the *Hessian* matrix $\nabla^2 f(x)$; A quadratic function is **convex** iff its second derivative matrix $\nabla^2 f(x) = Q$ is positive semidefinite

A formal definition of quadratic programming (QP): $\min f(x) := \frac{1}{2} x^T Q x + c^T x$, s.t. $Ax \geq b$.

- Q can be taken as symmetric; Hessian of the function $\nabla^2 f(x) = Q$
- If $Q \succeq 0$, it is a convex QP and solvable
- If $Q \not\succeq 0$, it is very hard to solve in general

A major difference between LP and QP is that optimal solutions for LP appear on boundaries of the feasible region (typically at corner points), but in QP, it may appear anywhere (depends on where is the "bottom of the quadratic bowl").

Least Squares Problem

A classic problem in statistics is **regression**: *fitting* a curve to a collection of data points. The objective is to minimize the error given by square of *Euclidean norm* $||Ax - b||^2 = \sum_{i=1}^{m} (\tilde{a_i}^T x - b_i)^2$. This is called the *least squares problem*.

Also see slides #29 for a perspective from solving a linear equation system. Notice that the function does not have to be restricted in any form (linear, quadratic, etc.). All we need to do is to identify the *coefficients* to decide, and do data calculation to squash it (e.g., if we have a component $a \cos(x)$, just compute all $\cos(x)$ values at the data points and decide on a).

Different types of norm could be used here:

- Minimize the largest residual (a.k.a. the ∞-norm): $||r||_\infty = \max_i |r_i|$, solve using minimax technique of `lp`
- Minimize the sum of absolute values (a.k.a. the 1-norm): $||r||_1 = \sum_i |r_i|$, solve using minimax technique of `lp`
- Minimize the Euclidean norm (a.k.a. the 2-norm, described above): $||r||_2 = ||r|| = \sqrt{\sum_i r_i^2}$, solve using `qcp`

See the `movingAV` model for an application on the *moving average* prediction problem.

Minimum-Norm Least Squares

Linear equations could be **underdetermined**, meaning that A is a wide matrix with fewer constraints than the number of variables x. There will be infinitely many solutions. A possible choice of "optimal" here is to pick the $\hat{x}$ with smallest norm.

- Such $\hat{x}$ must be orthogonal to any $\hat{x} - w$, i.e., $\hat{x}^T(\hat{x} - w) = 0$, where $Aw = b$; Replace $\hat{x} = A^T z$ for some z, and we get the equivalent problem: finding z and $\hat{x}$ s.t. $A\hat{x} = b$ and $A^T z = \hat{x}$
- If A has linearly independent rows, $\hat{x} = A^T(AA^T)^{-1}b$, where $A^\dagger := A^T(AA^T)^{-1}$ is called the *pseudoinverse* of A
- More generally, we can have the optimization problem: $\min_x ||Ax - b||^2$, subject to $Cx = d$
  - If $C = 0, d = 0$, this is the ordinary least squares
  - If $A = I, b = 0$, this is the *minimum-norm* least squares described above

Tradeoffs

We often want to optimize several different objectives simultaneously, but these objectives could be *conflicting*. Suppose we have $J_1 = ||Ax - b||^2$ and $J_2 = ||Cx - d||^2$, and we would like both J_1 and J_2 small. A sensible approach is to solve the problem: $\min_x J_1 + \lambda J_2$, where λ is the **tradeoff** parameter.

- $\lambda \to 0$: we place more weight on J_1
- $\lambda \to \infty$: we place more wieght on J_2

By solving the tradeoff-ed problem with different λ values, we can get a *pareto curve* of the optimum boundary.

See slides #30 and the `hovercraft` model example for a tradeoff between trajectory precision and fuel use.

Regression Machine Learning

The basic form of *machine learning* (ML) is to learn a function f based on *training data* $(x, f(x))$ pairs, then produce predictions $f(x')$ given new data x'. To evaluate an ML model, we need to evaluate over a separate set of *testing data* (otherwise we are cheating). However, a good ML problem means that the testing data come from the same *distribution* (the same data generation source) with the training data -- otherwise we cannot really generalize any mathematical relations under the hood.

There are many different choices of *loss functions*, depending on the problem scenario. We try to minimize the loss function.

Naïve loss functions could be very sensitive to *outliers* in data, which could be just data errors. We do not want to *overfit* to those outliers. One possible technique is to use the *Huber* loss function, where it uses quadratic function when residue is small, but uses linear function when residue is large.

$$\rho(r) = \begin{cases} \frac{1}{2}r^2 & |r| \le \sigma \\ \sigma|r| - \frac{1}{2}\sigma^2 & |r| > \sigma \end{cases}$$

This function cannot be modeled in GAMS. There is a way to re-formulate Huber estimation as a QP problem. Solving the dual of that formulation and the marginal values are the optimal coefficients.

Occam's Razor & Lasso

A classic idea in ML is the **Occam's razor**: smaller models with fewer features tend to generalize better. Hence, instead of doing regression on all of the features, we would like to do *feature selection*, i.e., set the number of features we want and select the best set that gives the best prediction. However, doing so converts the problem to `miqcp`, which are much harder to solve than `qcp` problems.

The *compress sensing* community typically approximates the 0-form (counting) of non-zero features by the 1-norm of features:

$$L = l(\beta) + \lambda||\beta||_1$$

There is a tradeoff -- by increasing λ, we require more "sparsity" in the result vector β.

This 1-norm is equivalent to the *least absolute shrinkage and selection operator* (Lasso), which says $||\beta||_1 \leq T$. It can be proven that choosing a value for λ in the 1-norm formulation is equivalent to choosing some value of T in the Lasso formulation.

The λ parameter here becomes a *hyper-parameter* of the model, and needs to be tuned through *cross-validation*.

Support Vector Machine (SVM)

SVMs have appeared so many times in my notes of other courses. I will omit the definitions here...

SVM trades off errors with the margin between support vectors. See slides #32 for the formulation and examples.

Portfolio Optimization

In *portfolio* optimization, we decide how to invest money, choosing between $i = 1, 2, \ldots, N$ different assets. Each asset can be modeled as a *random variable* Z_i with expected return μ_i and standard deviation σ_i. Our goal is to maximize the total return $R = \sum_{i=1}^{N} x_i Z_i$.

The new thing here is that we cannot optimize random variables. We must calculate the expected value (*mathematical expectation*) of the random variables and do optimization based on that. Expected return is linear:

$$\max \mathbb{E}(R) = \mathbb{E}(\sum_{i=1}^{N} x_i Z_i) = x^T \mu$$

The problem with the above formulation is that it results in going "all in" for the biggest mean, but ignores the standard deviation, i.e., the *risk*. We also want to minimize the total risk:

$$\min Var(R) = x^T \Sigma x, \text{ where } \Sigma := \begin{bmatrix} cov(Z_1, Z_1) & \ldots & cov(Z_1, Z_n) \\ \vdots & \ddots & \vdots \\ cov(Z_n, Z_1) & \ldots & cov(Z_n, Z_n) \end{bmatrix}$$

is called the *variance-covariance matrix*.

- If all assets are *highly-correlated* (i.e., Σ is a matrix of almost all 1's), then total expected return is close to "all in" the one with the highest mean
- If all assets are *not correlated* (i.e., Σ is an identity matrix), then total expected return might be smaller

We can tradeoff between total return and risk by solving the following QP optimization problem:

$$\min_x -x^T \mu + \lambda x^T \Sigma x$$
$$\sum_i x_i = 1$$
$$x_i \geq 0, \forall i \in [1, N]$$

Or, an alternative formulation is to just minimize risk, but having the constraint that the total expected return is no less than a threshold K.

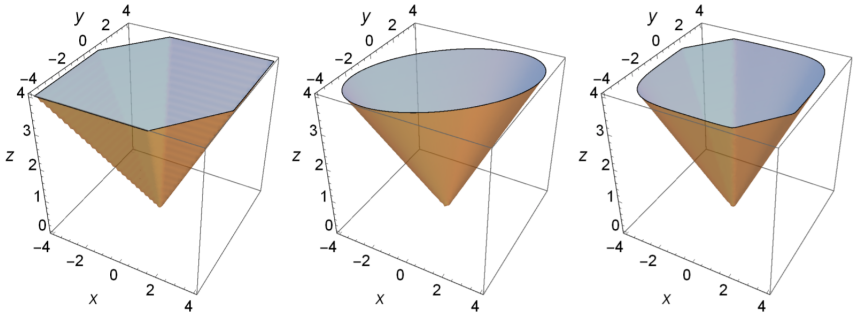
Second-Order Cone Programming (SOCP)

Recall that in LP, a linear constraint is a hyperplane and the feasible region is an intersection of halfspaces. In QP, a quadratic constraint $x^T Q x \leq 1, Q \succ 0$ corresponds to an *ellipsoid*, where on each of the z coordinates, the sphere is stretched by $\frac{1}{\sqrt{\lambda_i}}$, where λ_i is the eigenvalue for that coordinate.

- If some $\lambda_i = 0$, then $Q \succeq 0$ and the ellipsoid is degenerate: on some coordinate, it stretches out to infinity; Degenerate QP is harder for the solvers
- Norm constraints $||Ax - b||^2 \leq c$ are (possibly degenerate) ellipsoids: $A^T A \succeq 0$

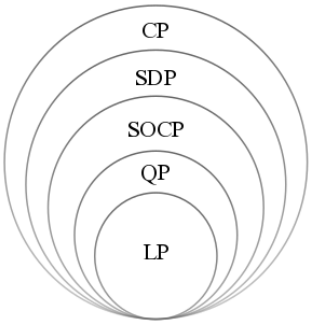
A *second-order cone* is the set of points satisfying $||Ax - b|| \leq c^T x + d$.

- If $A = 0$, it is LP;
- If $c = 0$ and $d \geq 0$, it is an ellipsoid QP;
- In general, its feasible region will be a *slice* of a convex *cone*. We cannot directly square both sides, otherwise we get a nonconvex quadratic constraint like $x_1^2 - x_2^2 \leq 0$. The cone $\mathcal{C}$ is convex if in addition, $x \in \mathcal{C} \wedge y \in \mathcal{C} \Rightarrow x + y \in \mathcal{C}$.



See slides #34 and the `sylvester` model.

Convex programming hierarchy:



Robust Programming

Suppose there is *uncertainty* in some of the a_i vectors for an LP problem, say $a_i = \bar{a_i} + \rho u$, where u is the uncertainty. We want the solution to be feasible given any possible u (which is very conservative).

- *Box* constraint form: feasibility must hold for all u with $||u||_\infty \leq 1$

  This implies the following LP problem:

$$\max_{x,t} c^T x$$
$$\bar{a_i}^T x + \rho \sum_{j=1}^{n} t_j \leq b_i, \forall i$$
$$-t_j \leq x_j \leq t_j, \forall j$$

- *Ball* constraint form: feasibility must hold for all u with $||u||_2 \leq 1$

  This implies the following SOCP problem:

$$\max_{x} c^T x$$
$$\bar{a_i}^T x + \rho ||x||_2 \leq b_i, \forall i$$

Stochastic Programming

In general real-world cases, we do not know the exact demand. Ways to approach real-world problem:

- Deterministic programming: do not include uncertainty, instead pay more attention on data collection and make assumptions
- Robust programming: include an uncertainty variable and constraint feasibility to hold under all possible uncertainty, like demonstrated above
- Stochastic programming: model variables and random variables and involve probability of different **scenarios**

The key idea is to separate *first-stage* variables (quantities that must be decided here and now, typically the decision variables and values derived directly from decision variables) from *second-stage* variables (quantities with uncertainty, e.g., demands, that may be different across different scenarios). Then, write out the program just like a normal program for a certain scenario, but in the final objective function, sum up the second-stage parts as expectation.

Details about stochastic programming omitted here, see slides >= #36.