

Goal

Aayush Bajaj

July 10, 2026

Contents

1	the deviation-variable device	1
1.1	complementarity for free (and when it is not)	2
2	weighted (archimedean) goal programming	2
2.1	the normalisation trap	3
3	preemptive (lexicographic) goal programming	3
4	goal programming as scalarisation	4
5	a workshop example	4
5.1	the lexicographic solve	5
5.2	the weighted solve, three ways	6
6	see also	8

most optimisation asks for the *best*; goal programming asks for *good enough*, several times over. you attach a numeric target to each of several objectives, measure how far the plan misses each target, and minimise the misses you dislike. the philosophy is herbert simon’s **satisficing**¹ — real decision makers do not maximise a grand utility function, they set aspiration levels and stop when they are met — and the machinery is pure **linear programming**: goal programming was invented by charnes and cooper as an LP device² and remains the most-used technique in practical multi-criteria decision making precisely because it never leaves LP territory.

the deviation-variable trick first appeared in a 1955 charnes-cooper-ferguson paper on executive pay; the name arrived with their 1961 book

1 the deviation-variable device

hard constraints define a feasible set X ; goals are soft. for each goal i with criterion function $g_i(x)$ and target t_i , introduce two non-negative **deviation variables** and convert the aspiration into an equality:

$$g_i(x) + d_i^- - d_i^+ = t_i, \quad d_i^-, d_i^+ \geq 0. \quad (1)$$

¹herbert simon, *rational choice and the structure of the environment*, psychological review 63(2), 1956 — aspiration levels and the coining of “satisficing”; the argument that bounded agents stop at good-enough runs through his 1947 *administrative behavior*.

²a. charnes, w. w. cooper and r. o. ferguson, *optimal estimation of executive compensation by linear programming*, management science 1(2), 1955, introduced the deviation variables; the term “goal programming” first appears in charnes and cooper’s *management models and industrial applications of linear programming* (wiley, 1961).

here d_i^- absorbs **underachievement** (the criterion falls short of the target) and d_i^+ absorbs **overachievement**. which deviation is *unwanted* depends on the sense of the goal:

- **at least** t_i (a profit floor): penalise d_i^- ; overshooting is free or welcome.
- **at most** t_i (an overtime ceiling): penalise d_i^+ .
- **exactly** t_i (a staffing level): penalise both.

the generic goal program is then

$$\begin{aligned} & \text{minimise} && h(d^-, d^+) \\ & \text{subject to} && g_i(x) + d_i^- - d_i^+ = t_i \quad i = 1, \dots, m \\ & && x \in X, \quad d^-, d^+ \geq 0, \end{aligned}$$

where the achievement function h is some non-decreasing combination of the unwanted deviations — a weighted sum, a lexicographic vector, or a max, giving the three classical variants below. when the g_i are linear and X is polyhedral, everything is an LP.

1.1 complementarity for free (and when it is not)

the equality only pins the *difference* $d_i^+ - d_i^- = g_i(x) - t_i$, so the pair $(d_i^- + \delta, d_i^+ + \delta)$ is feasible for any $\delta \geq 0$. the deviations mean what you want them to mean — $d_i^- = \max(0, t_i - g_i(x))$ — only when $d_i^- d_i^+ = 0$. nobody adds that (non-linear!) condition explicitly, for two reasons:

- **both weights positive**: if $d_i^- > 0$ and $d_i^+ > 0$ at a claimed optimum, subtract $\delta = \min(d_i^-, d_i^+)$ from both. feasibility is untouched and the objective drops by $\delta(w_i^- + w_i^+) > 0$ — contradiction. so any optimum is complementary.
- **one weight zero** (a one-sided goal): the argument fails and alternative optima with both deviations positive exist. but the two columns of d_i^- and d_i^+ in the constraint matrix are negatives of each other, so no *basis* can contain both — every vertex solution is complementary anyway. a simplex solver (or barrier with crossover) is safe.

a pure interior-point solve without crossover returns the analytic centre of the optimal face and can happily report both deviations positive

the robust habit: report deviations recomputed from x as $\max(0, t_i - g_i(x))$, never trust the raw solver values of unpenalised deviation variables. the same caution applies at intermediate stages of the lexicographic method, where lower-priority deviations are not yet in any objective.

2 weighted (archimedean) goal programming

collapse everything into one LP:

$$\text{minimise} \quad \sum_{i=1}^m \left(\frac{w_i^- d_i^- + w_i^+ d_i^+}{k_i} \right) \quad (2)$$

with importance weights $w_i^\pm \geq 0$ (zero on wanted deviations) and normalisation constants k_i . called *archimedean* because every goal trades against every other at a finite rate — the axiom of archimedes, no quantity infinitely larger than another.

2.1 the normalisation trap

the k_i are not optional decoration. raw deviations are **incommensurable**: d_1 is in dollars, d_2 in labour-hours, and a unit-weighted sum silently asserts one dollar = one hour. worse, the assertion changes if you re-express a goal in cents. standard normalisations:

- **percentage**: $k_i = t_i$, so each term is a fractional miss. intuitive, dimensionless, but undefined for $t_i = 0$ and misleading for targets near zero.
- **euclidean**: for a linear goal $a_i^\top x$, take $k_i = \|a_i\|_2$. geometrically honest — deviations become distances in decision space — but the numbers no longer mean anything to the decision maker.
- **range (zero-one)**: k_i = the spread of g_i between its best and worst values over X . robust, at the cost of $2m$ auxiliary optimisations to find the ranges.

the worked example below shows the trap numerically: the same problem under raw and percentage weights picks *different vertices*. the weights are part of the model, not a tuning afterthought.

a third variant worth knowing: **chebyshev (minmax) goal programming** minimises the largest normalised deviation, $\min D$ subject to $w_i d_i / k_i \leq D$ — it seeks balance between goals rather than the best total, and unlike the weighted sum it can reach efficient points off the convex hull of the criteria space.

3 preemptive (lexicographic) goal programming

often the decision maker refuses finite trade-offs: safety before profit before polish, and no amount of polish buys back a safety miss. assign goals to priority levels $P_1 \gg P_2 \gg \dots \gg P_L$ and minimise the achievement *vector*

$$\text{lexmin } (h_1(d), h_2(d), \dots, h_L(d)), \quad (3)$$

comparing vectors lexicographically: h_1 decides; ties go to h_2 ; and so on.³ textbooks write this as minimising $P_1 h_1 + P_2 h_2 + \dots$ with $P_k \gg P_{k+1}$, but the P_k are ordering symbols, not numbers. for LPs a large-enough finite weight ratio does reproduce the lexicographic optimum — the threshold depends on the instance data — yet it is unknowable in advance and numerically brutal, so the honest implementation is **sequential**:

- **stage 1**: minimise h_1 over the goal-augmented feasible set; record the optimum h_1^* .
- **stage k**: minimise h_k subject to the original constraints *plus* $h_j(d) \leq h_j^*$ for all $j < k$ — earlier achievements are locked in as constraints, never renegotiated.
- **stop**: after level L , or early if a stage has a unique optimum (nothing left to decide).

³the preemptive formulation was systematised by ijiri (1965) and given its modern sequential-LP treatment in ignizio, *goal programming and extensions* (lexington, 1976); the pareto-efficiency critique, the restoration repair and the sufficient condition “all unwanted deviations positive at the optimum implies efficiency” are surveyed in romero, *handbook of critical issues in goal programming* (pergamon, 1991) and tamiz, jones and romero, *goal programming for decision making: an overview of the current state-of-the-art*, european journal of operational research 111(3), 1998.

each stage is a plain LP, one extra row per completed level. two behavioural notes: the stage-1 minimiser is generally *not* the final answer (any point meeting the top goal ties at that stage, and the solver returns an arbitrary one); and a goal that ends up unmet at level k does not make lower levels moot — they still choose among the level- k -optimal points.

4 goal programming as scalarisation

against the general **multi-objective** problem, goal programming is one more **scalarisation**: a rule collapsing a vector of criteria into a single solvable objective. its distinguishing input is the target vector t , and therein lies the classic critique:

the critique and its repairs are the through-line of romero's 1991 monograph and the tamiz-jones-romero survey, ejor 1998

- if t is **utopian** (unreachable in every penalised direction), every unwanted deviation is strictly positive at the optimum, any dominating point would strictly lower the objective, and the GP solution is pareto-efficient.
- if t is **pessimistic** — set inside the region where all goals can be met — then *every* point of the zero-deviation set scores a perfect $h = 0$. the solver returns an arbitrary one, which may be badly dominated: same zero misses, strictly worse criteria. satisficing taken too literally leaves money on the table.

the standard repair is a **restoration** (efficiency-recovery) pass: after solving, fix all unwanted deviations at their achieved values and maximise a combination of the *wanted* deviations. this slides the solution along the optimal set to a pareto-efficient point without disturbing the achieved goals. (rewarding wanted deviations with negative weights in the original objective also works, but it breaks the complementarity argument above and risks unboundedness — the two-phase version is safer.) the worked example demonstrates the trap and the repair with real numbers.

5 a workshop example

a workshop assembles two products. per unit, product a needs 2 labour-hours, 1 machine-hour and 1 kg of stock, clearing \$6; product b needs 1 labour-hour, 3 machine-hours and 1 kg, clearing \$4. exactly 12 kg of stock arrives per day — a **hard** constraint. management states three goals in strict priority order:

- **P1 (profit)**: clear at least \$48 a day — minimise the shortfall d_1^- .
- **P2 (labour)**: stay within the 14 rostered labour-hours — minimise overtime d_2^+ .
- **P3 (machine)**: finish within the 20 machine-hours before the nightly maintenance window — minimise the overrun d_3^+ .

with $x, y \geq 0$ the daily production of a and b, the goal-augmented system is

$$\begin{aligned} 6x + 4y + d_1^- - d_1^+ &= 48 \\ 2x + y + d_2^- - d_2^+ &= 14 \\ x + 3y + d_3^- - d_3^+ &= 20 \\ x + y &\leq 12, \quad x, y, d^\pm \geq 0. \end{aligned}$$

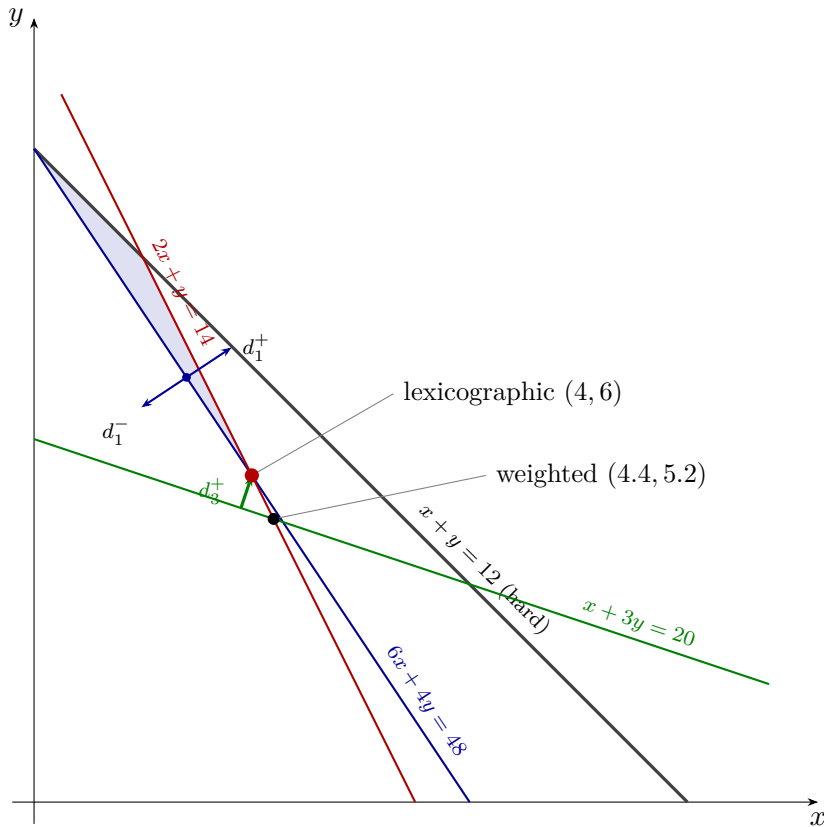


Figure 1: decision space for the workshop problem: three goal lines and the hard stock constraint. the shaded sliver satisfies the top two priorities (profit met, no overtime); the lexicographic point is its corner of least machine use, still 2 hours over the machine target — the short arrow is the residual deviation. the weighted (percentage-normalised) optimum trades a little profit to meet the machine goal exactly.

5.1 the lexicographic solve

the sequential method, honestly, as three `scipy.optimize.linprog` stages — each stage appends the previous level's achievement as a new constraint row:

```

1  import numpy as np
2  from scipy.optimize import linprog
3
4  # variables: z = [x, y, d1m, d1p, d2m, d2p, d3m, d3p]
5  A_eq = [[6, 4, 1, -1, 0, 0, 0, 0], # profit: 6x + 4y + d1m - d1p = 48
6          [2, 1, 0, 0, 1, -1, 0, 0], # labour: 2x + y + d2m - d2p = 14
7          [1, 3, 0, 0, 0, 0, 1, -1]] # machine: x + 3y + d3m - d3p = 20
8  b_eq = [48, 14, 20]
9  A_ub = [[1, 1, 0, 0, 0, 0, 0, 0]] # material: x + y <= 12 (hard)
10 b_ub = [12]
11 bounds = [(0, None)] * 8
12
13 # priority levels: P1 min d1m, P2 min d2p, P3 min d3p
14 levels = [("P1: profit shortfall d1-", [0, 0, 1, 0, 0, 0, 0, 0]),
15           ("P2: labour overtime d2+", [0, 0, 0, 0, 0, 1, 0, 0]),
16           ("P3: machine overrun d3+", [0, 0, 0, 0, 0, 0, 1, 0])]
17
18 extra_A, extra_b = [], []
19 for name, c in levels:
20     res = linprog(c, A_ub=A_ub + extra_A, b_ub=b_ub + extra_b,
21                 A_eq=A_eq, b_eq=b_eq, bounds=bounds, method="highs")
22     print(f"{name}: attained {res.fun:.4f} at (x, y) = ")

```

```

23         f"({res.x[0]:.4f}, {res.x[1]:.4f})")
24     extra_A.append(c)           # lock this level's achievement in
25     extra_b.append(res.fun + 1e-9) # as a constraint for later stages
26
27 x, y = res.x[:2]
28 print(f"\nfinal plan: x = {x:.4f}, y = {y:.4f}")
29 print(f"profit 6x+4y = {6*x+4*y:.2f} (target 48)")
30 print(f"labour 2x+ y = {2*x+y:.2f} (target 14)")
31 print(f"machine x+3y = {x+3*y:.2f} (target 20)")
32 print(f"material x+ y = {x+y:.2f} (limit 12)")

```

P1: profit shortfall d1-: attained 0.0000 at (x, y) = (0.0000, 12.0000)
P2: labour overtime d2+: attained 0.0000 at (x, y) = (4.0000, 6.0000)
P3: machine overrun d3+: attained 2.0000 at (x, y) = (4.0000, 6.0000)

final plan: x = 4.0000, y = 6.0000
profit 6x+4y = 48.00 (target 48)
labour 2x+ y = 14.00 (target 14)
machine x+3y = 22.00 (target 20)
material x+ y = 10.00 (limit 12)

reading the stages:

- **stage 1** attains zero shortfall — the profit goal is reachable — but at (0,12), an arbitrary member of the zero-shortfall set (make only product b, flood the machine). intermediate stage answers are ties, not decisions.
- **stage 2**, with $d_1^- \leq 0$ locked in, also attains zero: the shaded sliver of the figure, with corners (4,6), (2,10), (0,12), meets the profit floor without overtime.
- **stage 3** cannot reach zero. minimising machine use $x + 3y$ over the sliver gives 22 at the corner (4,6) — an irreducible overrun of $d_3^+ = 2$ hours, since the corner values are 22, 26 and 36.

the plan: 4 units of a, 6 of b. profit lands exactly on \$48, labour exactly on 14 hours, the machine runs 2 hours into the maintenance window, and 2 kg of stock is left over — the hard constraint is slack, which is fine; hard constraints bound, goals aspire. note what preemption means here: no finite amount of machine relief could have bought even a cent of profit shortfall.

5.2 the weighted solve, three ways

same system, archimedean objective, three weightings — plus the pessimistic-target trap and its restoration repair:

```

1  import numpy as np
2  from scipy.optimize import linprog
3
4  # A_eq, b_eq, A_ub, b_ub, bounds as above
5
6  def solve(c, b_eq=b_eq, tag=""):
7      r = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq,
8                  bounds=bounds, method="highs")
9      x, y = r.x[:2]
10     print(f"{tag}: obj = {r.fun:.6f}, (x, y) = ({x:.4f}, {y:.4f})")
11     print(f"    profit {6*x+4*y:.2f} | labour {2*x+y:.2f} | machine
12           ↪ {x+3*y:.2f}")
13     return r

```

```

14 # raw weights: one dollar of shortfall trades 1:1 against one hour
15 solve([0, 0, 1, 0, 0, 1, 0, 1], tag="raw w = (1, 1, 1)")
16
17 # percentage-normalised weights: deviations as fractions of target
18 solve([0, 0, 1/48, 0, 0, 1/14, 0, 1/20], tag="norm w = (1/48, 1/14, 1/20)")
19
20 # pessimistic profit target (36 instead of 48): pareto trap
21 b_lo = [36, 14, 20]
22 r = solve([0, 0, 1/36, 0, 0, 1/14, 0, 1/20], b_eq=b_lo,
23           tag="lazy profit target t1 = 36")
24
25 # restoration pass: keep unwanted deviations at their optima (all zero),
26 # maximise the wanted deviation d1+ (profit above target)
27 A_lock = A_ub + [[0, 0, 1, 0, 0, 0, 0, 0],
28                  [0, 0, 0, 0, 0, 1, 0, 0],
29                  [0, 0, 0, 0, 0, 0, 0, 1]]
30 b_lock = b_ub + [r.x[2] + 1e-9, r.x[5] + 1e-9, r.x[7] + 1e-9]
31 r2 = linprog([0, 0, 0, -1, 0, 0, 0, 0], A_ub=A_lock, b_ub=b_lock,
32             A_eq=A_eq, b_eq=b_lo, bounds=bounds, method="highs")
33 x, y = r2.x[:2]
34 print(f"restored: d1+ = {r2.x[3]:.2f}, (x, y) = ({x:.4f}, {y:.4f})")
35 print(f"    profit {6*x+4*y:.2f} | labour {2*x+y:.2f} | machine {x+3*y:.2f}")

```

```

raw w = (1, 1, 1): obj = 0.285714, (x, y) = (4.5714, 5.1429)
    profit 48.00 | labour 14.29 | machine 20.00
norm w = (1/48, 1/14, 1/20): obj = 0.016667, (x, y) = (4.4000, 5.2000)
    profit 47.20 | labour 14.00 | machine 20.00
lazy profit target t1 = 36: obj = 0.000000, (x, y) = (2.0000, 6.0000)
    profit 36.00 | labour 10.00 | machine 20.00
restored: d1+ = 11.20, (x, y) = (4.4000, 5.2000)
    profit 47.20 | labour 14.00 | machine 20.00

```

four lessons, one per solve:

- **raw weights** pick (32/7, 36/7): profit and machine hit exactly, and the whole miss is dumped on labour as $d_2^+ = 2/7$ of an hour. why labour? at a 1:1 exchange rate, $2/7 \approx 0.29$ “units” of overtime is cheaper than \$0.80 of profit shortfall or 2 machine-hours. the model just asserted that an hour and a dollar are the same thing.
- **percentage weights** pick (4.4, 5.2): sacrifice \$0.80 of profit — a 1.7% miss against its target — to hit labour and machine exactly, since $0.8/48 \approx 0.017$ beats $(2/7)/14 \approx 0.020$ beats $2/20 = 0.1$. different normalisation, different vertex.
- compare both with the **lexicographic** answer (4, 6), which refused to trade any profit at all and ate the 2-hour machine overrun. three achievement functions, three plans, one feasible set: the weights *are* the model.
- **the pessimistic-target trap**: with the profit target slackened to \$36, every goal is comfortably attainable, the objective bottoms out at zero, and the solver returns (2, 6) — profit \$36, zero overtime, machine exactly at target. nothing in the model objects, yet (4.4, 5.2) earns \$47.20 with the *same* zero unwanted deviations: the returned point is dominated in everything the decision maker actually cares about. the **restoration pass** — lock the unwanted deviations at zero, maximise the wanted d_1^+ — recovers it, banking $d_1^+ = 11.2$ of free profit.

6 see also

- **linear programming** — the solver technology underneath every variant here
- **multi-objective** — the general vector-optimisation problem goal programming scalarises
- **integer programming** — goal programs over discrete decisions; same deviation device, harder solves
- **constraint** — when every requirement really is hard
- **robust** — a different response to not trusting a single objective